

References

1. E. Astesiano, E. Zucca, "Parametric channels via label expressions in CCS*", Theoretical Computer Science, Vol 33, 1984.
2. B. Berthomieu, T. Le Sergent, "Programming with behaviors in an ML framework, The syntax and semantics of LCS", European Symposium in Programming, Edinburgh, April 1994, Springer Verlag LNCS Vol 788, 1994.
3. B. Berthomieu, C. le Moniés de Sagazan, "A Calculus of Tagged Types, with Applications to Process Languages", TAPSOFT Workshop on Types for Program Analysis, Aarhus, Denmark, May 95 (DAIMI report PB-493, University of Aarhus).
4. G. Berry, G. Boudol, "The Chemical Abstract Machine", Theoretical Computer Science, Vol 96, pp. 217-248, Elsevier 1992.
5. W. H. Burge, "Recursive Programming Techniques", Systems Programming Series, Addison Wesley, 1976.
6. G. A. Andrews, "Concurrent Programming, Principles and Practice", Benjamin/Cummings Publishing Company, 1991.
7. A. Giacalone, P. Mishra, S. Prasad, "Facile: A symmetric integration of concurrent and functional programming". Int. Journal of Parallel Programming, 18(2), April 1989.
8. ISO, "ISO-LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour", Int. Standard ISO 8807, ISO, 1989.
9. T. Le Sergent, B. Berthomieu, "Incremental multi-threaded garbage collection on virtually shared memory architectures", Int. Workshop on Memory Management, St. Malo, France, Sept. 1992.
10. T. Le Sergent, B. Berthomieu, "Balancing load under large and fast load changes in distributed computing systems - A case study", CONPAR 94 - VAPP VI Third Joint Int. Conference on Vector and Parallel Processing, Linz, Austria, Sept. 94.
11. T. Le Sergent, "Méthodes d'exécution et machines virtuelles parallèles pour l'implantation distribuée du langage de programmation parallèle LCS", Ph.D. thesis, Feb. 93.
12. D. Matthews, "A distributed concurrent implementation of Standard ML", EurOpen Autumn 1991 Conference, Budapest, Hungary, 1991.
13. D. Matthews, T. Le Sergent, "LEMMA, A Distributed Shared Memory with Global and Local Garbage Collection", Int. Workshop on Memory Management, Kinross, Scotland, Sept. 1995.
14. B. C. Pierce, D. Rémy, D. N. Turner, "A Typed Higher-Order Programming Language Based on the Pi-Calculus", Edinburgh University, 1993.
15. R. Milner, *Communication and Concurrency*, Prentice Hall international series in Computer science, C.A.R. Hoare Ed, 1989.
16. R. Milner, J. Parrow, D. Walker, "A Calculus of Mobile Processes", ECS-LFCS-89-85, LFCS report series, Edinburgh University, 1989.
17. C.M.P Reade, "Elements of Functional Programming", International Computer Science Series, Addison Wesley, 1989.
18. J. H. Reppy, "CML: A higher-order concurrent language", ACM SIGPLAN Conf. on Programming Language Design and Implementation, SIGPLAN Notices 26(6), 1991.

5. Conclusion

Among the parallel-functional languages designed for similar purposes [7, 12, 18], LCS is the sole strictly based upon CCS for its naming aspects. In terms of expressiveness, CCS naming forbids label passing. On another hand, one does never have to declare communication ports or pass them as parameters, this greatly simplifies programs, and the basic set of CCS process combinators is fairly easily understood. The extension LCS brings to CCS, and its embedding into SML, solves most of the limitations of CCS in terms of expressiveness. There are also good perspectives for verification tools for a parallel subset of LCS, based on the many available methods for proving process equivalences in CCS's style calculi (see e.g the work around the LOTOS [8] specification language).

Communication in LCS is certainly more difficult to implement than in languages having "first-class" channels; the LCS treatment amounts to a dynamic binding of channels to communication ports. This could be costly in terms of efficiency if naively implemented; LCS implementations uses a software cache for speeding accesses to channels (that could not be described here); channels are searched when first accessed (bound to a communication port) and then accessed directly in the cache; this greatly reduces the cost of dynamic binding on most programs. Still about efficiency, an interesting feature of LCS is that the stack of a thread is empty when it performs expansions or interactions, making thread creations and suspensions very fast. This results from the strict layering between processes and functions.

