A detailed discussion of performance and implementation issues is the subject of another paper [2].

References

[1] *Carriero N., Gelernter D.* **How to write parallel programs: A guide to the perplexed** ACM Computer Surveys, Sept 1989.

[2] *Czajkowski G., Zieliński K.* **Extention of Strand with Linda-like operations — Implementation and performance study** Internal Report, Institute of Computer Science, UMM

[3] *Foster I., Taylor S.,* **Strand - New Concepts in Parallel Programming** Prentice-Hall, Inc, 1990.

[4] *Gelernter, D. Parallelism to the People, or, Experience with Programminig in Linda.*

[5] *POSYBL-1.102 - system documentation.*

[6] *Sterling L., Shapiro E.* **The Art of Prolog** The MIT Press Cambridge, 1986

[7] *Strand Software Technologies Limited* **STRAND'88 USER MANUAL, BUCKINGHAM RELEASE, JUNE 1990**

[8] *Tick E.* **Parallel Logic Programming** MIT Press, Cambridge MA, 1991

[9] *Tick E., Banerjee C.* **Performance Evaluation of the Monaco Compiler and Runtime Kernel** International Conference on Logic Programming, Budapest, June 1993

each of them containing the same data but in (usually) different order. If X is the youngest and best earning man in the company, his position in data structures sorted according to age and salary will be different (ascending order of sorting). In the extended model, every entry may be equipped with four numbers, each of them giving the entry's position according to appropriate key. With such approach, one would have to maintain only one data structure (that is, STS, and that is done automatically in our implementation). One could also encapsulate complex queries in one call. For example *st_inp({ Name, _, _, _, _, 1, _, 1 }, Found)* might be used to check if the youngest gets the highest salary. In STRAND, one would have to check two data structures. There are possible numerous variants to such techniques. The important thing is that STS objects are addressed only by their contents and not by their position in any data structure. Adding a field giving certain information about STS object to this object places the object in the *logical* hierarchy with respect to some new order, without the need to create *physically* a data structure sorted according to that order.

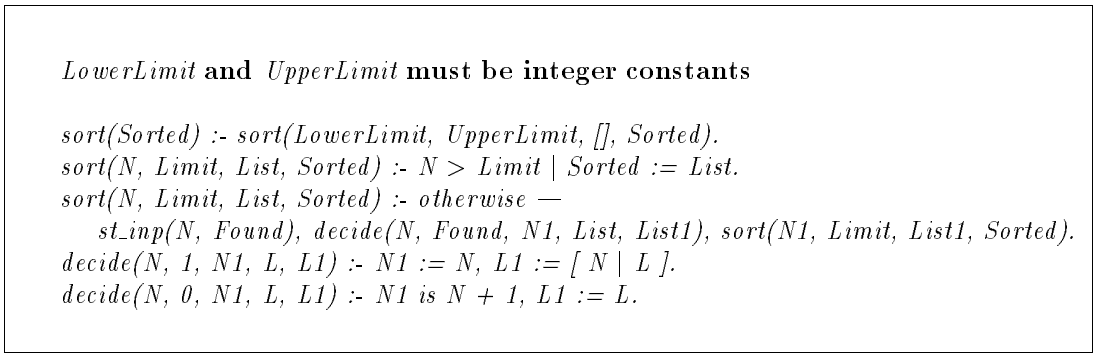

Figure 11: **Sorting objects in STS - problem's complexity is proportional to the number of elements in STS**

.

The extended model may also offer more effective approach to sorting. Fig. 11 presents the program that returns the list of integers sorted in descending order. These numbers are initially in STS and each of them is not smaller than some lower bound and not greater than some upper bound. If the difference between bounds is m and the number of integers in STS is n, than the number of calls to *st_inp* grows linearly with n and m, and is equal exactly to $m + n$, giving linear complexity.

Of course, the algorithm is not perfect (cannot sort real numbers, for example) but gives the general idea of how STS and its interface operations may be used.

