Combining SETL/E with Linda[*]

W. Hasselbring

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

June 1991

Abstract

SETL/E 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 this paper a proposal will be presented for enhancing the basic Linda model with multiple tuple spaces, additional nondeterminism and the facility for changing tuples in tuple space to combine set-oriented programming with generative communication: SETL/E-Linda.

1 Introduction and overview

The set theoretic language SETL/E 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 instead of forcing it into a sequence. Most real systems are of parallel nature.

Parallel programming is conceptually harder than sequential programming, because a programmer has to think in simultaneities, but one can hardly focus on more than one process at a time. Programming in Linda provides a spatially and temporally unordered bag of processes, instead of a process graph thus making parallel programming conceptually the same order of task as conventional, sequential programming.

The Linda primitives were at first inserted in C, see e.g. [Gel85, ACG86]. Meanwhile there exist also versions for higher level languages such as Modula [BHK89], C++ [MPG90], Smalltalk [MK88], Eiffel [Jel90], Lisp [Hut90], Scheme [DM90] or Prolog [Mac89] and for operating systems [Lel90]. In section 3 a proposal for combining SETL/E with Linda will be presented. We will assume a basic knowledge of tuple space communication.

Section 4 sketches implementation issues and section 5 future research directions for our work.

2 The prototyping language SETL/E

The following subsections will present a brief introduction to data and control structures of the language and a short example. The high level structures that SETL/E provides qualify the language for prototyping. For a full account on prototyping with set theoretic languages we refer to [SDDS86] or [DF89]. Persistence is supported by the language and discussed in [Dob90].

[*]In the Proceedings of the Workshop on Linda-Like Systems and Their Implementation, editor G. Wilson, Edinburgh Parallel Computing Centre TR91-13, Edinburgh, UK, June 24, 1991.

2.1 Data structures

SETL/E provides the data types `atom`, `integer`, `real`, `string`, `boolean`, `tuple`, `set`, and `function`. Variables or constants of these types have first-class rights. Each variable or constant is meant to be an *object* for our terminology. 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, being comparable for equality with other objects, being returnable from or passable to a procedure. Both SETL and SETL/E are weakly typed.

Integer, real, string and boolean objects are used as usual. Atoms are unique for one machine and across machines. Additionally to the first-class rights, you can only create them. Function objects are either constructed via anonymous `lambda` expressions or by using a procedure name without a parameter list. You can call them.

Atom, integer, real, string, boolean and function objects are basic, because they are not built from other objects. Tuples and sets are heterogeneous and may be 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}]
```

Sets consisting only of tuples of length 2 are called maps. There's no genuine data type for maps, because in set theory maps are handled 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 so-called *multi-map selection* for relations.

There's 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 Ada as one of its ancestors. There are `if`, `case`, `loop`, `while` and `until` statements as usual and in addition some structures that are tailor-made to the compound data structures. At first have a look on tuple and set forming expressions:

```
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 quantifiers $(\exists x)$ and $(\forall x)$ of predicate calculus are provided, e.g.:

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

Again the visibility of `x` is bound to the loop resp. the quantifier. Additionally SETL/E provides the `whilefound` loop:

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

The loop body is executed, if an `exists` expression with the same iterator would yield `true`. The bound variables are local to the `whilefound` loop as they are in `for` loops. The iterator is reevaluated for every iteration unlike in `for` loops.

