the idiosyncrasies of the new Linda and change the form of calls to *in()*, *inp()*, *rd()*, *rdp()* and *out()*.

8 Conclusions

STRAND execution model may be extended with a set of Linda-like operations and a concept of tuple space. The 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. Important factors influencing the efficiency of the extended model have been examined. The extended model may in future have much better performance than it has now if the C code of commercial Linda implementations can be incorporated into the STRAND Abstract Machine.

Acknowledgements

Both authors were sponsored by Polish Research Council (KBN) under grant no.: 8.0077.91.01.

References

1. N. Carriero and D. Gelernter. Applications Experience with Linda. In *Proceedings of the ACM/SIGPLAN PPEALS*, 1988.
2. I. Foster and S. Taylor. Strand — New Concepts in Parallel Programming. Prentice-Hall Inc., 1990.
3. L. Sterling and E. Shapiro. The Art of Prolog. MIT Press, 1986.
4. E. Tick. Parallel Logic Programming. MIT Press, 1991.
5. E. Tick E. and C. Banerjee. Performance Evaluation of the Monaco Compiler and Runtime Kernel. In *International Conference on Logic Programming*, MIT Press, June 1993. Also available as University of Oregon CIS-TR-92-21.
6. K. Zieliński, M. Gajęcki and G. Czajkowski. Software Environments for LAN Distributed Computing — a Comparison Study. Internal Report, Institute of Computer Science, UMM, Kraków, Poland.

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

decode them and put back into STS, which results in *2N* calls to underlying Linda tuple space. In contrast, our approach would require only one call to Linda tuple space to make sure no matching tuple is available. In short, the best results would be obtained with optimizing only for a single data type. More general optimizations could lead to a degraded performance for some classes of problems.

Our system can be started with one of the three options (they exclude each other) — optimizing for lists of integers, lists of strings and tuples. If tuple-oriented optimization is chosen, a list of first-level fields with types must be specified in a command line of the system, e.g.
stl -tuple 1i 2s 3r would optimize for tuples with a first field being integer and/or a second one — string and/or a third one — real.

7 Perspectives

Our project was based on a rather inefficient Linda implementation. Availability of commercial Linda systems producing code that could be incorporated into the STRAND abstract machine could result in much better performance of the extended model. The cost of replacing POSYBL with some other compiler would then be very low because all the work has already been done.

The tables below present some performance parameters of SCA Linda.

- Transmission of data between workstations is much faster, as shown in Fig. 14.
- Time of execution of *out(1)* and *in(1)* (tuple space was empty before putting an integer into it and contained only one number, *1*, before extracting it), as compared in Fig. 19, is much better in SCA Linda than in POSYBL.

	POSYBL	*SCA Linda*
out	2.04	0.20
in	3.12	0.19

Fig. 19. Averaged time of executing *out(1)* and *inp(1)* in two Linda implementations.

In previous sections we have listed reasons why SCA Linda could not be used as a basis of our implementation and why POSYBL was chosen. However, there are no reasons why fast, commercially available Linda systems cannot be implemented with the speed similar to that of SCA Linda and open run-time system similar to those of POSYBL. With such a system, provided that its C code could be incorporated into STRAND Abstract Machine, our implementation would be much faster (about one order of magnitude) than it is now. What is also important, all the work has already been done. Destructuring tuples, rebuilding them and mapping between STS and TS objects has already been implemented. With new Linda all that would have to be done to have new, extended STRAND up and running would be to examine all

suitable. For example, programs which pass lists of integers between processes would run much faster if the list-oriented optimization policy were employed. Instead of breaking a list containing N integers into $2N + 1$ Linda tuples, it might be stored as a single Linda tuple, containing a vector of appropriate integers. Fig. 18 gives the comparison of performance between usual and optimized methods of representing lists in the Linda tuple space. Optimized STS interface operations are much faster. Because of the fact that *out()* and *inp()* are called only once in the system optimized for lists regardless of the list length, the execution time of an operation grows very slowly with the length of a list.

length	*st_out*		*st_in*	
	standard	*optimized*	*standard*	*optimized*
0	4	4	4	4
1	12	4	13	4
2	19	4	19	4
3	25	4	25	4
4	32	4	31	4
5	39	4	37	4
6	45	4	43	4
7	52	4	49	4
8	58	4	56	4
9	65	4	62	4
10	72	4	68	4
15	105	4	99	5
20	138	4	129	5
25	171	4	160	5
50	336	5	313	6
75	502	5	467	7
100	667	6	620	8
200	1329	6	1234	9
500	3313	8	3076	10

Fig. 18. Time of executing *st_out* and *st_inp* as a function of list's length.