4 Conclusions

STRAND execution model may be extended with a set of Linda-like operations and a concept of tuple space. These STRAND tuple space is orthogonal to STRAND process pool. Processes may access STS objects without setting up a shared data structure connection first. Connection is established only for the time of the call duration. There is a wide class of problems that could be programmed more easily and readably in the presented model. The presented extension of STRAND has been implemented and the performance of presented techniques has been evaluated. This study proved that the proposed approach may be implemented and effectively used.

a) program written in STRAND

```
flatten(List, Flattened, Done) :- f(List, Flattened / [], done, Done).
f([], A / B, Left, Right) :- Right := Left, A := B.
f([ H | T ], A / B, Left, Right) :- f(H, A / M, Left, Middle), f(T, M / B, Middle, Right).
f(IntVal, A / B, Left, Right) :- integer(IntVal) | A := [ IntVal | B ], Right := Left.
```

b) program written in the extended model

```
flatten(List, Flattened) :- st_out(working), f(List, Flattened / []).
f([], A / B) :- A := B, st_in(working).
f([ H | T ], A / B) :- st_out(working), f(H, A / M), f(T, M / B).
f(IntVal, A / B) :- integer(IntVal) | A := [ IntVal | B ], st_in(working).
```

Figure 10: *short-circuit* **technique used for the process termination detection and solution offered by our model**

- we do not have to specify communication links between the blackboard managing process and blackboard using processes
- if our implementation is efficient (STS search optimization) programs may be much more efficient than their LP counterparts
- in the LP approach, a blackboard is usually a list or tuple of elements. Adding or removing of one element causes (because of the single assignment rule) the whole new data structure to be rebuilt. In our model a blackboard is represented not as a one big data structure but as a set of small objects. Thus, changing one of them do not affect the other ones at all.

We have already given an example of a blackboard, in Fig 7.

Although it is the most naive approach to the problem (for example, no scanning of data structure is performed), we can see that better *manager* definitions will be far more complex. Our model does not introduce any complexity. STS acts as a blackboard and underlying TS operations provide efficient database searching. We will not give an example because the extended model operations described earlier fit most naturally to distributed data management. Every operation can be expressed in terms of reading, entering or taking data out of STS.

3.7 New techniques

As has been shown, many traditional parallel logic programming techniques providing interprocess communication and data management may be effectively replaced by methods offered by the extended model. However, new computational models usually enable new approaches. In this subsection we will show how the idea of virtual associative memory may influence some aspects of sorting.

Imagine a database of personal information entries. Each entry would include employee's name, age, number of children and salary. We would like to have the database sorted according to all these four keys. In STRAND we would have to maintain four different data structures,

STRAND solution - difference lists are used

```
go(InitsList, Perms) :- perm(InitList, [], Perms / []).
perm([], NewPerm, A / B) :- A := [ NewPerm | B ].
perm(List, SoFar, A / B) :- List =[] | perm([], List, SoFar, A / B).
perm(_, [], _, A / B) :- A := B.
perm(Aside, [ H | T ], SoFar, A / B) :- append(Aside, T, NewInitList),
        perm(NewInitList, [ H | SoFar ], A / M), perm([ H | Aside ], T, SoFar, M / B).
append([], X, Y) :- X := Y.
append([ H | T ], X, Y) :- Y := [ H | Z ], append(T, X, Z).
```

extended model approach - solutions are entered into STS

```
go(InitsList) :- perm(InitList, []).
perm([], NewPerm) :- st_out(NewPerm).
perm(List, SoFar) :- List =[] | perm([], List, SoFar).
perm(_, [], _).
perm(Aside, [ H | T ], SoFar) :- append(Aside, T, NewInitList),
        perm(NewInitList, [ H | SoFar ]), perm([ H | Aside ], T, SoFar).
```

Figure 9: **Collecting all solutions**

variable of the initial process. It is illustrated by the next example — nested list flattening (Fig 10a).

Again the extended model offers a simpler solution (Fig 10b). Every *Right := Left* assignment is replaced by removing *working* symbol from STS and introducing the *Middle* variable is replaced by putting this symbol into STS. Thus, the work is complete if there are no *working* symbols in STS. There is less code in this solution and one additional benefit — any process that knows that flattening has happened may check out for itself if it has been completed, without sharing *Done* variable. All it has to do is to ask STS if there is still any *working* symbol in it: *st_rdp(working, Status).*

An observant reader will notice that the two last programs are not fully correct. In the first program the process pool manager may choose only *Right := Left* assignments and *f/4* rocesses to be reduced. In consequence, *Done* will be instantiated to *done* but *Flattened* will still be a variable. Such a situation is highly unlikely (and such programming style is extremely common in logic programming). It can also happen in the second program. Both programs require exactly the same amount of work to rule out such a possibility completely.

