Prototyping Parallel Algorithms with ProSet-Linda[*]

Wilhelm Hasselbring

University of Essen
Fachbereich Mathematik und Informatik — Software Engineering
Schützenbahn 70, 45117 Essen, Germany
willi@informatik.uni-essen.de

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 called generative communication, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly.
This paper presents ProSet-Linda which adapts the concept for process creation via Multilisp's futures to set-oriented programming and integrates Linda's concept for synchronization and communication via tuple space. This new approach to integrating futures and generative communication into a prototyping language extends the basic Linda model with multiple tuple spaces, the notion of limited tuple spaces, selection and customization for matching, specified fairness of choice, and the facility for changing tuples in tuple space.
The subject of this paper is the construction of prototypes and not the transformation of prototypes into production-quality programs. Therefore, we consider only the early phases in the process of software construction.

1 Introduction

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

As has been observed [5], no matter how effective the system software and hardware of a parallel machine are at delivering performance, it is only from

[*] In: J. Volkert: Parallel Computation (Proc. Second International ACPC Conference), Lecture Notes in Computer Science, Vol. 734, Springer-Verlag, pages 135–150, October 1993, Gmunden, Austria.

new algorithms that orders of magnitude improvements in the complexity of a problem can be achieved:

> "An idea that changes an algorithm from n^2 to $n \log n$ operations, where n is proportionate to the number of input elements, is considerably more spectacular than an improvement in machine organization, where only a constant factor of run-time is achieved." [5, page 250]

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

Current programming environments for distributed memory architectures provide inadequate support for mapping applications to the machine. In particular, the lack of a global name space forces algorithms to be specified at a relatively low level. This greatly increases the complexity of programs, and also stipulates algorithm design choices, inhibiting experimentation with alternate algorithm choices or problem decompositions.

Process communication and synchronization in Linda is reduced to concurrent access to a large data pool, thus relieving the programmer from the burden of having to consider all process inter-relations explicitly. The parallel processes are decoupled in time and space in a very simple way: processes do not have to execute at the same time and in the same address space. 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 (Sect. 3). 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. The tuple space is the fundamental medium of communication in Linda.

Linda and ProSet both provide tuples thus it is quite natural to combine set-oriented programming with generative communication on the basis of this common feature to form a tool for prototyping parallel algorithms.

Programming in Linda provides a spatially and temporally unordered bag of processes. Each task in the computation can be programmed (more-or-less) independently of any other task. This enables the programmer to focus on one process at a time thus making parallel programming conceptually the same order of problem-solving complexity as conventional, sequential programming. 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 be able to reflect a particular top-down approach to building software, and not a particular machine architecture. This is also important to support portability across different machine architectures. Implementations of Linda have been performed on a wide variety of parallel architectures: on shared-memory multi-processors as well as on distributed memory architectures [3, 26]. 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.

C was the first computation language in which Linda has been integrated [11]. Meanwhile there exist also integrations into higher-level languages supporting the object-oriented, functional, and logic programming paradigm, respectively [26]. The present paper presents the combination with a set-oriented language, where process creation via Multilisp's futures is adapted to set-oriented programming and combined with the concept for synchronization and communication via tuple spaces. We regard tuple spaces primary as a device for synchronization and communication between processes, and only secondary for process creation.

Sections 2 and 3 provide brief introductions to the prototyping process and to the language ProSet, respectively. In Sect. 4 the combination of ProSet with Linda will be presented. We refer to [4] for a full account to programming with C-Linda. Essential enhancements to the basic Linda model for this combination are multiple tuple spaces, the notion of limited tuple spaces, selection and customization for matching, specified fairness of choice, and the facility for changing tuples in tuple space. Section 5 sketches implementation issues and Sect. 6 draws some conclusions.

Henri Bal, Mike Factor, Jerry Leichter and Greg Wilson provided useful comments and suggestions on various aspects of this work. The comments on drafts of this paper by Ernst-Erich Doberkat are gratefully acknowledged.