Unfortunately, POSYBL implementation makes it impossible to transform effectively *any* STRAND data structure into a vector of integers, containing coded field types and their values. Such approach is feasible, but completely inefficient. Such *one STRAND object — one Linda tuple* mapping would cause STRAND objects to be stored as one big set in TS, without grouping them according to their types. This would introduce great overhead. For example, in order to check if a five-field STRAND tuple exists in STS the searching algorithm would have to take every Linda tuple in turn out of STS and examine it, until all have been checked or until the matching one has been found. In our approach, only the set (possibly empty) of five-field STS tuples is scanned. If there is no five-field STRAND tuple in STS and N other objects, the list-optimized algorithm must take all of them out of STS,

nesting level	*st_out*	*st_in*	analysing — 1	*analysing — 2*	*creating*
0	12	18	2	0.14	0.27
1	15	20	2	0.24	0.28
2	18	21	2	0.24	0.29
3	22	23	2	0.25	0.31
4	25	25	2	0.25	0.32
5	27	27	2	0.25	0.32
6	30	29	2	0.25	0.34
7	33	30	2	0.25	0.36
8	37	32	2	0.26	0.38
9	40	35	2	0.26	0.40
10	45	37	3	0.26	0.41
15	63	48	3	0.28	0.45
20	97	63	3	0.29	0.53
25	105	68	3	0.31	0.59
50	195	128	4	0.37	0.84
75	301	196	6	0.41	1.13
100	489	270	8	0.47	1.26
200	993	679	15	0.69	2.42
500	2493	1571	34	1.43	5.88

Fig. 17. Time of executing *st_out* and *st_inp* as a function of nesting level of STS object.

was called, which returned immediately). The column *analyzing - 2* shows the time C code needs to scan a STRAND tuple, that is to descend recursively through all nesting levels to a constant *1*. After comparing *analyzing - 1* and *analyzing - 2*, we can see that scanning STRAND data is very fast. A considerable amount of time is consumed by process providing unique identifiers, and that contributes mainly to much bigger numbers in *analyzing - 1* than in *analyzing - 2*. Measured times of building STRAND tuples are presented in the column labeled *creating*.

The second parameter influencing the speed of extended model operations is the size of the argument. As the measurements have shown, a change in the length of a string (from 16 bytes up to 1 Kbyte) that is taken and entered into STS does not practically influence the performance.

6 Optimization methods

In the previous section we have mentioned that STS interface operations are optimized for tuples that have a string as the first field and an integer value as the second field (provided that the arity is at least *2*). These fields are stored in the Linda tuple describing the top-level layout of the STRAND tuple T, instead of forming Linda tuples associated with the particular field of T. The choice of such fields and types was based on statistics — most tuples have a string as their first field and an integer as the second one. However, for some programs different optimizations might be more

5.4 Sorting lists

In this section we would like to compare performance of two programs that sort lists. In STRAND we use quicksort. In the extended model we use a program already presented (Fig. 11). It hardly has a counterpart in logic programming.

Length of a sorted list	*STRAND — quicksort*	*Extended model — program from Fig. 11*	*Ratio of the times*
1000	1244	5840	0.21
2000	2069	8760	0.24
5000	9687	17403	0.56
10000	17461	31593	0.55
15000	33903	45441	0.74
20000	49857	58973	0.84
25000	79218	72807	1.08
30000	107280	85764	1.25
35000	134654	90150	1.49
40000	157431	114826	1.36
45000	232596	127734	1.82
50000	298451	139539	2.13

Fig. 16. Average time of list sorting in standard and extended STRAND.

Fig. 16 presents results of sorting randomly chosen integers smaller than 1001 and greater than 0. Performance of the extended model program is worse for lists shorter than about 25000 . However, the time of computation grows linearly with the number of sorted integers so for longer lists this algorithm sorts faster than quicksort.