3.6 Blackboards

A data structure to which a number of processes require an access is called a blackboard. In LP languages a blackboard is usually managed by a process reading a stream of merged access requests. Our extended model offers a completely different approach. There is no managing process. The data structure is distributed over the network. Our five operations are enough to do everything we want with that data structure — check for the elements existence, read, remove and add them. We can immediately show several advantages of such an approach:

- we do not have to specify the actions of the managing process for there is no such a process

program written in STRAND

```
startup_process :- Buffer := [ _, _, _ | Tail ], producer(Buffer), consumer(Buffer, Tail).
producer([ Head | Tail ]) :- Head := item, producer(Tail).
consumer([ item | Tail ], LastTail) :- LastTail := [ _ | NewLastTail ], consumer(Tail, NewLastTail).
```

program written in the extended model

```
startup_process :- Msg := { produce, go }, st_out(Msg), st_out(Msg), st_out(Msg), producer, consumer.
producer :- st_in({ produce, Go }), producer(Go).
producer(go) :- st_out({ item, go }), producer.
consumer :- st_in({ item, Go }), consumer(Go).
consumer(go) :- st_out({ produce, go }), consumer.
```

Figure 8: *bounded buffers* **solved in different computational models**

A producer may produce only after having removed (possibly after blocking itself) a *produce* string from STS. A consumer enters *produce* into STS on the completion of consuming.

3.4 Difference lists

A difference list technique is well known in LP [6] . It allows lists to be concatenated in constant time and constructed by many processes in parallel. In parallel logic programming languages this latter feature is very desirable. A lack of the built-in backtracking mechanism makes gathering of all solutions of the given problem a programmer's task, in contrast to Prolog, where a system predicate *findall/3* may be used. A program in Fig 9a shows how difference lists may be used to find a list of *all* permutations of a given input list.

In the extended model the solution to this problem is much simpler (Fig 9b). No distributed difference list is needed because all solutions will be finally entered into STS, from which they can be fetched any time. A user does not have collect results, thus adding to the complexity of the solution. Instead, as soon as a solution is generated it is *out*ed into STS. A programmer does not have to think about how to collect solutions — he can devote all his attention and effort to solve the problem.

3.5 Short circuit and process termination detection

In parallel logic programming languages it is sometimes very important to detect a process termination. Subsequent actions of the other processes may depend on the results this process returns. One of the methods is to let the process do its work and make the processes expecting results be blocked in their guard sections. However, a better approach, called short-circuit, has been devised. Two variables, *Left* and *Right* are added to all processes cooperating on the common task. When a process has completed its subtask the *Right* := *Left* assignment is made. When the process forks, a *Middle* variable (possibly more then one of such variables) is introduced, becoming an indirect link between *Left* and *Right*. In such a way, a value assigned to *Left* is propagated through the chain of processes, becoming eventually assigned to the *Right*

program written in STRAND - definition of the distributor **process**

distr(ProducerStream, ConsumerStream,PendingGoods, PendingRequests)
distr([Item | PS], CS, PG, [Request | PR]) :- Request := Item, distr(PS, CS, PG, PR).
distr([Item | PS], CS, PG, []) :- distr(PS, CS, [Item | PG], []).
distr(PS, [request(Item) | CS], [Stored | PG], PR) :- Item := Stored, distr(PS, CS, PG, PR).
distr(PS, [request(Item) | CS], [], PR) :- distr(PS, CS, PG, [Item | PR]).

approach offered by our model - a whole **program**
all data items are managed by STS internal processes

producer :- st_out(item), producer.
consumer(go) :- st_in(Item), decide(Item).
decide(Item) :- consume(Item, Go), consumer(Go).
consume(item, Go) :- Go := Go.