LCS allows one to comfortably experiment with the concurrent programming concepts introduced by CCS and related behavioral formalisms, and to gain experience in using these paradigms. It is also a valuable teaching tool for these concepts. LCS includes all capabilities of CCS and inherits most of its theory. Concurrent applications can be written abstractly so that users may reason about their programs, and they can be refined to run with good performances. The fact that abstract enough specifications and efficient enough implementations can be written in a single framework is certainly an advantage for formal development of programs.

Among the results of the experiment is the fairly general implementation technique described in Section 4, that could be used for implementation of many other process languages. It has a number of unique features (not all of them discussed here) such that both strong and light processes, and a general technique for handling preemption. Our short term goals include completion of parallel and distributed implementations of the language, on top of the memory management layer discussed in the previous section. Introduction of some synchrony will be investigated too, to handle some classes of reactive applications such as multimedia applications.

of ready threads. There are many possible methods to schedule the threads in the system, locally on each processor and globally; LCS implements the following:

- Each physical processor is assigned an abstract processor (t,R) , in which t is the active thread and R is a queue of ready threads;
- Upon a process composition, one of the threads created is suspended locally in R ;
- Upon an unsatisfied communication offer, the offer is suspended in the relevant channel queue. Upon communication, the receiver is resumed and the sender is suspended in R .
- When the active thread of a processor terminates, or is suspended, the next non-preempted thread found in the local R register is resumed, if any. If none is found (`DeQueue` returns $(\mathbf{0},\text{nil})$ then), then a global protocol is initiated that attempts to import locally a thread found in the R register of another processor (this is abstracted by the global “migrate” transition in Table 4.3).
- Time-slicing prevents diverging or looping threads to take the local processor forever. As already discussed, it also serves to terminate preempted threads.

Two implementations have been investigated: a sequentially running version, and a multi-processor version discussed in [11].

The sequential implementation obviously implements a degenerate abstract machine with parameter $n=1$, and without migration transition. In parallel implementations, a load-balancing algorithm [10] implements a more flexible version of the migrate rule of Table 4.3. Its characteristics takes into account the fact that an LCS application may create thousands of micro-processes in a short time. Each processor performs a source or server algorithm for migrating processes when its load (measured by the length of its local resumption queue R) crosses some assigned lower or upper bound; these bounds are dynamically adjusted.

4.4 Memory management

The parallel LCS abstract machine does not make precise any physical representation for the data it handles. However, this problem is an important issue in distributed implementations of the abstract machine. The choice one is faced with in that case is whether the different physical processors should operate in the same address space or in disjoint address spaces. LCS distributed implementations use the first approach, this both simplifies implementation of the abstract machine, and avoids address space conversions when transferring data from a process to another.

Practically, the parallel abstract machine is implemented on top of a memory management layer implementing a Distributed Shared Virtual Memory, combined with Garbage Collection Services. That memory management layer is described in length in [9, 13]. The store is paginated; the pages are loaded on a processor when accessed; the DSVM and GC protocols enforces consistency of the pages holding updatable data. Garbage Collection is generational, with minor collections performed locally on each processor, and major collections performed globally and incrementally.

channels here, computed so that two synchronizable threads with opposite polarity may communicate iff they refer to the same channel. The map associating channels with ports is kept in the local registers L; register Z is a shared channel server.

Besides being a method for matching threads, channels can be conveniently used to implement rendez-vous communication efficiently. A thread offering an interaction currently not matched by any other thread may be suspended until such a partner is created. To speed-up partner search, the channels can be used as references to pairs of queues holding the threads suspended on input on that channel, and those suspended on output, respectively. Note that these queues may be simultaneously nonempty here, due to the choice operators. Finally, registers L and Z are defined as follows:

A port is a label associated with an extension (a value admitting equality); let Σ be the set of all labels, and V be the set of all possible extensions. Register L encodes a curried function of type (label \rightarrow extension \rightarrow channel). Register Z holds the channels, each associated with a pair of thread queues.