5.5 Factors influencing extended model performance

There are two important parameters which must be taken into account while evaluating extended model operations. First of them is the nesting level of STS object. Fig. 17 contains averaged times of executing *st_out(Tuple, _)* and *st_inp(X, _)*. *Tuple* had one field, { *tuple,* { .. { *1* } .. } } , where the experiment parameter was the nesting level of this field, that is, the number of parentheses pairs bracketing integer constant *1*. *X*, the argument to *st_inp/2*, is an unbound variable. There are no other objects in STS at the time *st_inp/2* is called, so no time is spent on searching STS.

The dependency is almost linear. The time of executing *st_out/2* grows faster than that of *st_inp*. The reason is that entering the structured object into STS requires calling the process providing the networkwide unique identifiers.

Two separate factors contribute to the overall time of entering and extracting a tuple out of STS — calling POSYBL system and destructuring or rebuilding STRAND data. The column labeled *analyzing - 1* in Fig. 17 gives the time of preparing the tuple for being entered into STS (instead of *out()*, the dummy C procedure

The total execution time of a process in STRAND may be measured using *time_call({ Module:ProcessName , Time }, Status)* system call. The accuracy is one millisecond. This call returns time taken to execute *Module:Call* .

POSYBL and SCA Linda use similar built-in timer utilities. The fragment of the code which execution time is measured is bracketed by *start_timer()* and *timer_split()* system calls. In both systems the accuracy is one microsecond.

The values of all measurements throughout this paper are given in milliseconds.

5.3 Managing large sets of data

Benchmark programs will have to extract and add items of data to the set of data. In the extended model, the set will be represented by STS. The user cannot influence the order in which STS is being searched. Providing data with additional tags (i.e { *input, 0* } and { *output, 7* } instead of *0* and *7*, respectively) may constrain the set of possibly matching STS objects and, in consequence, speed up a searching process, but the user cannot modify the basic STS searching algorithm.

In standard STRAND a user must program the searching procedure himself (it is usually called a *blackboard* [2]) , choosing the most appropriate data structures and algorithms. We have measured execution times of programs that add and remove data from an ordered list, an ordered but unbalanced binary tree and an AVL tree. During the whole experiment each of those structures contained about 5^6 (15625) entries. These entries (tuples) were also entered into STS. The second field of each entry consisted of an unique integer key. The first field was an empty list or a string. With the optimization mentioned in the former section, { *[], IntVal* } would be represented in TS as two Linda tuples, while { *StringVal, IntVal* } would be represented as one Linda tuple. Thus, locating the matching tuple (if there exists one) would take the same time in both cases, while extracting the tuple from STS would require two *inp()* calls in the first case and one such call in the second case.

Computational model	*Extended model*		*STRAND*		
Data structure	STS first field is an empty list	STS first field is a string	ordered list	unbalanced binary trees	AVL trees
Time of reading an item	12	9	174.4	14.9	8.3
time of adding an item	81	77	153.5	78.0	30.4

Fig. 15. Average time (in milliseconds) of reading and adding data to STS and managing different data structures in STRAND.

Fig. 15 gives averaged times of adding and reading a data item from STS and from an appropriate STRAND data structure maintained by a blackboard process. 10000 elements to be entered and extracted were chosen at random. The program written in the extended model is a viable solution for managing distributed sets of data. AVL trees are faster but the cost is a considerable complexity of the program.

packet size in bytes	*STRAND*	*POSYBL*	*SCA Linda*
16	30.04	27.56	2.71
64	30.01	27.96	3.46
256	30.06	27.90	4.17
1024	30.03	28.01	4.68

Fig. 14. Data transmission times (im milliseconds) between two SPARC-2 worksations in POSYBL, SCA Linda and STRAND systems.

between different machines. As a consequence, comparing execution times of programs that run on the same machine or are distributed over the network will give us the same information about efficiency of extended model operations versus standard STRAND techniques. To put it more concisely, we propose that the equation $t_c(S_n) \approx t_c(E_n)$ is true for all $n > 0$, where S and E denote programs written in pure and extended STRAND, respectively, n is the number of processors the computation runs on and t_c returns the time taken to transmit data between workstations during the program execution. Because of that fact we will analyze programs executed on one workstation. Performance of the extended model STS interface operations in programs executed on one workstation is influenced by two separate factors: time taken to decompose or rebuild STRAND data structure and time taken to execute appropriate Linda operations. This will be analysed in detail in one of the next sections.