Figure 7: **Accessing distributed data**

with such a process may be rather tedious, especially when many processes may require unique identifiers. It may also be wasteful. Imagine a process which does not need to get a unique number — but some of its body processes may need to. In such a case, a communication channel must be passed down to the body process. A situation may become even more complicated when two (or more) body processes will have to have an access to *uniq_id*. Their message streams would have to be merged into a single stream. In complex distributed applications providing an access to *uniq_id* to many processes may become a nuissance. On the other hand, our model gives very simple solution (Fig 6).

Multiple producers and multiple consumers are the last variant of the producer—consumer protocol with incomplete messages to be discussed. The simplest STRAND solution would be to equip each producer process with one output channel. All producers' channels would be merged into one stream, delivered to some *distributor* process. All consumers would communicate with *distributor*, and their messages would also have to be merged in one stream. For simplicity we show only a possible *distributor* definition (Fig 7a). It can be treated as a finite state machine. A related problem will be discussed in the section covering blackboards.

Even though in this example conditions are somewhat relaxed (i.e. the producer does not care if anybody consumes its items) a complete program is complicated and it is not obvious at the first glance how it works. Our model offers much simpler solution (Fig 7b). Producers just produce and consumers — consume. There are no problems with merging or distributing.

Let us now consider a similar problem but with an additional constraint: producers cannot go more than n items ahead of consumers. Programming such a global inter—producer buffering in the traditional LP would require implementing some additional producer—distributor communication protocol, further complicating the solution. There exists a LP technique, called bounded buffers, to deal with this problem. A producer is given a list of n variables. The tail of the list is unbound, too. A consumer is given an access to this tail and every time it consumes something it extends the list by one (Fig 8a).

In our model entering a fixed number of *produce* strings into STS is allt is needed (Fig 8b).

startup_process :- st_out({ uniq_id, 0 }), **... other processes.**

process p/0 **wants to get a unique identifier**
p :- uniq_id(Num), **... further** p**'s actions**

uniq_id/1 **does not have a second argument indicating a process completion, because it can be checked by** integer/1 **STRAND kernel test**

uniq_id(Num) :- st_in({ uniq_id, Num }), Num1 is Num + 1, out_new_num(Num).
out_new_num(Num) :- integer(Num) | st_out({ uniq_id, NewNum }).

Figure 6: **Providing network-wide unique identifiers**

a worker asks for the next task. Assigning problems dynamically, at run time, only when a worker is idle, allows for a better processor utilization — static, compile-time task allocation could lead to situations where some workers finished all their tasks long before the others. A well known LP solution is to maintain a farmer process, with which workers can communicate. On completion of its task a worker asks for another (possibly sending its results at the same time) and farmer's response is either a new task or a special *done* message, indicating end of work. In our model a farmer is not concerned with replying to workers' queries for there are no queries. All tasks are put into STS at the beginning of the program and all the farmer is supposed to do is to collect and process results.

Because of the nondeterministic STS searching scheme, workers grab tasks in an unordered way, in contrast to the traditional LP solution, where the list of tasks is maintained and tasks are assigned in the same order they are stored in the list (situation resembling the sliding window protocol). Of course, our model may order tasks, but in most farmer-worker applications nondeterminism is preferred to message ordering.

3.3 Incomplete messages and bounded buffers

Because of the nondetermistic choice of reduced processes, the following rather undesirable scenario might happen in both previous programs: after having reduced *producer/1* process, *consumer/1* is reduced and alternating two—process reduction scheme is repeated, leaving all the *do_work/0* processes (their number is still growing) in the process pool. A synchronisation technique, called *incomplete messages* has been devised in order to overcome this difficulty. In this variant, *producer* must wait until the produced item has been consumed. Our model gives equally straightforward solution, as shown in Fig 4.

In both examples, data flows in both directions. However, it is a rather clumsy task to generalize the program 4a to handle multiple consumers (without using *merger/2*, predefined in newer STRAND versions). For n consumers, n channels would have to be used and n clauses added to the program. Program 5 shows how to solve this problem in the extended model.