2 The Prototyping Process and Parallel Programming

One of the more recent approaches for complementing the classical model of software production using the life cycle approach is rapid prototyping. Prototyping refers to the well defined phase in the production process of software in which a model is constructed which has all the essential properties of the final product, and which is taken into account when properties have to be checked, and when the further steps in the development have to be determined [10]. We want to note that a prototype is a model, and that this model taken as a program has to be executable so that at least part of the functionality of the desired end product may be demonstrated on a computer. Prototyping has been developed as an answer to deficiencies in the waterfall model, but it should not be considered as an alternative to this model. It is rather optimally useful when it complements the waterfall model. It is plausible that prototyping may be used during the early phases of the design.

The idea of prototyping is being adopted in software engineering for different purposes: prototypes are used *exploratively* to arrive at a feasible specification, *experimentally* to check different approaches, and *evolutionary* to build a system incrementally. Our approach to prototyping is an evolutionary development in versions. The prototype evolves in accordance with the changing environment. The linear ordering of development steps in the classical waterfall model is mapped here into successive development cycles. This implies that the users are involved in the system development process which supports the communication between users and developers.

ProSet also contains a Pascal-like subset that facilitates prototyping by allowing a program to be refined into successively finer detail while staying within the language. Prototypes should be built in very high level languages to make them rapidly available in the early phases of the production process. To be useful, prototypes must be built rapidly and designed in such a way that they can be modified rapidly. Consequently, a prototype is usually not a very efficient program since the runtime system has a heavy burden for executing the highly expressive constructs. To obtain a more efficient production-level version program transformations are desirable to refine the prototype design into a production-quality product [22]. The subject of this paper is the construction of prototypes and not the transformation of prototypes into production-quality programs. Therefore, we consider only the early phases in the process of software construction.

Prototyping means constructing a model. Since applications which are inherently parallel should be programmed in a parallel way, it is most natural to incorporate parallelism into the process of model building. Opportunities for automatic detection of parallelism in existing programs are limited and furthermore, in many cases the formulation of a parallel program is more natural and appropriate than a sequential one. Most systems in real life are of a parallel nature, thus the intent for integrating parallelism into a prototyping language is not only increased performance. If one wants to model an inherently parallel system, it is reasonable to have features for specifying (coarse-grained) processes that communicate and synchronize via a simple communication medium, and not to force such inherent parallelism into sequences. Our work intends to provide a tool for prototyping parallel algorithms and modeling parallel systems.

3 The Prototyping Language ProSet

The procedural, set-oriented language ProSet [8] is a successor to SETL [24]. This section will present a brief introduction to data and control structures of the language and a short example. The high-level structures that ProSet provides qualify the language for prototyping. For a full account to prototyping with set-oriented languages we refer to [7].

ProSet provides data types for atom, integer, real, string, Boolean, tuple, set, function, and module values. It is a *higher-order* language, because functions and modules have first-class rights. ProSet is weakly typed, i.e., the type of an object is in general not known at compile time. 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. Tuples and sets are compound data structures, which may be heterogeneously composed. Sets are unordered collections while tuples are ordered. There is also the undefined value `om` which indicates, e.g., selection of an element from an empty set. As an example consider the expression `[123, "abc",` `true, {1.4, 1.5}]` which creates a tuple consisting of an integer, a string, a Boolean, and a set of two reals. This is an example of what is called a *tuple former.* As another example consider the set forming expression `{2*x: x in [1..10] | x>5}` which yields the set `{12, 14, 16, 18, 20}`. 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 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 the `for` and `whilefound` loops which are custom tailored for iteration over the compound data structures. The quantifiers ($\exists$, $\forall$) of predicate calculus are provided.