There is no reason in comparison of the performance of *the same* STRAND program in two different models, since such comparison would give two identical results (because, when not used, the mere existence of the extended model operations does not influence th execution times of STRAND programs) and no information about added Linda-like operations. The conclusion is, that the performance of two completely different programs must be measured, because we are comparing different computational models, not different compilers and the quality of compiled code. The programs will be functionally equivalent but one of them will be written in standard STRAND while the other will use the extended model operations. It would be also desirable to use only added STS interface operations to communicate processes in programs written in the extended model. Programs using a mixture of interprocess communication techniques, some from pure and some from extended STRAND, would not give us accurate information about performance of added operations. In our view the most representative examples are managing large sets of data and sorting lists. Following subsections analyze performance of programs dealing with problems mentioned above. Before giving performance results, we must describe shortly how the three systems used in the project measure time.

5.2 Measuring time

Three systems, STRAND88, POSYBL and SCA Linda were used in performance measurement experiments. Each of these systems has its own timer utilities.

T. If a tuple S was extracted and S does not match T, S is not entered into TS until the exhaustive search is over. The rationale behind such policy is that S will not be (unnecessary) examined once again by one of the recursive calls to *find_tuple()* and the searching of TS will be faster. If S were to be returned to TS *before* the recursive call in *find_tuple()*, in the worst case the algorithm would never terminate (in practice, stack overflow would stop computations) . S would be taken from TS, examined, put back, taken from TS ... over and over again, because the TS searching is nondeterministic.

In Fig. 13 we have omitted, for readability's sake, declarations of variables and definitions of some functions. One point that requires explanation is that *TupleRestore()* and *match()* can actually be performed as one function. In such a case, if a tuple S extracted from TS does not match an argument to *find_tuple()*, all Linda tuples defining S may be left in TS with the exception of the one describing the layout of the top level of S. In the simplified example in Fig. 13 such optimization is not performed.

5 Performance study

As has already been shown, for a wide class of problems the extended model offers smaller, more compact and readable solutions. This alone makes the idea of using Linda-like operations in STRAND interesting and validates our effort. This section gives comparison of performance between certain problems written in pure and extended STRAND.

5.1 Choice of benchmark problems

In previous sections we have listed reasons why the POSYBL Linda implementation was chosen as a basis of our project. POSYBL has some features, not offered by other available Linda implementations, suited very well for our needs. Due to them we could implement all presented operations and develop and test new programming techniques in the extended model. However, being a public domain product, POSYBL is not very efficient. In particular, it does not offer very high transmission speeds. It sends data between workstations slightly faster than STRAND but is still much slower than some of the commercial products[2]. On the average, transmitting data between workstations using POSYBL is about 7 per cent faster than using STRAND, as shown in Fig. 14. This number is the limit of any possible speed gains in programs sending large amounts of data between workstations. Thus, moving the responsibility for data transport between workstations from the STRAND kernel to Linda internal tuple space managing processes does not result in considerable efficiency gains.

The fact that STRAND and POSYBL transmit data at roughly the same speed implies that the extended model will not be able to provide faster transmission