2.3 An example

The program in Figure 1 for topologically sorting the nodes of a directed graph provides an 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.

```
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 1: Topologically sorting the nodes of a directed graph.

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 is performed with the operator "+" and intersection with "*". Provided with this information the example should be easy to understand. One possible result would be the following tuple in `SortTup`:

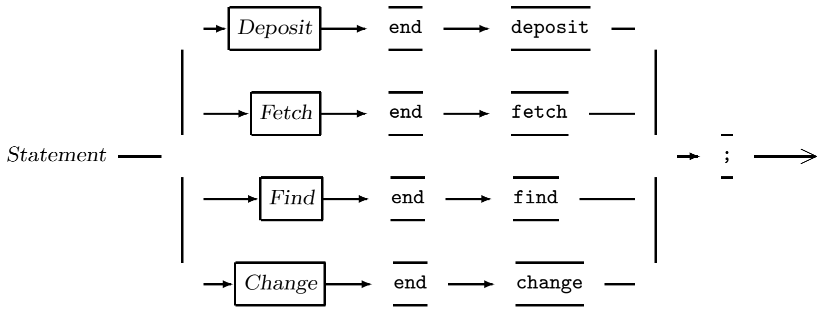
```
[1, "a", 2, "b", 3, "c", 4, "d", 5]
```

3 Generative communication in SETL/E

The following subsections will present a proposal for syntax and meaning of tuple space operations in SETL/E along with some small examples. We will assume a basic knowledge of tuple space communication, which is discussed in [CG90].

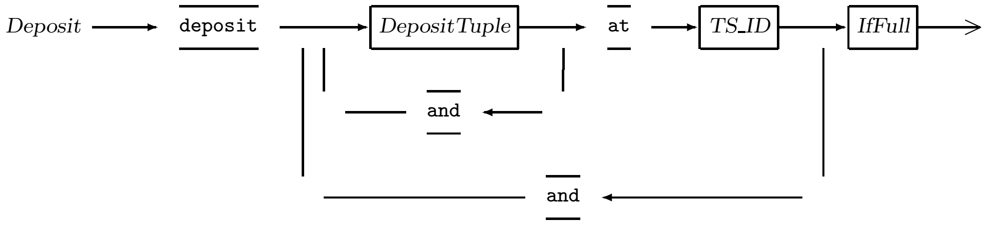
3.1 Tuple space operations

SETL/E provides four tuple space operations:

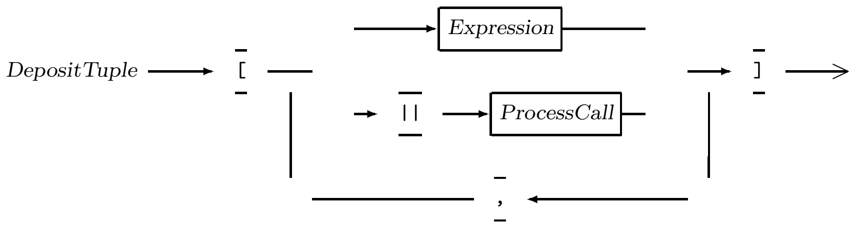


3.1.1 Deposit

The `deposit` operation merges the `out` and `eval` operations of C-Linda. It deposits new passive and active tuples into a specified tuple space. A tuple space is as usual a multiset of tuples. The syntax:



It is possible to deposit several tuples 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 any (optimal) way. The identifier *TS_ID* will be discussed in section 3.2. One single tuple to be deposited consists of a list of ordinary expressions and so-called *process calls*, resp.:



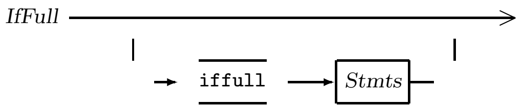
All expressions are evaluated in arbitrary order, before any tuple is put into the tuple space. All types with first-class rights are permitted to occur in tuple fields. If there are only ordinary expressions in a tuple, then this tuple is added as a passive one (cp. `out` of C-Linda).

The "`||`"-preceded function calls are called processes, because they are running concurrently with the process that executed the `deposit`. If there are any such process calls in a tuple, then this tuple is added as an active one to the tuple space. You might compare it with `eval` of C-Linda, but it is not exactly the same, however, because all fields of an `eval` tuple are executed concurrently and not only fields that were marked by the programmer.

When all processes in an active tuple have terminated their execution, the tuple space manager adds a passive tuple with the return values of these processes in the corresponding tuple fields to the tuple space. Active tuples are invisible to other operations until they convert to passive ones. The other three operations apply only to passive tuples (see below).

Side effects are not allowed for processes. Communication and synchronization is done only via tuple space operations.

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 to the programmer. E.g. in [Hut90] `out` is blocked until space is available again. In SETL/E-Linda this is done by default[1], too. Additionally the programmer can specify by option some statements to be executed, when there is no more memory available for a `deposit` operation:



You could e.g. raise an exception, when your tuple space is full:

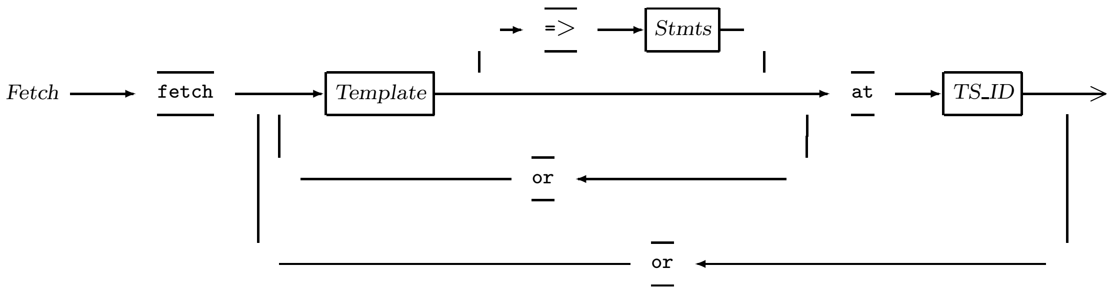
```
deposit [ x ] and [ "name", || f(x) ] at TS
  iffull raise TSisFull();
end deposit;
```