In Fig. 1 a solution for the so-called *Queens' Problem* in ProSet 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. Our program does not solve the above problem directly. It prints out the set of all positions in which the n queens do not attack each other. If it is not possible to place n queens in non-attacking positions, this set will be empty. We denote fields on the chessboard by pairs of natural numbers for convenience (this is unusual in chess, where characters are used to denote the columns). `[1,1]` denotes the lower left corner.

Note that there are no explicit loops and that there is no recursion in the program. All iterations are done implicitly. One may regard this program also as a (executable) specification of the Queens' Problem.

4 Parallel Programming in ProSet

The following subsections will present and discuss process creation and tuple-space communication in ProSet.

4.1 Process Creation

In C-Linda [4] 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 C-Linda's `eval` operation 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.

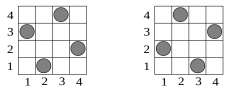
```
program Queens;
   constant N := 4;
begin
   fields := {[x,y]: x in [1..N], y in [1..N]};
   put({NextPos: NextPos in npow(N, fields) | NonConflict(NextPos)});

   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)))));

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

Fig. 1. Solution for the Queens' Problem.
The predefined function `npow(k, s)` yields the set of all subsets of the set `s` which contain exactly `k` elements. The predefined function `abs` returns the absolute value of its argument. `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`.
This program produces this set as a result: `{{[1,3], [2,1], [4,2], [3,4]}, {[3,1], [1,2], [2,4], [4,3]}}`

which corresponds to these positions:



Multilisp augments Scheme with the notion of *futures* where the programmer needs no knowledge about the underlying process model, inter-process communication, or synchronization to express parallelism. We refer to [14] for a full account to Multilisp. The semantics of futures is based on *lazy evaluation*, which means that an expression is not evaluated until its result is needed.

Futures in Multilisp provide a method for process creation but not much help for synchronization and communication between processes. The only synchronization and communication mechanism is 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*.

Process creation in ProSet is provided through the unary operator `||`, which may be applied to an expression (preferably a function call). A new process will be spawned to compute the value of this expression concurrently with the spawning process analogously to futures in Multilisp. If this *process creator* `||` is applied to an expression that is assigned to a variable, the spawning process continues execution without waiting for the termination of the newly spawned process. At any time the *value* of this variable is needed, the requesting process will be suspended until the future resolves (the corresponding process terminates) thus allowing concurrency between the *computation* and the *use* of a value. Consider the following statement sequence to see an example:

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

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 to a variable, we do not need the values of the enumerated components immediately, thus allowing concurrency as above. Additionally, statements such as "`|| p();`", which spawn new processes, are allowed.

4.2 Tuple-Space Operations

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

Depositing Tuples. The `deposit` operation deposits a tuple into a specified tuple space. We distinguish between passive and active tuples in tuple space. If there are no executing processes in a tuple, then this tuple is added as a passive one (cp. `out` of C-Linda [4]):

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

`TS` is the tuple space at which the specified tuple has to be deposited. See Sect. 4.4 for a discussion of multiple tuple spaces in ProSet. If there are executing processes in a tuple, then this tuple is added as an active one to the tuple space:

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

Depositing a tuple into a tuple space does not touch the value. 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 (see the following subsections).

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.

In ProSet, the predefined exception `ts_is_full` will be raised by default when no memory is available for a `deposit` operation. This exception is raised with the `signal` statement of ProSet. Signal exceptions permit the operation raising the exception to be either terminated or resumed at the handler's discretion. We refer to [13] for a discussion of exception handling in general and to [8] for a discussion of exception handling 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. If the associated handler then executes a `return` statement, the statement following the `deposit` will be executed and the tuple of the respective `deposit` will not be deposited. If the handler executes a `resume` statement, then the `deposit` operation tries again to deposit the tuple.

Optionally, the programmer may specify that a `deposit` operation will be suspended on a full tuple space until space is available again. The suitable handling of full tuple spaces depends on the application 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.