[2] The authors have carried out extensive tests, comparing data transmission speed of numerous LAN distributed computing systems. Both STRAND and POSYBL are one order of magnitude slower than the fastest systems, PVM, P4 and SCA Linda v.2.4.6 [6].

```
void st_inp(Object, Status) — visible by the STRAND user
    switch (get_type(Object))
        case VAR: get any object from STS ; break;
        case INT: * Status = inp(lint(INT), lint(Integer(Object))); break;
        case STRING: * Status = inp(lint(STRING), lstring(String(Object))); break;
        case TUPLE: * Status = find_tuple(Object); break;
        case ELIST: * Status = inp(lint(ELIST), lint(0), lint(0), lint(0)); break;
        case LIST: * Status = find_list(Object); break;

int find_tuple(Tuple)— searches TS for the tuple matching its argument
    ok1 = ok2 = 0;
    if ((get_type(arg = GetArg(Tuple, 1))) == STRING)
        ok1 = 1;
        str_val = String(arg);
    if (arity > 1)
        if ((get_type(arg = GetArg(Tuple, 2))) == INT)
            ok2 = 1; int_val = Integer(arg);
    if (ok1)
        if (ok2)
            if (!inp(lint(TUPLE), qlnint(&id_tab, &id_tab_len),
                lint(arity), lstring(str_val), lint(int_val)))
                    return 0;
        else
            if (!inp(lint(TUPLE), qlnint(&id_tab, &id_tab_len),
                lint(arity), lstring(str_val), qlint(&int_val)))
                    return 0;
    else
        if (ok2)
            if (!inp(lint(TUPLE), qlnint(&id_tab, &id_tab_len),
                lint(arity), qlstring(str_val), lint(int_val)))
                    return 0;
        else
            if (!inp(lint(TUPLE), qlnint(&id_tab, &id_tab), lint(arity),
                qlstring(&str_val), qlint(&int_val)))
                    return 0;
    NewTuple = TupleRestore(id_tab, arity, ok1, str_val, ok2, int_val);
    if ((flag = match_objects(NewTuple, Tuple, &BindList)) == 1)
        bind_vars(BindList);
    else
        flag = find_tuple(Tuple);
        st_out(NewTuple, &int_val); returning NewTuple to TS
    return flag;
```

Fig. 13. Definition of *st_in()* given in the pseudocode.

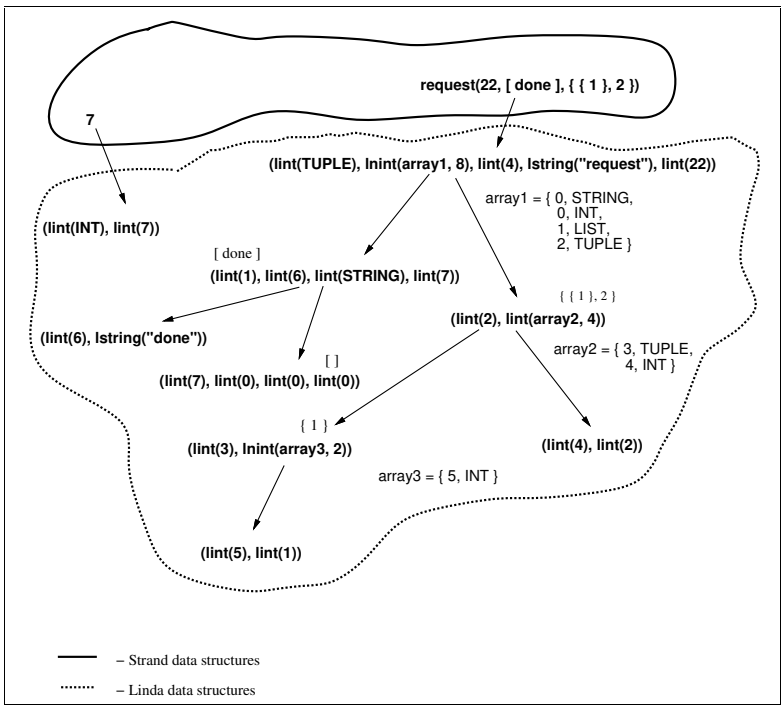

Fig. 12. Destructuring Strand data.

i.e. *inp(7, Status)*. The Linda code calls *inp(lint(INT), lint(7))* and searching TS is done by TS managing processes.

If the user calls *st_inp(Var, Status)* and *Var* is an unbound variable (every STS object matches variable), the Linda code searches STS in the following order: tuples, lists, integers, strings, reals. Checking if there are objects of one of these types in STS is done by TS managing processes but looping over types is the task of added code.

The most frequently exchanged information between STRAND processes are tuples, usually not nested or with a small nesting level. STS first level tuples are represented with three additional fields - arity, the value of first field (if it is a string) and value of second field (if it is an integer). Using these three fields speeds the searching up. Adding arity to the Linda tuple representing the first level STRAND tuple limits searching to STS tuples of the same arity. Without this limitation, *inp()* would carry on the search in the space of all tuples, irrespective of their arity, and in the worst case return all tuples.