Initially, L_0 is defined for all labels: For each label a , $(L_0 a)$ associates a unique channel with each possible extension of a ; Z_0 maps all these channels to pairs of empty queues. Registers L and Z are updated by restrictions, relabelling and aliasing, as follows:

$$\begin{aligned} \{ /a \} (L, Z) &= ([a: l]L, [c:(nil, nil)]_{c \in \text{Ch } Z}) \\ &\quad \text{where } l \text{ is any bijection from } V \text{ to a set of channels } \text{Ch} \text{ disjoint from } \text{Dom}(Z). \\ \{ a/b \} (L, Z) &= ([b: L a]L, Z) \\ \{ a\#x/b\#y \} (L, Z) &= ([b: [y:L a x](L b)]L, Z) \end{aligned}$$

As can be seen, Restrictions renew the channels associated with a label, extending Z, while Relabelling and Aliasing override the current mapping L. The matches predicate \mathcal{M} in Table 4.1 can be defined as follows, replacing paths by pairs (L,P):

$$\begin{aligned} (a\#x, (L, P)) \mathcal{M} (b\#y, (L', P')) &\Leftrightarrow L a x = L' b y \wedge P // P' \\ \text{where } P // P' &\Leftrightarrow P = \emptyset \vee P' = \emptyset \vee (\forall ((t, a), (t', a')) \in P \times P'. t = t' \Rightarrow (a = a' \vee a * a' = 0)) \end{aligned}$$

Upon an input (resp. output) offer on port $p\#v$, a thread searches a partner for communication in the output (resp. input) suspension queue Q of channel (L p v). The predicate “NoMatch(P,Q,A)” is true if none of the threads (S',E',C',L',P') in queue Q satisfies $\neg(\text{Preempted}(P', A)) \wedge (P' // P)$. In that case, the current thread is suspended in Q (Z is updated), otherwise communication occurs and the preemption information of both partners are committed.

The actual implementation is closely based on that abstraction. Register L is implemented by a balanced tree, associating with each label another balanced tree, the latter tree associating channels with the possible extensions. The balanced trees at both levels, which represent infinite structures, are dynamically extended.

Sharing a finite number of processors, Scheduling

The number of available physical processors is typically much smaller than the number of threads in an application. This is why our abstract machine does not create a virtual processor every time a thread is created, but rather add it to the local queue R

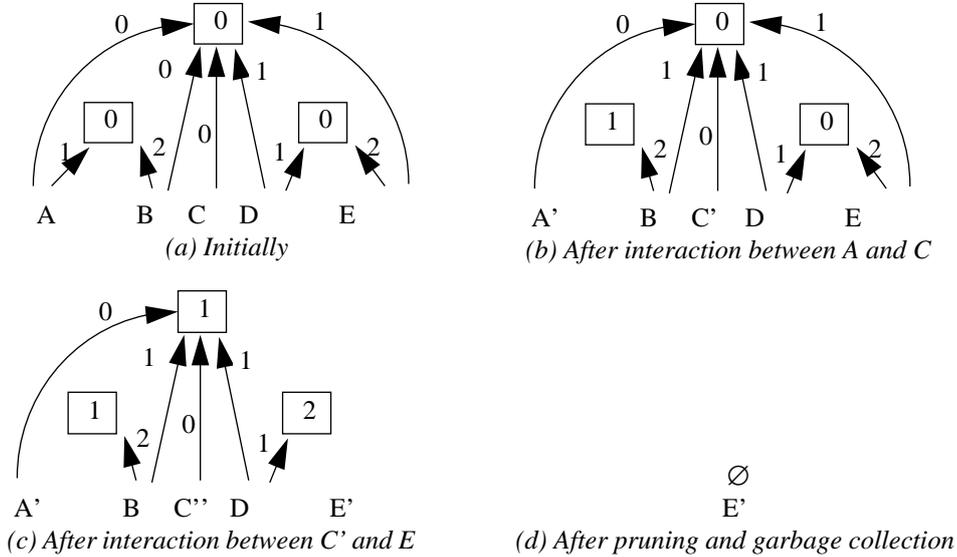


Figure 4.4 Management of Preemption Information

This is conveniently implemented by a time-slicing mechanism that periodically suspends the current thread in R (if it was preempted, then it will not be resumed again, from the local resumption transition) That time-slicing transition, not shown in Table 4.4, can be abstracted as follows, in which (time expired) is an external event:

$$\frac{(\text{time expired})}{t, R \rightarrow t', R' @ t \quad \text{where } (t', R') = \text{DeQueue}(R, A) \quad (R \neq \text{nil})}$$

Obsolete (i.e. preempted) threads are removed when found by functions DeQueue, NoMatches and Match, that scan thread queues to resume or match threads, and by the interaction transitions. The remaining obsolete threads are removed by a dedicated pruning routine.