The suitable adjustment depends on the application you have to program. Thus a general setting seems not 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 before producing more tasks, when your tuple space is full. If you would instead block both, master and worker, a deadlock might occur, which should not occur according to the meaning of pure tuple space communication. The quality of this problem is comparable with "`malloc(size) == NULL`" in C. The meaning of the *IfFull* statements is not really satisfying: while they are executed, there might already be some space available. But we prefer to make the programmer aware of the problem of finite memory, instead of hiding it in the system.

Why merging `out` and `eval` to `deposit`? In `eval` *each* field of a tuple is evaluated concurrently. This is the semantics, but we do not believe that there exists any system that creates a new process to compute e.g. a plain integer constant. Thus the system has to decide, which fields to compute concurrently and which in a sequence. The same problems arise in automatic parallelization of functional languages: you have to reduce the existing parallelism to a reasonable granularity. In our approach the programmer has to report his knowledge about the granularity of his application to the system.

3.1.2 Fetch

The `fetch` operation merges `select` of Ada and `in` of C-Linda:



[1]We will change the default to raise a predefined exception.

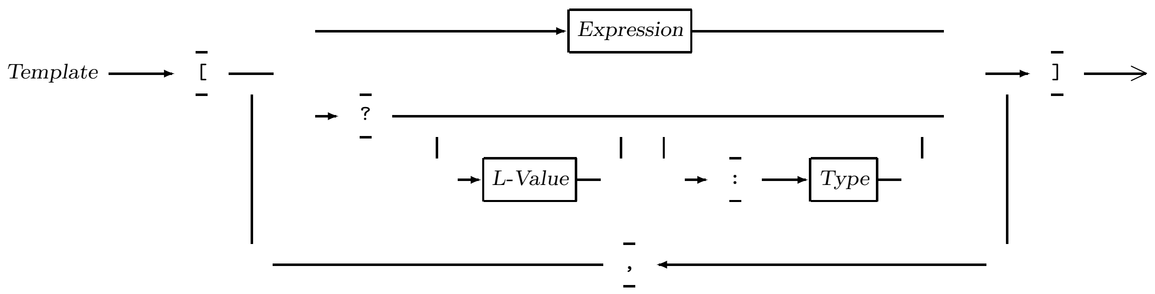
It is possible to specify several templates for multiple tuple spaces in one statement, but only one template may be selected nondeterministically[2]. The statement blocks until a match occurs. The selected tuple is removed from tuple space. If there are statements specified for the selected template, these statements are executed. There are two sources for nondeterminism:

1. There are several matching tuples for the templates: one tuple will be selected nondeterministically.
2. The selected tuple matches several templates: one template will be selected nondeterministically. The SETL/E-`case` statement does pattern matching in the same way.

There are several ways of handling fairness while selecting tuples or templates that match:

- Selection is done by the system without any consideration of fairness.
- Fairness could be achieved if the system always selects the oldest tuple resp. template that matches. But does that really fit with associative matching?
- One could at least specify that each tuple or template will indeed be selected, if it is infinitely often possible to select it (see also [CM88]). The implementation has to guarantee this in some way.

We prefer the last alternative. Arguments on this subject would be appreciated. However, if a specific order is necessary, it has to be enforced via *structured naming* [Gel85]. One single template consists of a list[3] of ordinary expressions and so-called *formals*:



The expressions are called *actuals*. They are at first evaluated in arbitrary order. The fields that are preceded by a question mark are the *formals* of the template. Tuples and templates match, if

- the number of fields is equal,
- types and values of actuals in templates are equal to the corresponding tuple fields and
- the types in the formals are equal to the types of the corresponding tuple fields. You can specify any of the types mentioned in section 2.1. If there is no type specified in a formal, this field matches unconditionally.

The *L-Values* specified in the formals are assigned the values of the corresponding tuple fields. 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. However, a specified type is considered. An example:

[2]We ponder adding the facility for specifying multiple templates in conjunction to avoid unnecessary sequences for fetching tuples.

[3]We ponder replacing the enclosing brackets by other symbols to set the templates apart from SETL/E-tuples. Such a replacement may be beneficial for `deposit`-tuples, too.