Two other fields may limit the set of STS matching tuples even further. If the first two fields have appropriate types, their values are stored in the Linda tuple. Fig. 13 shows an implementation of *st_inp()* with searching optimizations mentioned above. The function *find_tuple()* scans TS for the first tuple matching the given one,

restoring. The first level STS tuple is remembered in TS as a five field POSYBL tuple: *(lint(TUPLE), lnint(id_table, arity * 2), lint(arity), lstring(field1), lstring(field2))*. A vector of integers, *id_table*, contains a sequence of pairs: *(FieldId, FieldType)*. The meaning of *FieldType* is obvious. *FieldId* is network-wide unique number, associated with a particular field. This facilitates fast tuple fields searching, for there is only one unique hash value for that field, namely *FieldId* (if *FieldId* is equal to zero, then the value of this particular STRAND data is stored directly in the first-level Linda tuple — this is used for optimization, as shown below). A process providing unique numbers is written in POSYBL.

The last three fields are used for optimization (we show only one possible approach to optimization here — one of the later subsections shows other optimization methods). If the first argument is a string, the fourth field gets its value. Similarly, if the second field is an integer its value is stored in the fifth field. A description of *st_inp/2* will show how they are used for efficient TS searching.

The first-level STS list is represented as a four-field tuple: *(lint(LIST), lint(head_id),*
lint(head_type), int(tail_id)). The second and fourth fields contain unique identifiers of the head and tail of the list, respectively. The third field is used for optimization. First level empty list, which is represented by STRAND as a tuple of arity *0*, in POSYBL becomes *(lint(ELIST), lint(0), lint(0), lint(0))*.

So far we have discussed the first level objects only. Lower level objects (fields of tuples or list components) will be scanned like the first level objects, but with two differences. Instead of having a predefined integer constant describing a data type as the first field, lower level objects will have unique identifiers there. It will prevent an unauthorized access, binding the field with the structured object. The second difference is that lower level tuples do not carry arity and values of their first two fields with them. It is a reasonable approach, since including this information in a Linda tuple describing a nested STRAND object would not speed up searching. A unique value of the first field of Linda tuple suffices. Fig. 12 shows how STS objects are represented in TS. Besides, if an object is a string composing the first field of the first level tuple or an integer composing its second field, it is not stored in Linda tuple space. Information about its value is present in Linda tuple describing a first level STRAND object.

The definition of *st_out/2* is rather simple and will not be given here. A structured object is scanned down recursively and if data is a variable, the procedure uses STRAND supplied *WaitB()* in order to wait for its binding. Only after all structure components are stored in STS, a Linda tuple giving the information about the first level of the object is *out*ed.

Implementing st_inp/2 .

An object in STS may be accessed by processes from a STRAND process pool. If some process calls *st_inp(Y, Status)*, STS must be searched. This search terminates when either some STS object, X, matches Y or there is no such an object X. The searching of STS should be as efficient as possible.

The searching of STS for the first level objects of basic types is rather simple. A user may be interested in finding out whether a given object is in the tuple space,

The last point requires some explanations. There are only three out of five Linda operations acting on passive tuples implemented in POSYBL, namely *out()*, *in()* and *rd()*. We have changed the code of the last two and turned them into their predicate counterparts, *inp()* and *rdp()*. In that way for example *st_inp/2* may be implemented directly, using *inp()*.

We would like to stress that there is no connection between the STRAND process pool and the STRAND tuple space. Processes from the pool may access STS but STS itself is managed by processes from outside the pool.

Using the existing Linda system code was only the first step towards a successful implementation. A second, more interesting for us, was how to represent the STRAND data in the Linda tuple space. The next subsection covers this topic.

4.2 Data representation

Before describing a translation between STS objects and Linda tuples we must describe what kinds of data may be stored in POSYBL TS.