The actual implementation is close in spirit to this description, the coding details are left out, as well as optimizations (it would create less tokens). The pruning routine is combined with garbage collection; it scans all process queues in the system, removing the preempted threads and simplifying the P registers of the remaining threads (tokens which have been set are removed). The tokens not referenced in any P register of any thread are then “naturally” removed by the garbage collector, keeping the size of the preemption information to its necessary minimum.

Naming, communication channels

The definition of predicate \mathcal{M} in Section 4.2 sums up two conditions that we called *synchronizable* ($//$) and *name matching*. The preemption registers can be used here to compute the synchronizable condition too, since we have $\pi // \pi'$ iff $\pi \mathcal{P} \pi'$ or $\pi' \mathcal{P} \pi$. To encode the latter condition on names, a solution is to use aliases for labels, called

Preemption, Pruning

Removing the preempted threads as soon as an interaction occurs, as suggested in Table 4.1, would be costly in practice. An alternative is to delay pruning by recording in some shared register which threads have performed preemptive reactions, and preventing preempted threads to interact. Removal of the preempted thread (*pruning*) may be taken care of by a separate mechanism. This strategy is implemented in Table 4.3 with the local P registers and the shared A register. The simple technique explained here is a fair abstraction of the implementation; it is also close in spirit to the implementation of preemption in [12], though the symmetry of the persistence predicate there allows a simpler treatment.

An *access* is a nonnegative integer. A *preemption token* is a reference to an access. The *preemption* register P in each thread holds a set of pairs (t,a), each holding a token and an access. The tokens are shared by all threads and are kept in register A (A maps tokens to accesses, [t:c]A is the function extending A with the pair (t:c)). The contents of registers P and A are determined from their previous contents and the composition operators. Initially, $A_0 = \emptyset$ (the empty map), and $P_0 = \emptyset$. The functions computing preemption information in Table 4.3 can be defined by:

$$\begin{aligned} \wedge (P,A) &= (P,P,A) \\ \vee (P,A) &= (P \cup \{(t,1)\}, P \cup \{(t,2)\}, [t:0]A) \quad (t \notin \text{Dom}(A)) \\ < (P,A) &= (P \cup \{(t,0)\}, P \cup \{(t,1)\}, [t:0]A) \quad (t \notin \text{Dom}(A)) \end{aligned}$$

In an interaction, all processes involved *commit* their preemption information. This consists, for each positive access a found in some pair (t,a) of register P, of setting the content of token t in register A to a (meaning that the process has “taken” the token). Those pairs may be subsequently removed from register P. Function “Commit (P, etc.) A” in Table 4.3 returns both the new preemption informations (P’, etc.) and the updated A’ register. The Preempted predicate is defined by:

$$\text{Preempted}(P,A) \Leftrightarrow \exists (t,a) \in P. A t \neq 0 \wedge A t \neq a$$

Figure 4.4 illustrates preemption management on a simple system of processes. The expression evaluated is **start** ((A \ B) / \ C) < (D \ E) ; (for some A, B, C, D, E); yielding after reduction and expansion the system of processes in 4.4 (a). Only the P registers of processes are shown, the arrows represent the preemption tokens labelled by the accesses. Figures 4.4 (b) and 4.4 (b) show the effects of interactions. 4.4 (d) shows the effects of pruning and garbage collection.

It should be noted that the non-interaction transitions in Table 4.3 do not check whether the current thread has been preempted or not. In multiprocessor implementations, it may actually happen that the current thread of some processor is preempted by a thread running on another processor. In absence of side-effects, running preempted threads is semantically correct as long as they are not allowed to interact, but this is clearly a waste of computing resources. It could be prevented by a synchronization of all processors on interactions, but this would be costly. A reasonable alternative is to leave preempted threads run (thought not interact), but to make sure that they will not run for a long time without being checked that they are not preempted.