```
fetch [ "name", ? x : integer ] => put("Integer fetched");
   or [ "name", ? x : set ]     => put("Set fetched");
   at TS
end fetch;
```

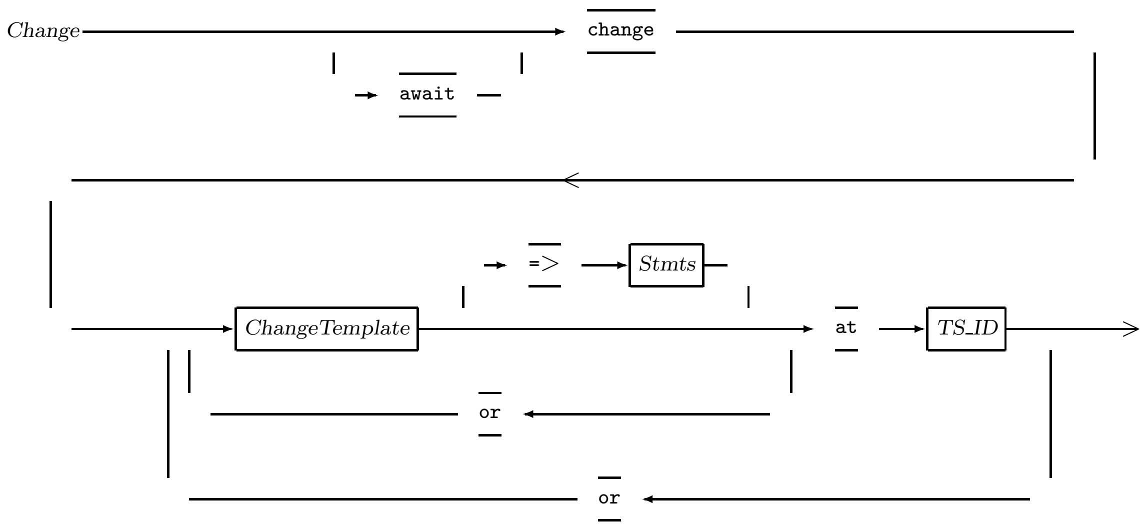
Why did we choose not to specify `else` statements to be executed, if none of the templates matches? Especially in a distributed environment there may be matching tuples available while these statements are executed, because of concurrent access to a tuple space. Thus the meaning of these statements 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. Because of this fact there would be severe problems if tuple spaces would be given first-class rights (section 3.2). The statement "The tuple space is full!" (section 3.1.1) has another quality, however, because it is not known to the coordination language Linda, since a `deposit` resp. `out` operation is always non-blocking. Reaching a full tuple space is due to finiteness of real machines.

3.1.3 Find

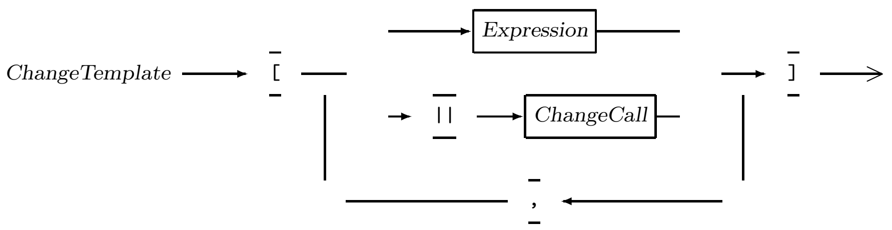
The `find` operation merges `select` of Ada and `rd` of C-Linda. Except for the fact that is leaves the tuple it found in tuple space it works like the `fetch` operation. You have simply to exchange the keyword `fetch` with `find` in the first syntax diagram of section 3.1.2 to obtain the syntax for the `find` operation.

3.1.4 Change

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 and introduce the `change` operation for overcoming it:



This looks like the `fetch` operation, but the *change template* is different:



The ordinary expressions are evaluated as usual. The *change calls* preceded by "`||`" are function calls where exactly one actual parameter has to be a formal. You should not get confused by actual and formal parameters of functions and actuals and formals of templates! We omit the syntax diagram here to save space. These formals are used to create templates, which are used for matching as usual. The statement blocks until a match occurs. The selected tuple is removed from tuple space. The values of the tuple fields that were fetched for the corresponding formals of the template are assigned to the corresponding formal parameters of the change calls. Thus it is not allowed to specify *L-Values* for these formals, but optionally types. Additional actual parameters (expressions) in the change calls are handled as usual for function calls. They do not affect matching. The change calls are concurrently executing processes in active tuples, such as those produced by `deposit` (see section 3.1.1).

In summary: a passive tuple is converted into an active one, which will eventually convert into a changed passive tuple (if each change call terminates). Only the tuple fields that correspond to change calls are exchanged with the return values. Indivisibility is guaranteed, because active tuples are invisible to other processes. Moreover, fetching the passive tuple at starting and depositing the changed passive one at the end of the user-defined operation on shared data are atomic operations.

You can `await` termination of these processes by an option. This is not possible for `deposit`. If there are statements specified for the selected change template, these statements are executed. Matching is done in the same way as described in section 3.1.2. The following example:

```
change [ "x", || add (? integer, 2), || eat (?) ] at TS end change;
```

is equivalent to

```
fetch [ "x", ? a : integer, ? b ] at TS end fetch;
deposit [ "x", || add (a, 2), || eat (b) ] at TS end deposit;
```

But with the `change` operation expensive copying of compound data may be avoided.

3.2 Multiple tuple spaces

The idea of splitting the tuple space into multiple ones is not new. [MK88] introduces a new class `TS` and [Gel89, Lel90] propose new data types (`ts` resp. `dict`) to organize them. But in the basic tuple space operations these authors use values of objects of these types as identifiers/references for tuple spaces and not as the value of a tuple space itself. However, in principle it is rarely possible to make any statements on the *actual value* of a tuple space (see also section 3.1.2). A tuple space may be viewed as the dynamic envelope of a growing/shrinking multiset of passive and active tuples that controls the communication and synchronization of parallel processes.

Our approach is to use atoms to identify tuple spaces. As mentioned in section 2 atoms are unique for one machine and across machines. They have first-class rights. We pondered introducing a new data type for tuple space communication, but did not see a justification.

Every SETL/E 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 blocked, the tuple space manager should terminate with an appropriate warning message. Tuple space communication in SETL/E is designed for *multiprocessing* (single application running on multiple processors) as opposed to *multiprogramming* (separate applications) which is done via a separate mechanism for handling persistent data, based on critical regions for persistent objects.

SETL/E provides several library functions to handle multiple tuple spaces:

`createTS()`: Calls the standard function `newat` to return a fresh atom. The tuple space manager is informed to create a new tuple space for this atom. The atom will be returned by `createTS`. Thus you can only use atoms that were created by `createTS` to identify tuple spaces.
We ponder introducing a parameter specifying the expected or wanted size for the tuple space.

`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)`: Removes `TS` from the list of existing tuple spaces and calls `clearTS(TS)`.

`CreateTS` provides information hiding to tuple space communication. Tuple space identities have first-class rights. Processes that live in active tuples may also operate in the tuple space they live in, if the corresponding tuple space identity was passed to them.

3.3 Examples

We will now present the solutions for two frequently discussed examples in parallel computing: the dining philosophers problem and parallel matrix multiplication.

The dining philosophers problem was posed originally by Dijkstra, and is often used to test the expressivity of new parallel languages. The problem description together with a C-Linda solution and reasoning about deadlock freedom is given in [CG90, pages 182–185]. We use this solution as starting point for the program in Figure 2. Fairness in the C-Linda solution relies on a fair runtime system. Our solution assures fairness by *assigning* room tickets to philosophers: each philosoph can only fetch tickets with *his* number and has to return a ticket for the next one (modulo `n`). Before he gets this ticket again, each of the others must have taken and returned it. Thus no philosoph will be omitted forever from obtaining a ticket. The solution works for arbitrary `n > 1`.