Consumers are parametrized but there is only one *producer/1* clause. There are no permanent producer—consumer connections. There may be as many consumers as necessary. The only requirement is that every newly created consumer gets a unique number. Programming such *uniq_id* process is very simple in STRAND, but establishing proper communication channels

a) standard logic programming approach

```
startup_process :- producer([], Channel), consumer(Channel).
producer([], Channel) :- Channel := [ do_work(OK) | Channel1 ], producer(OK, Channel1).
consumer([ do_work(OK) | Channel ]) :- do_work(OK), consumer(Channel).
do_work(OK) :- display_nl(done, OK)!.
```

b) extended model approach

```
startup_process :- producer(1), consumer.
producer(1) :- st_out(do_work), st_in({ done, X }), producer(X).
consumer :- st_in(do_work), do_work(OK), consumer(OK).
consumer([]) :- st_out({ done, 1 }), consumer.
do_work(OK) :- display_nl(done, OK)!.
```

display_nl/2 **assigns an empty list to its second argument after printing the message**

Figure 4: *incomplete-messages* **protocol in LP and its functional equivalent in the extended model**

```
startup_process :- producer, consumer(1), consumer(2).
producer :- st_out({ do_work, 1 }), st_out({ do_work, 2 }),
    st_in({ done, X }), producer(X).
producer(X) :- integer(X) | out({ do_work, X }), st_in({ done, Y }), producer(Y).
consumer(Num) :- st_in({ do_work, Num }), do_work(OK), consumer(OK, Num).
consumer([], Num) :- st_out({ done, 1 }), consumer(Num).
do_work(OK) :- display_nl(done, OK)!.
```

Figure 5: **The extended model solution to the single-producer and multiple-consumers problem**

orders received.

Our extended model allows one to use STS operations to achieve a similar goal, as shown in Fig 2b. If the number of consumers is constant, i.e. number of processes reading from *Channel* does not change, program 2b gives performance roughly the same as 2a but it is simpler to write and analyze. Employing of an extended model to deal with the changing consumers number problem is of course possible, but more complicated than using traditional producer-consumer protocol, based on sharing variables. However, this is the simplest and most primitive protocol and we will not discuss it any further. In following subsections we try to show usefulness of our model in more complex and more frequently used protocols.

Before we do that we would like to point out that our model is particularly well suited to perform distributed computations in a *farmer-worker* fashion, where a number (possibly changing) of worker processes waits for orders from the farmer (Fig 3). When the work is done,

A task is a tuple { TaskNumber, TaskData } , **(where** TaskData **is application dependent STRAND data structure) or** 'done' **string, indicating work completion**

```
manager(TaksList, Tasks WorkersNum) :-
    set_workers(WorkersNum), out_list(TaksList), collect_results(TasksNum, WorkersNum).
```

set_workers/1 **distributes** worker/1 **processes on available processors;**
collect_results **handles and counts results entered into STS by workers;**
upon completion WorkersNum 'done' **strings are put into STS**

```
out_list([]).
out_list([ Head | Tail ]) :- st_out(Head), out_list(Tail).
worker([]) :- st_in(Task), decide(Task).
decide(done).
decide({ TaskNumber, TaskData }) :- do_task(TaskData, Result, Done),
    out_result(Done, { TaskNumber, Result }), worker(Done).
out_result([], Result) :- st_out(Result).
```

do_task(TaskData, Result, Done) **- performs task with input data TaskData; on completion** Result **is assigned computed value and** Done **becomes empty list**

Figure 3: **Extended model solution to** *farmer-worker* **class of problems**

is identical with putting only one process there which contains the above mentioned k processes in its body. This single process is immediately reduced.

There are several basic programming techniques providing interprocess communication, namely: producers-consumers, incomplete messages, bounded buffers, difference lists, short circuits and blackboards. To cite the STRAND's authors, *to a large extent, programming in STRAND revolves around the repeated use of these six techniques in different guise* [3]. In the following subsections we show how we can use the extended model operations to obtain programs functionally equivalent to the ones written in pure STRAND. In this comparison we stress easiness of programming, complexity and efficiency issues.