Global Transitions	
$\Rightarrow \{(S_0, E_0, C_0, L_0, P_0), \text{nil}\}, (\mathbf{0}, \text{nil}), \dots, (\mathbf{0}, \text{nil})\}, Z_0, A_0$	(initial state)
$t, R, Z, A \rightarrow t', R', Z', A'$	(context)
$\frac{\{ \dots, (t, R), \dots \}, Z, A \Rightarrow \{ \dots, (t', R'), \dots \}, Z', A'}$	
$\{ \dots, (\mathbf{0}, \text{nil}), \dots, (t, R), \dots \}, Z, A \Rightarrow \{ \dots, (t', \text{nil}), \dots, (t, R'), \dots \}, Z, A$	(migrate, $t \neq \mathbf{0}, R \neq \text{nil}$)
	where $(t, R') = \text{DeQueue}(R, A)$
Preempted (P,A)	(prune, α is a port)
$\frac{\{ \dots, (S, E, C, L, P, R), \dots \}, Z, A \Rightarrow \{ \dots, (\mathbf{0}, R), \dots \}, Z, A$	
$W, [\alpha: (I+(S, E, C, L, P), O)]Z, A \Rightarrow W, [\alpha: (I, O)]Z, A$	
$W, [\alpha: (I, O+(S, E, C, L, P))]Z, A \Rightarrow W, [\alpha: (I, O)]Z, A$	
Local Transitions (the shared registers are omitted when unchanged)	
(reduction transitions omitted, all have shape $(S, E, C, L, P), R \rightarrow (S', E', C', L, P), R$, for some S', E', C').	
$\mathbf{0}, R \rightarrow t, R' \quad \text{where } (t, R') = \text{DeQueue}(R, A)$	(resume, $R \neq \text{nil}$)
$((e, c)::S, E, \text{Enter}; C, L, P), R \rightarrow (-, e, c, L, P), R$	
$(S, E, \text{Stop}; C, L, P), R \rightarrow \mathbf{0}, R$	
$((e, c)::S, E, \text{Par}; C, L, P), R, A \rightarrow (S, E, C, L, P_1), R @ (-, e, c, L, P_2), A'$	where $(P_1, P_2, A') = \wedge (P, A)$
$((e, c)::S, E, \text{Amb}; C, L, P), R, A \rightarrow (S, E, C, L, P_1), R @ (-, e, c, L, P_2), A'$	where $(P_1, P_2, A') = \vee (P, A)$
$((e, h)::S, E, \text{Int}; C, L, P), R, A \rightarrow (S, E, C, L, P_1), R @ (-, e, h, L, P_2), A'$	where $(P_2, P_1, A') = < (P, A)$
$(S, E, \text{Restr}(a); C, L, P), R, Z \rightarrow (S, E, C, L', P), R, Z'$	where $(L', Z') = \{ /a \} (L, Z)$
$(S, E, \text{Relab}(a, b); C, L, P), R, Z \rightarrow (S, E, C, L', P), R, Z'$	where $(L', Z') = \{ a/b \} (L, Z)$
$(v::u::S, E, \text{Alias}(a, b); C, L, P), R, Z \rightarrow (S, E, C, L', P), R, Z'$	where $(L', Z') = \{ a\#u/b\#v \} (L, Z)$
$(S, E, \text{Commit}; C, L, P), R, Z, A \rightarrow$	
	if Preempted (P,A) then $\mathbf{0}, R, Z, A$ else $(S, E, C, L, P'), R, Z, A'$ where $(P', A') = \text{Commit } P \ A$
$((e, c)::x::S, E, \text{Inp}(a); C, L, P), R, Z \text{ as } [L \ a \ x: (I, O)]ZZ, A \rightarrow$	
	if Preempted (P,A) then $\mathbf{0}, R, Z, A$
	if NoMatches (P,O,A) then $\mathbf{0}, R, [L \ a \ x: (I, O @ (S, e, c, L, P))]ZZ, A$
	else $(S, m::e, c, L, P_1), R @ (S', E', C', L', P_2), [L \ a \ x: (I, O')]ZZ, A'$
	where $((S', m::E', C', L', P'), O') = \text{Match } (P, O, A)$ and $((P_1, P_2), A') = \text{Commit } (P, P') \ A$
$(m::x::S, E, \text{Out}(a); C, L, P), R, Z \text{ as } [L \ a \ x: (I, O)]ZZ, A \rightarrow$	
	if Preempted (P,A) then $\mathbf{0}, R, Z, A$
	if NoMatches (P,I,A) then $\mathbf{0}, R, [L \ a \ x: (I @ (S, m::E, C, L, P), O)]ZZ, A$
	else $(S', m::E', C', L', P_2), R @ (S, E, C, L, P_1), [L \ a \ x: (I', O)]ZZ, A'$
	where $((S', E', C', L', P'), I') = \text{Match } (P, I, A)$ and $((P_1, P_2), A') = \text{Commit } (P, P') \ A$