Fetching Tuples. A `fetch` operation tries to fetch and remove exactly one tuple from a tuple space. It is possible to specify several templates for the specified tuple space in a statement, but only one template may be selected nondeterministically (see also Sect. 4.3). We start with a first example for a `fetch` operation with a single template:

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

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

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

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

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

Here both templates consist of an *actual* (the expression `"name"`), a so-called *formal* preceded by a question mark, and a template condition. The template lists are enclosed in parentheses and not in brackets in order to set the templates apart from tuples. The `else` statement will be executed, if none of the templates matches. We will use the notion *non-blocking matching* if `else` statements are specified as opposed to *blocking matching* if no `else` statements are specified.

Meeting Tuples. The `meet` operation *meets* and leaves one tuple in tuple space. It is possible to change the tuple while meeting it. Except for the fact that a `meet` operation, which does not change the met tuple, 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 [1]. 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. Consider

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

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

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

If there are `into`s specified after the formals as in this example 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.

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

4.3 Nondeterminism and Fairness while Matching

There are two sources for nondeterminism while matching:

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

If in any case there is only one candidate available, this one will be selected. There are several ways for handling fairness while selecting tuples or templates that match if there are multiple candidates available. We will now discuss *fairness of choice* which is important for handling the nondeterminism derived from matching. There exist some fairness notions [18]. Weak fairness means that, if a process is enabled continuously from some point onwards then it eventually will be selected. Weak fairness is also called *justice*. Strong fairness means that, if a process is enabled infinitely often then it will be selected infinitely often. 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. Fairness is also important for processes which are blocked on full tuple spaces:

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

In cases (2.) and (3.) processes are involved and enabled after selection, whereas in case (1.) this is not the case for deposited tuples. Therefore, it is reasonable to employ weakly fair selection in cases (2.) and (3.), and unfair selection in case (1.). These fairness properties are specified formally by means of temporal logic in [16].

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. 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. We see no justification to guarantee strong fairness.

4.4 Multiple Tuple Spaces

Atoms are used to identify tuple spaces. As mentioned in Sect. 3 atoms are unique for one machine and across machines. They have first-class rights.

ProSet provides several library functions for handling multiple tuple spaces dynamically. The function `CreateTS(limit)` creates a new tuple space and returns its identity (an atom). Since one has exclusive access to a fresh created tuple-space identity, `CreateTS` supports information hiding. 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. The function `ExistsTS(TS)` yields `true`, if `TS` is an atom that identifies an existing tuple space; else `false`. The function `ClearTS(TS)` removes all active and passive tuples from the specified tuple space. 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. The function `RemoveTS(TS)` calls `ClearTS(TS)` and removes `TS` from the list of existing tuple spaces.

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. 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 in ProSet is done via a separate mechanism for handling persistent data objects [6].

4.5 The Queens' Problem Revisited

In Sect. 3 the Queens' Problem was introduced together with a sequential solution. In Fig. 2 a parallel solution based on the master-worker model is given. It is recommended to examine the sequential solution in Fig. 1 again. In a master-worker application, the task to be solved is partitioned into independent subtasks. These subtasks are placed into the tuple space, and each process in a pool of identical workers then repeatedly retrieves a subtask description from the tuple space, solves it, and deposits the solutions into the tuple space. In our example, these subtasks are the possible positions. The master process then can collect the results. Among the advantages of this programming approach are load balancing and transparent scalability.

4.6 Discussion

This section discusses some design issues for the presented language constructs. A more detailed discussion of these and other issues may be found in [16].

Process Creation. The `deposit` operation comprises the `out` and `eval` operations of C-Linda [4]. 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 [4]. 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. Furthermore, 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.

Extending the Type System. 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 [19] 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.