3.2 Producer-Consumer protocol

Fig. 2a shows a process making its three body processes share a variable — a possible communication channel. Let us have a look how these processes could transfer data. We will rule out the possibility of more than one of them trying to assign a value to the shared variable. The reason is simple — there is no language construct allowing a process to do the following in one step: *test if a given data is variable — and if so, lock it not allowing other processes to write to it.* As a consequence, two processes may simultaneously check if the data is a variable and then try to assign a value to it. However only one will be successful — the other one will cause the run-time system to issue a rather unpleasant message informing of a single-assignment rule violation. Thus, a shared variable message interprocess communication protocol is inherently a producer-consumer model. One process issues commands and the other base their actions on

a) solution offered by the standard logic programming approach

```
startup_process :- producer(Channel),consumer(Channel), consumer(Channel).
producer(Channel) :- Channel := [ everybody_works | Channel1 ],producer(Channel1).
consumer([ everybody_works | Channel1 ]) :- do_work, consumer(Channel1).
do_work :- display_nl(done)!.
```

b) solution offered by the extended model

```
startup_process :- producer, consumer(1), consumer(2).
producer :- st_out({ everybody_works, 1 }),st_out({ everybody_works, 2 }), producer.
consumer(Num) :- do_work, st_in({ everybody_works, Num }), consumer(Num).
```

Figure 2: **Two approaches to the simple producer-consumer protocol**

STS managing processes ensure that STS object matching an argument to *st_in/1* or *st_inp/2* is atomically locked before being removed from STS. A consequence is that after *st_out(1, _)* only one of the two simultaneous calls to *st_inp(1, Done)* will succeed.

The next section compares pure and enhanced STRAND programming techniques.

3 Programming in extended model

This section shows examples of programming interprocess communication and data management in the STRAND extended model. These techniques are compared with well—known LP programming methods. We have decided to show short comparatory examples rather than a few big programs because the chosen techniques are building blocks from which all more complex protocols are composed.

3.1 Overview of communication protocols

The key idea in the parallel logic program execution is a process pool. A process pool can be thought of as a set of processes waiting for their turn to become reduced. A nondeterministically chosen process is taken out of a process pool and matched against its clauses. In practice, a choice of a process is not always nondeterministic — different ordering policies may be employed. Similarly, although trying out all clauses in parallel is feasible, some implementations base their decisions concerning which clauses to try first on textual order of the source program. If a process is reduced successfully, its body processes are entered into the process pool. An unsuccessful reduction attempt causes the process to be put back into the pool. A computation is finished if an initially non-empty process pool becomes empty as a result of numerous reductions. An important point to note is that there is no global state or global variables. A process can pass variables and input data to its offspring — that is, to its body processes. A quick conclusion is that there is only one way to set up a communication between two processes: they must have a common ancestor (not necessarily in one reduction step) that should equip them with common variables. A slight variation to this scheme is sharing a variable between two processes from the initial process pool. This is a slight variation indeed, for putting k processes in the process pool

Second, processes (so called *blackboards*, described later on) managing data that may be accessed by other processes are almost unnecessary. STS may be treated as a blackboard and its interface operations as queries to a blackboard. Besides simplifying a source code (there is *no* definition of a blackboard) a programmer does not have to set up any interprocess communication.

Third, an additional degree of nondeterminism is introduced because order of searching STS by added operations is not determined. Writing a process that has to pass two two integers to two other processes (each process gets one integer) is very simple in STRAND. However, if passing integers must be nondeterministic, such a program becomes more complex. The extended model provides equally simple techniques for both deterministic and nondeterministic behavior of programs.

2.4 Operations of the extended model

In the previous subsection a term STS (STRAND tuple space) has been introduced. Such a name was chosen for the sake of uniformity with Linda terminology. However, in LP languages tuples are important but not only data structures. For that reason data found in STS will be called *objects*.

In order to assess usefulness of different operations, we have decided to implement the following five:

- *st_out(Data, Status)* — enters *Data* into STS. *Data* may be any STRAND data — list, tuple (possibly nested) or basic type data. If *Data* contains variables, *st_out/2* blocks until they are assigned values. After the operation has been completed, Status is assigned an empty list. For users convenience *st_out/1*, without the synchronizing output argument, *Status*, was also implemented.