Table 4.3. The LCS abstract machine and the evaluation relation

$\llbracket \text{stop} \rrbracket$	$= \text{Ldcl}(\text{Stop}; \text{Enter})$
$\llbracket p \ \wedge \ q \rrbracket$	$= \text{Ldcl}(\llbracket p \rrbracket; \text{Par}; \llbracket q \rrbracket; \text{Enter})$
$\llbracket p \ \vee \ q \rrbracket$	$= \text{Ldcl}(\llbracket p \rrbracket; \text{Amb}; \llbracket q \rrbracket; \text{Enter})$
$\llbracket p < q \rrbracket$	$= \text{Ldcl}(\llbracket p \rrbracket; \text{Int}; \llbracket q \rrbracket; \text{Enter})$
$\llbracket \text{do } e \Rightarrow q \rrbracket$	$= \text{Ldcl}(\llbracket e \rrbracket; \text{Commit}; \llbracket q \rrbracket; \text{Enter})$
$\llbracket a \# e ! e' \Rightarrow q \rrbracket$	$= \text{Ldcl}(\llbracket e \rrbracket; \llbracket e' \rrbracket; \text{Out}(a); \llbracket q \rrbracket; \text{Enter})$
$\llbracket a \# e ? x \Rightarrow q \rrbracket$	$= \text{Ldcl}(\llbracket e \rrbracket; \llbracket \text{fn } x \Rightarrow (q; \text{Enter}) \rrbracket; \text{Inp}(a))$
$\llbracket p \ \{ (/a_i)_{i \in E} \} \rrbracket$	$= \text{Ldcl}(\text{Restr}(a_i)_{i \in E}; \llbracket p \rrbracket; \text{Enter})$
$\llbracket p \ \{ (a_i/b_i)_{i \in E} \} \rrbracket$	$= \text{Ldcl}(\text{Relab}(a_i, b_i)_{i \in E}; \text{Restr}(a_i)_{i \in E}; \llbracket p \rrbracket; \text{Enter})$
$\llbracket p \ \{ (a_i \# e_i / b_i \# f_i)_{i \in E} \} \rrbracket =$	
	$\text{Ldcl}(\llbracket e_i \rrbracket_{i \in E}; \llbracket f_i \rrbracket_{i \in E}; \text{Alias}(a_i \# e_i, b_i \# f_i)_{i \in E}; \text{Restr}(a_i)_{i \in E}; \llbracket p \rrbracket; \text{Enter})$
$\llbracket \text{start } p \rrbracket$	$= \llbracket p \rrbracket; \text{Par}$

Table 4.2. Compilation of behavior expressions

A *thread* (or process, indifferently) is either the *dead* thread $\mathbf{0}$ or a tuple (S, E, C, L, P) . S, E, C have been described earlier; L is the *Channels* register (or *Lines*); P is the *Preemption* register (both to be described).

The machine is parameterized by a number n of processors. A *state* is a triple W, Z, A in which:

- W is a tuple of n *processors* (t, R) , each constituted of an *active thread* t and a queue R of *ready threads*;
- Z is a shared set of *channels*, each referencing a pair (I, O) of queues of *suspended threads*.
- A is a shared set of *preemption tokens* (to be described).