```
program ParallelQueens;
   constant N := 8, NumWorker := argv(2), -- program argument
            WORK := CreateTS (om),        -- no limit specified
            RESULT := CreateTS (om);
begin
  for i in  [1 .. NumWorker] do      -- spawn the worker processes
     deposit [ || Worker(WORK,RESULT) ] at WORK end deposit;
  end for;
  deposit [ {} ] at RESULT end deposit;  -- initialize the result set
  deposit [ 0 ] at RESULT end deposit;   -- initialize the counter

  AllPositions := npow(N, {[x,y]: x in [1..N], y in [1..N]});
  for NextPosition in AllPositions do
     deposit [ NextPosition ] at WORK end deposit;
  end for;

  fetch ( #AllPositions ) at RESULT end fetch; -- wait for the workers
  fetch ( ? ResultPos |(type $(1) = set) ) at RESULT end fetch;
  put (ResultPos);
  ClearTS(WORK);  -- terminate the worker processes

  procedure Worker (MyWORK, MyRESULT); begin
     loop
        fetch ( ? MyPosition ) at MyWORK end fetch;
        if NonConflict (MyPosition) then
           -- add the position:
           meet ( ? into ($(1) with MyPosition) |(type $(1) = set) )
             at MyRESULT
           end meet;
        end if;
        -- increase the counter of evaluated positions:
        meet ( ? into ($(1) + 1) |(type $(1) = integer) )
          at MyRESULT
        end meet;
     end loop;
  end Worker;
end ParallelQueens;
```

Fig. 2. Parallel Solution for the Queens' Problem.
See Fig. 1 for the procedure `NonConflict`. The resulting set of non-conflicting positions is built up in tuple space `RESULT` via changing `meet` operations. The master program spawns `NumWorker` worker processes. This number is an argument to the main program. The counter in tuple space `RESULT` is necessary to let the master wait until all positions are evaluated. The unary operator `#` returns the number of elements in a compound data structure. The binary operator `with` adds an element to a compound data structure. Tuple space `WORK` is cleared after work has been done thus also terminating the workers.

Multiple Tuple Spaces. Multiple tuple spaces allow the programmer to partition the communication medium as he sees fit. The representation of individual tuple spaces can be customized based on their contents and usage. Compile-time analysis is simplified with respect to partioning of tuple space, and modularity and information hiding is supported. The idea of splitting the tuple space into multiple spaces is frequently applied. New data types and classes are often proposed to organize them [26]. In most proposals for introducing multiple tuple spaces, values of objects of these types are used as identifiers/references to tuple spaces and not as the value of a tuple space itself. Conversely, in [12] operations on tuple spaces as first-class objects are supported (e.g. suspension). However, because of concurrent access it is rarely possible to make any sensible statement with respect to the *actual value* of a tuple space (tuple-space constants make not much sense, except for the creation of tuple spaces). 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.

5 Implementation Issues

The definition of C-Linda has been presented informally [11] and, as a result, has included several ambiguities. E.g., [20] summarizes four basic types of process creation used in implementations of C-Linda's `eval` operation. These are different interpretations of the informal specification of the `eval` operation. Additional discussions of problems with the semantics of the `eval` operation may also be found in [19] and in [16]. Such a situation demands a more precise definition.

In [15], a formal semantics of tuple spaces in ProSet by means of the formal specification language Z [25] has been presented to avoid such problems. We refined this formal specification into an implementation design and implemented a prototype from the formal specification. The prototype allows immediate validation of the specification by execution, and provides us with a *touch-and-feel* experience necessary to test the specification. The prototype enables us to avoid the large time lag between specification of a system and its validation in the traditional model of software production using the life cycle approach.

We can only sketch some implementation issues here. An implementation of a graphical debugger [23], and an implementation on a network of workstations [21] are in progress. Multiple tuple spaces provide a direct approach for distributing the tuple space on a distributed memory architecture. The representation of individual tuple spaces can be customized according to their contents and usage.

6 Conclusions

In this paper, we presented ProSet-Linda which adapts the concept for process creation via Multilisp's futures to set-oriented programming and integrates