- *st_inp(Data, Status)* — operation similar to *inp()* in Linda. Status indicates if the required tuple was found, that is, it is assigned a constant *1* if *STSData*, matching *Data*, was found in STS and removed from it and *0* otherwise. If *Data* contains variables they will be assigned appropriate values. For example, after *st_out({ 1, { 2 }, 3}, _)* has been completed, *st_inp({ 1, A, B }, Status)* will succeed, that is A will become { *2* }, B will be assigned a value *3* and *Status* equal to *1* will indicate success.

- *st_rdp(Data, Status)* — same as *st_inp/2*, but the matching object (if there is such) is not removed from STS.

- *st_in(Data)* — this process will not be reduced until an appropriate object is found in STS. Similarly to *in()*, a matching object will be removed from STS.

- *st_rd(Data)* — this process will block until an appropriate object is found and read (but not removed) from STS.

It should be emphasized that after having put an object to STS, in order to be read it needs no longer to be an argument (or part of an argument) to any process. Such an object may be accessed by any STRAND process using only extended model operations. If an object has been *out*ed on a given processor, it is not moved from it by internal STS managing processes. Any object from STS is accessible from all STRAND processors, and all calls are *transparent* — there is no difference in syntax and semantics between accessing local and remote STS objects.

plementation is based on C-Linda, in this subsection we use function and constant denotational conventions of C. Linda introduces a notion of a **tuple space** (called hereinafter TS), that is, an abstract model of virtual distributed associative memory. The tuple space may be used by all processes in a uniform fashion, regardless of the fact if data to be accessed are stored locally or remotely. The only entities to be found in the tuple space are *tuples*. They may come in two flavours. *Passive* tuples are sequences of typed fields with values. An example can be *("str1", 7)*. *Active* tuples are computations that turn into passive tuples after a completion of their tasks. As a spawning of new processes is very easy in concurrent logic committed choice languages in general and in STRAND in particular, active tuples and their evaluation mechanism will not be considered. Passive tuples can be manipulated using the following operations, each of them taking a tuple as its argument:

- **void out()** — enters its argument tuple into TS, i.e.: *out("str1", 7)*
- **void in()** — blocks until the tuple matching its argument tuple is found in TS. When a matching tuple is found, it is removed from TS and required argument tuple fields may be assigned a value. For example, *in("str1", ? IntVal)* will take (possibly after an arbitrary long period of waiting) a two field tuple out of TS. The first field of this tuple must be a string *"str1"*, the second — integer, whose value is assigned to *IntVal*.
- **int inp()** — a predicate version of *in()*. If a matching tuple is found a constant *1* is returned. If there is no matching tuple in TS — *inp()* returns immediately with *0*.
- **void rd()** — similar to *in()*, with one important difference: the matching tuple is left without any change in TS.
- **int rdp()** — a predicate version of *rd()*

2.3 STRAND extended model

The STRAND extended model consists of the standard STRAND computational model and STRAND tuple space (referred to as STS) communication semantics. Syntax and description of these operations are given in the next subsection. This subsection presents a general view of the extended model.

Semantics of STS operations is very similar to that defined by Linda. To a large extent STS has been designed like a Linda tuple space, described earlier. However, in STRAND it has a different meaning than a tuple space in, for example, *C*. In sequential languages Linda allows computations to be parallelized. As an addition to a parallel logic programming language, Linda operations do not provide anything basically new because processes may transfer data using LP protocols. However, there is a large class of problems that could benefit from using Linda operations.

First, communication protocols may be considerably simplified. Processes using Linda operations become strongly decoupled. Instead of setting up a connection between two processes using a shared variable, these processes may use STS as a shared memory and access it whenever there is such a need. Interprocess communication becomes symmetric, whereas in the LP model sharing a variable forces processes to communicate over it using a *producer—consumer* protocol. In STRAND a connection between processes is *permanent* while our model offers *temporary* connections, that is, a process interfaces with STS for the duration of the call only. To use an analogy from computer networks, STRAND communication resembles circuits while our model offers more a flexible connectionless protocol.