The rules in Table 4.3 define the transition relation \Rightarrow on global states from a transition function \rightarrow linking local states. The transition $\{ \dots, (t, R), \dots \}, Z, A \Rightarrow \{ \dots, (t', R'), \dots \}, Z', A'$ in the context rule means that processor (t, R) in register W is replaced by (t', R') , while the shared Z and A registers are replaced by Z' and A' . The local transition function \rightarrow is deterministic; a local state (t, R, Z, A) has only one possible successor. However, the whole machine appears non-deterministic since local transitions may be interleaved in an arbitrary way.

The specification uses queues and stacks: $-$ is the empty stack; $e::S$ matches any stack with e on top and S as tail; nil is the empty queue; $@$ is the “enqueue” constructor ($Q@t$ matches any queue ending with t); $Q+t$ matches any queue holding item t (Q is that queue in which t is removed).

The transitions for reduction are omitted in Table 4.3. None of them updates registers L, P, Z or A . The $\text{Ldcl}(c)$ instruction, for instance, makes a closure from code c and the current environment, and pushes it on the stack; it realizes the transition:

$$(S, E, \text{Ldcl}(c); C, L, P) \rightarrow ((c, E)::S, E, C, L, P)$$

- $(\alpha, \pi_1) \mathcal{M}(\beta, \pi_2)$ is true if communication is possible between the thread located at π_1 and interacting on α , and that located at π_2 and offering a complementary interaction on β . We have $(\alpha, \pi_1) \mathcal{M}(\beta, \pi_2)$ if for some π , π^1 , π^2 , $\1 and $\2 , we have $\pi_1 = \pi^1 \$^1 \pi$ and $\pi_2 = \pi^2 \$^2 \pi$, with:
 - $\{\$^1, \$^2\} \in \{\{\wedge_r, \wedge_l\}, \{<_r, <_l\}\}$
 - Both π^1 / α and π^2 / β are defined, and $\pi^1 / \alpha = \pi^2 / \beta$, where π / μ returns the port obtained after application of the relabellings and aliasings found in path π to port μ ($\{a\} / a\#v$ and $\{a/b\} / a\#v$ are undefined).

We will refer to condition (i) as *synchronizable*, and to condition (ii) as *name-matching*. (i) translates the fact that π_1 and π_2 are paths of threads resulting from a parallel or interrupt composition.

LCS implementations are based on this representation. The expressions are compiled into abstract code, paths are replaced by informations recording only the necessary structural information for computing predicates \mathcal{P} and \mathcal{M} , and locality and determinism of reactions are enforced: Progress of a system of threads results from deterministic progress of its individual threads, rather than from subsets of them.

4.3 The LCS abstract machine

Compiling LCS expressions

Core-LCS expressions are reduced using a customized and optimized SECD abstract machine (the SECD machine is described in, e.g. [5] or [17]). That machine implements a superset of the SECD instruction set, and uses five registers: A stack S combining the S and D stacks of Landin's machine, an environment E , the code register C , a resumption stack X for exception handling and a store identifier G . Both registers X and G will be omitted from the presentation here; Expansion and Interactions require two additional registers: L and P , soon to be described.

Code generation for functional expressions is mostly application of known principles, so we will not discuss it here. Behavior expressions are compiled as shown in Table 4.2. `Ldc1` is the SECD "load closure" instruction; `Enter`, `Par`, `Amb`, etc. are new instructions. Note that behavior expressions are compiled as closures; a behavior occurring in operand or message position yields a value (like a function value), but does not create a process. Processes are created from behavior closures by instruction `Enter`, as will be seen. A subsequent optimization pass replaces code fragments of shape `(Ldc1 (c) ; Enter)` by the simpler code `(c)`; this saves many behavior closure constructions and resumptions.

Architecture, States

The LCS abstract machine is basically a set of such SECD-based machines, reducing and expanding threads asynchronously, and cooperating for interactions. The structural information needed to compute the predicates \mathcal{P} and \mathcal{M} discussed above is kept into two new registers: L and P (local to the threads). The machine is shown in Table 4.3.