Linda's concept for synchronization and communication via tuple space. The basic Linda model is enhanced with multiple tuple spaces, the notion of limited tuple spaces, selection and customization for matching, specified fairness of choice, and the facility for changing tuples in tuple space. It is fairly natural to combine set-oriented programming with generative communication on the basis of tuples, as both models, ProSet and Linda, provide tuples.

The small example presented here did not fully demonstrate the advantages of multiple tuple spaces. However, in more sophisticated problem domains such as process trellises [9] 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 [17].

We implement ProSet-Linda in a somewhat unconventional way: the informal specification is followed by a formal specification, which serves as the basis for a prototype implementation before the production-level implementation is undertaken. Applying formal methods early in the design stage of computer systems and software can increase the designer's productivity by clarifying issues and eliminating errors in the design. A side effect is that very few design errors will prevail at the implementation stage; design errors detected at the implementation stage are often more expensive to correct. When formal methods are systematically applied to all stages of design and implementation, we can increase our confidence that the software is robust and correct. A formal development process is more expensive in terms of time and education, but much cheaper in terms of maintenance. There may be bugs, but they are less likely to be at the conceptual level.

Our goal is to make parallel program design easier through prototyping of parallel algorithms. The high level of ProSet's constructs for parallel programming enables us to rapidly develop prototypes of parallel programs and to experiment with parallel algorithms.

References

1. H.E. Bal. A comparative study of five parallel programming languages. *Future Generations Computer Systems*, 8:121–135, 1992.
2. R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors. *Approaches to Prototyping*. Springer-Verlag, 1984.
3. N. Carriero. *Implementation of tuple space machines*. PhD thesis, Yale University, New Haven, CT, December 1987.
4. N. Carriero and D. Gelernter. *How to write parallel programs*. MIT Press, 1990.
5. J. Cocke. The search for performance in scientific processors. *Communications of the ACM*, 31(3):249–253, 1988.
6. E.-E. Doberkat. Integrating persistence into a set-oriented prototyping language. *Structured Programming*, 13(3):137–153, 1992.
7. E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL*. Leitfäden und Monographien der Informatik. Teubner-Verlag, 1989.

8. E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, editor, *Proc. Third International Workshop on Rapid System Prototyping*, pages 235–248, Research Triangle Park, NC, June 1992. IEEE Computer Society Press.
9. 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.
10. C. Floyd. A systematic look at prototyping. In Budde et al. [2], pages 1–18.
11. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
12. D. Gelernter. Multiple tuple spaces in Linda. In *Proc. Parallel Architectures and Languages Europe (PARLE'89)*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer-Verlag, June 1989.
13. J.B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.
14. R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.
15. W. Hasselbring. A Formal Z Specification of ProSet-Linda. Informatik-Bericht 04-92, University of Essen, September 1992.
16. W. Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language*. PhD thesis, University of Essen, 1993. (in preparation).
17. 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.
18. M.Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989.
19. J.S. Leichter. *Shared tuple memories, buses and LAN's — Linda implementations across the spectrum of connectivity*. PhD thesis, Yale University, New Haven, CT, July 1989.
20. J.E. Narem. An informal operational semantics of C-Linda V2.3.5. Technical Report 839, Yale University, New Haven, CT, December 1989.
21. R. Naujokat. Entwurf und Implementierung einer Laufzeitbibliothek für ProSet-Linda auf einem lokalen Netzwerk. Master's thesis, University of Essen, 1994. (in preparation).
22. H.A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.
23. H. Pohland. Entwurf und Implementierung eines graphischen Debuggers für ProSet-Linda. Master's thesis, University of Essen, 1994. (in preparation).
24. J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL*. Springer-Verlag, 1986.
25. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.
26. G. Wilson, editor. *Proc. Workshop on Linda-Like Systems and Their Implementation*. Edinburgh Parallel Computing Centre TR91-13, June 1991.

This article was processed using the LaTeX macro package with LLNCS style