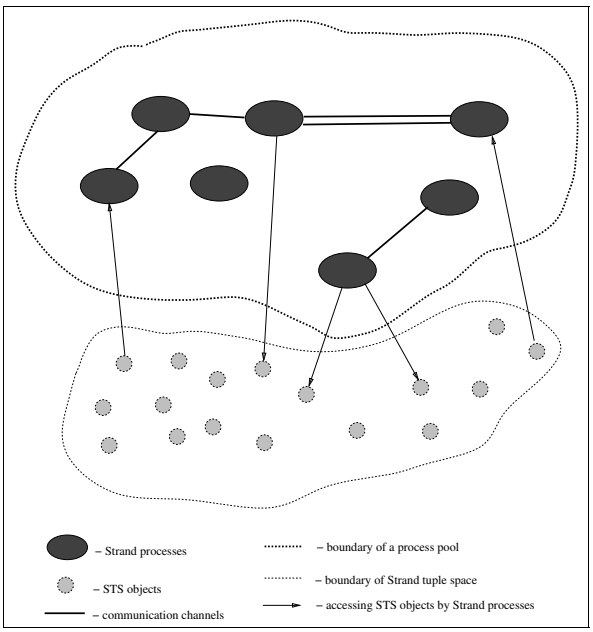

Figure 1: STRAND processes using STRAND tuple space.

power to do without all LP techniques. However, efficiency gains and simplicity achieved for a certain class of problems validates our effort.

Linda has a twofold role in our project. First, its operations serve as a model for the ones we added to STRAND. Second, a public domain C-Linda version, POSYBL-1.102[1] was used as a basis of our implementation.

2 Extending existing model

Before moving to main topics of this paper, the STRAND88 language and the idea of Linda are shortly described. The extension of STRAND with Linda—like operations is presented then.

2.1 STRAND abstract computational model

A basic abstraction of a STRAND computational model [7] is a **process pool**, that is, a dynamically changing set of concurrent processes. They communicate via shared variables and data structures. If there are more than one clause matching a process to be reduced — only one of them is chosen, in a nondeterministic way. Variables in processes from a process pool can be assigned values only by an explicit assignment — not by matching and subsequent binding. STRAND is a committed choice language.

2.2 Linda

Linda is not an independent language on its own. It is rather a small set of operations added to the existing language and facilitating the distributed computation execution [1]. As our im-

[1]POSYBL-1.102 was written by G. Sxoinas from the University of Crete

Extending the Parallel Logic Programming Paradigm with Linda-like Operations

Grzegorz Czajkowski, Krzysztof Zieliński
Institute of Computer Science,
University of Mining and Metallurgy,
al. Mickiewicza 30, Krakow
Poland *

Abstract

The aim of this paper is to analyze the concept of extending the parallel logic programming paradigm with Linda-like operations, thus facilitating distributed data storage, access and management. Data are accessed in a uniform fashion, regardless if they are stored locally or remotely. Neither perpetual processes to maintain the distributed data nor establishing (sometimes very complex) of interprocess communication channels to broadcast messages or collect results is needed, in contrast to the traditional parallel logic programming approach. However, relevant parallel LP techniques cannot be discarded since our model does not offer enough synchronizing power for some classes of problems. The paper describes operations added to STRAND and the semantics of the extended computational model. A comparison of standard logic programming techniques with the ones offered by our extended model is presented. Every example is followed by a detailed analysis. Finally, we present some programming problems that are solved in our model completely differently than in the standard STRAND.

1 Introduction

Concurrent logic programming languages offer a high-level, elegant, natural and often efficient approach to programming distributed systems [8]. The growing number of such languages, ranging from experimental university research projects to fully fledged commercial products proves their attractiveness. Many applications written in parallel logic languages show that even most intricate problems of distributed computing may be effectively solved using only pure logic programming techniques. However, there is a price to be paid for retaining a clear logical meaning of a program. Managing distributed (possibly ordered) data with (preferably fast) multi-process access and complex interprocess communication protocols are not easy tasks to accomplish in the pure LP. Programs that have to implement features mentioned above are often clumsy, difficult to read and inefficient.

The purpose of this paper is to present how an extension of the concurrent LP computational model by providing an abstraction of virtual distributed associative memory could influence programming in these languages. As a case study STRAND has been chosen and extended with operations similar to those of Linda. Our extended model does not offer enough synchronizing

*This work was sponsored by Komitet Badań Naukowych under grant no.: 8.0077.91.01