```
program DP;
   visible constant n := 5,                    -- #philosophers
                    ROOMTICKETS := createTS ();
                    CHOPSTICKS := createTS ();
begin
   for i in [ 0 .. n-1 ] do
      deposit [ i ] and [ || phil(i) ] at CHOPSTICKS end deposit;
      if i /= n-1 then
         -- one ticket less than #philosophers
         deposit [ i ] at ROOMTICKETS end deposit;
      end if;
   end for;

   procedure phil (i);
   begin
      loop
         think ();
         fetch [ i ] at ROOMTICKETS end fetch;            -- own ticket
         fetch [ i ] at CHOPSTICKS end fetch;             -- left chopstick
         fetch [ (i+1) mod n ] at CHOPSTICKS end fetch;   -- right chopstick
         eat ();
         deposit [ i ] and [ (i+1) mod n ] at CHOPSTICKS
             and [ (i+1) mod n ] at ROOMTICKETS           -- for the next phil
         end deposit;
      end loop;
   end phil;
end DP;
```

Figure 2: Solution for the dining philosophers problem.

The program in Figure 3 on page 11 presents a parallel matrix multiplication. Because SETL/E provides no arrays, we use tuples of tuples to represent two-dimensional matrices. The unary operator "`#`" returns the number of elements in a compound data structure. 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 `change` operations. Tuple space `WORK` is cleared after work has been done.

This program does not support load balancing, because the number of workers is fixed to 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].

The small examples presented here did not demonstrate the advantages of multiple tuple spaces. However, in more sophisticated problem domains such as process lattices [FG88] the advantage of information hiding is obvious. The enhanced facilities for nondeterminism will support distributed implementation of backtracking, such as branch-and-bound applications [KBT89].

4 Implementation

The compiler construction system Eli is the central tool for implementing SETL/E. Eli integrates off-the-shelf tools and libraries with specialized language processors to provide a system for generating complete compilers quickly and reliably [GHK$^+$90]. A first development step using Eli is documented in [Has91].

The tuple space communication is not yet implemented. To obtain an executable system for testing and code development an uniprocessor simulation will be implemented as a first step. A real parallel implementation will be done on an Ethernet-based LAN of workstations. 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. Also a Transputer-based implementation is planned.

5 Future work

Future work will concern the refinement of syntax and semantics of the presented tuple space operations and the implementation aspects that were discussed in section 4. The author is also interested in specification of formal semantics for generative communication.

Acknowledgements

Thanks to E.-E. Doberkat, M. Goedicke and R. Rüdiger for their comments and discussions on earlier versions of this paper.

References

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

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

[Bri90] British Computer Society. *High Performance and Parallel Computing in Lisp*, EUROPAL Workshop, Twickenham, London, UK, November 1990.

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

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison Wesley, 1988.

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

```
program matrix;
   visible constant WORK := createTS (),
                    RESULT := createTS ();
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
      find [ "A", i, ? row : tuple ] at WORK end find;
      find [ "B", j, ? column : tuple ] at WORK end find;
      Dot := DotProduct (row, column);
      change [ "C", || AddDot(?, i, j, Dot), || Incr(? integer) ]
          at RESULT
      end change;

      procedure DotProduct (row, column);
         ...
      end DotProduct;
      procedure AddDot(C, i, j, Dot);
      begin
         C(i)(j) := Dot;
         return C;
      end AddDot;
      procedure Incr (count);
      begin
         return count+1;
      end Incr;
   end worker;
end matrix;
```

Figure 3: Parallel matrix multiplication.

[DGH90a] E.E. Doberkat, U. Gutenbeil, and W. Hasselbring. SETL/E – A prototyping system based on sets. In W. Zorn, editor, *Tagungsband 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.

[DM90] U. Dahlen and N. MacDonald. Sheme-Linda. In [Bri90].

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

[FG88] M. Factor and D. Gelernter. The parallel process lattice as an organizing scheme for realtime knowledge daemons. Technical memo, Yale University, March 1988.

[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 of *Springer-Verlag LNCS 366*, pages 20–27, June 1989.

[GHK$^+$90] R.W. Gray, V.P. Heuring, S.P. Krane, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. Software Engineering Group Report 89-1-1, University of Colorado, Boulder, June 1990.

[Has91] W. Hasselbring. Translating a subset of SETL/E into SETL2. Informatik-Bericht 02-91, University of Essen, January 1991.

[Hut90] D. Hutchinson. Linda meets Lisp. In [Bri90].

[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.

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

[Mac89] N.B. MacDonald. A Distributed Linda Kernel. Technical Report ECSP-TN-33, Edinburgh Concurrent Supercomputer Project, 1989.

[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.

[MPG90] M.J. Manthey, H.S. Pedersen, and G. Gunnlaugsson. ARIADNE — A Linda Kernel for Transputer Networks. Technical Report R 90-17, University of Aalborg, Denmark, April 1990.

[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.