POSYBL tuples can be composed of the basic type data (e.g. *int*, *char*) and vectors of these types. A value of every tuple field must be contained inside a type indicating function, e.g. *(lint(7), lstring(char_ptr))*, *(lint(int_val), lnint(int_ptr, vect_len))*. If the type indicating function starts with a letter '*q*', it denotes data which value will be taken from TS (we will call such fields *q-fields*), i.e. *in(lint(7), qlint(&int_val), qlstring(&str_ptr))* will remove the first matching tuple from TS and assign appropriate values to *int_val* and *str_ptr*. No tuple may have a q-field as its first field, that is, the first field of every tuple must have a value. POSYBL uses the type and value of this field for hashing data in TS. As will be shown later, this information was very important for our implementation.

A foreign procedure may access the STRAND data via STRAND system calls. These calls provide information about a type and value (if any) of the STRAND data. A user can also build the STRAND data inside foreign procedure and then pass them to STRAND. We will discuss detailed implementation issues on two examples - *st_out/2* and *st_inp/2*.

Implementing *st_out/2* Objects of basic types (e.g. integers, strings) may be transformed into POSYBL tuples directly. They get an additional tag field, indicating their type and the fact that they are the first level objects (that means, they actually were arguments to *st_out/2* and not components of some argument). Thus *st_out(1, _)* will call POSYBL in *out(lint(INT), lint(1))*, where *INT* is an integer constant. Such policy causes the first level STS objects of the same basic type to be stored in TS with the same hash key. That does not degrade a performance because almost all STRAND programs pass tuples between processes and not integers or strings. When strings are passed (i.e. '*done*' messages indicating a work completion in farmer-worker applications) they are usually identical strings and searching of a hash table for them is very fast. Similarly, it is rather unusual to pass big numbers of different integers or reals between processes. Objects of these types are usually encapsulated within a tuple.

A tuple must be linearized, that is, its tree-like structure must be transformed into a set of Linda tuples. This transformation must allow for a subsequent tuple

```
LowerLimit and UpperLimit must be integer constants

sort(Sorted) :- sort(LowerLimit, UpperLimit, [], Sorted).
sort(N, Limit, List, Sorted) :- N > Limit | Sorted := List.
sort(N, Limit, List, Sorted) :- otherwise | st_inp(N, Found),
    decide(N, Found, N1, List, List1), sort(N1, Limit, List1, Sorted).
decide(N, 1, N1, L, L1) :- N1 := N, L1 := [ N | L ].
decide(N, 0, N1, L, L1) :- N1 is N + 1, L1 := L.
```

Fig. 11. Sorting objects in STS - problem's complexity is proportional to the number of elements in STS.

4 Implementation issues

There were two main problems to be solved in implementing our project:

- Providing distributed STRAND tuple space maintenance.
- Representation of STRAND data structures in C.

This section gives some insight into how we have coped with the above mentioned.

4.1 Maintaining thge STRAND tuple space

The most important decision in our project was how to implement STS. As our major objective was to assess usefulness of extending STRAND model and we wanted to concentrate on programming techniques in the extended model we decided to reuse existing code and base our STS implementation on the POSYBL-1.102 compiler.

This system is not as fast as commercial Linda implementations (like SCA Linda, for example) but it has three very important advantages:

- POSYBL uses the standard C compiler (*cc*), which allows the POSYBL code to be incorporated into the STRAND Abstract Machine.
- POSYBL daemons run in background even if no Linda program is being executed. Due to that fact, POSYBL may be started before STRAND. STS operations will have a direct access to Linda tuple space (TS) during the whole time of computations. In contrast, the SCA Linda system runs only during the computation of some Linda program. The fact that POSYBL runs all the time may be utilized in one more way. Instead of reading the data from a disk they can be kept in STS from one STRAND session to another. Our implementation was also successfully used to provide communication between two users, each of them using STRAND.
- POSYBL system is supplied with its C sources. We have taken advantage of this fact and changed slightly definitions of some operations to make them better suit our requirements.

– 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, 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.

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.

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) 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).
```

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

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
- 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

3.5 Short circuit and process termination detection

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).
```

Fig. 9. Collecting all solutions

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* 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).*

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). 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.

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.
```

Fig. 8. *bounded buffers* **solved in different computational models.**

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.

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.
```

Fig. 7. Accessing distributed data.

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.

(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.

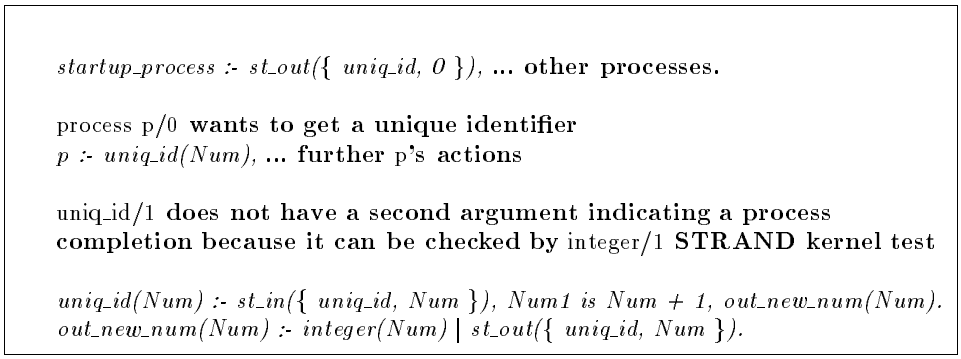

Fig. 6. Providing network-wide unique identifiers.

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 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*.

```
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) :- | 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)!.
```

Fig. 5. The extended model solution to the single-producer and multiple-consumers problem.

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 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 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

the initial process pool. This is a slight variation indeed, for putting k processes in the process pool is identical with putting only one process there which contains the above mentioned k processes in its body. This single process is immediately reduced.

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.**

Fig. 4. *incomplete-messages* **protocol in LP and its equivalent in the extended model.**

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 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

A task is a tuple { TaskNumber, TaskData } , **(where** TaskData **is an 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 the computed value and** Done **becomes an empty list.**

Fig. 3. Extended model solution to *farmer-worker* **class of problems.**

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

- *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. 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

a) solution offered by the standard logic programming approach

```
startup_process :- producer(Channel),consumer(Channel), consumer(Channel).
producer(Channel) :- Channel := [ works | Channel1 ], producer(Channel1).
consumer([ 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({ works, 1 }),st_out({ works, 2 }), producer.
consumer(Num) :- do_work, st_in({ works, Num }), consumer(Num).
```

Fig. 2. Two approaches to the simple producer-consumer protocol.

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.

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.

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 — a list, a 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.

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 implementation 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

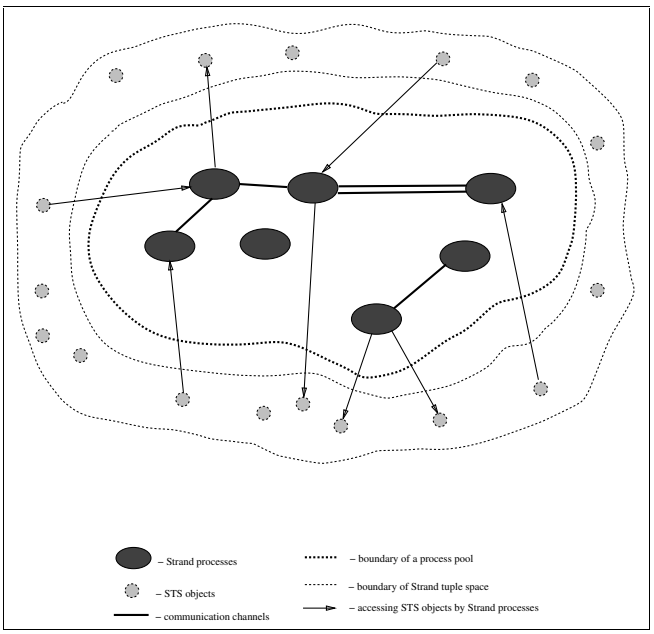

Fig. 1. STRAND processes using STRAND tuple space.

overview of our implementation of such a system. 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 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

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

How Concurrent Logic Programming Could Benefit from Using Linda-like Operations

Grzegorz Czajkowski and 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, the implementation and performance issues are addressed. Linda plays a twofold role in our project — first, it serves as a model of our extension and second, one of Linda implementations provides underlying communication mechanisms.

Keywords: Concurrent Logic Programming, Linda, Blackboards

1 Introduction

Concurrent logic programming languages offer a high-level, elegant, natural and often efficient approach to programming distributed systems [4]. 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. We also present an