Generic	
$\Rightarrow \{P \text{ at } \varepsilon\}$	(initial)
$T \rightarrow_{\Pi} T'$	
<hr/>	
$S \cup T \Rightarrow \{(p \text{ at } \pi) \in ((S-T) \cup T') \mid \forall \pi' \in \Pi. \pi \mathcal{P} \pi'\}$	(context)
Reactions (e, e', etc are expressions; v, v', etc are values)	
$\{e \text{ at } \pi\} \rightarrow \{e' \text{ at } \pi\}$	$((e, e') \in \mathcal{R}, \text{ reductions})$
$\{\text{stop} \text{ at } \pi\} \rightarrow \emptyset$	
$\{(e_1 \wedge e_2) \text{ at } \pi\} \rightarrow \{e_1 \text{ at } (\wedge_l \pi), e_2 \text{ at } (\wedge_r \pi)\}$	
$\{(e_1 \vee e_2) \text{ at } \pi\} \rightarrow \{e_1 \text{ at } (\vee_l \pi), e_2 \text{ at } (\vee_r \pi)\}$	
$\{(e_1 < e_2) \text{ at } \pi\} \rightarrow \{e_1 \text{ at } (<_l \pi), e_2 \text{ at } (<_r \pi)\}$	
$\{e \{ / a \} \text{ at } \pi\} \rightarrow \{e \text{ at } (\{ / a \} \pi)\}$	
$\{e \{ \phi \} \text{ at } \pi\} \rightarrow \{e \text{ at } (\phi \pi)\}$	
$\{\text{do } v \Rightarrow e \text{ at } \pi\} \rightarrow_{\{\pi\}} \{e \text{ at } \pi\}$	
$(a \# v, \pi_1) \mathcal{M} (b \# v', \pi_2)$	
<hr/>	
$\{a \# v? x \Rightarrow e_1 \text{ at } \pi_1, b \# v'! v' \Rightarrow e_2 \text{ at } \pi_2\} \rightarrow_{\{\pi_1, \pi_2\}} \{\{v'/x\} e_1 \text{ at } \pi_1, e_2 \text{ at } \pi_2\}$	

Table 4.1. A reduction system for core-LCS

- Finally, *interactions* interpret those behavior constructions that may have a preemptive effect on other threads in the system. Interactions are labelled by the paths of the threads involved in the reaction.

The context rule expresses how progress of a set of threads results from progress of its constituents. After a reaction, the whole system of threads must be pruned from the threads preempted (if any) by those having performed the reaction; this pruning step is embedded into the context rule.

The predicates \mathcal{P} and \mathcal{M} would be easily defined by auxiliary inference systems, but we prefer to give more intuitive definitions:

- $\pi_1 \mathcal{P} \pi_2$ is true if a thread located at path π_1 resists (is not preempted by) an interaction of a thread located at path π_2 , or, in other words, if these paths are attached to threads part of groups resulting from a parallel or interrupt composition (with π_1 assigned to a thread of the right side t of $<$, in the latter case).

Using our canonical representation of paths, we have that $\pi_1 \mathcal{P} \pi_2$ iff, for some π , π^1 , π^2 , $\1 and $\2 , we have $\pi_1 = \pi^1 \$^1 \pi$ and $\pi_2 = \pi^2 \$^2 \pi$, with

$$(\$^1, \$^2) \in \{(\wedge_l, \wedge_l), (\wedge_l, \wedge_r), (<_r, <_l)\}.$$

the **signal** and **catch** expressions which are derived from the available constructs and an additional combinator $<$ (spelled **interrupt**).

Combinator $<$ acts as a parallel composition having in addition an asymmetric preemptive effect: an interaction performed by a process on its right hand side aborts all processes on its left hand side. Event catching is translated below as a communication between the processes signalling the event and its handler, composed with combinator $<$. The aliasing of ports implies the scope rules retained for events: only the innermost handler for an event may catch it. $\llbracket _ \rrbracket$ is the translation function; variables x_i are assumed not to occur free in expressions e or h_j , and labels χ and ζ are assumed not to occur in any user written expression

$$\begin{aligned} \llbracket \mathbf{signal} \ e \ \mathbf{with} \ m \rrbracket &= \chi\# \llbracket e \rrbracket ! \llbracket m \rrbracket \Rightarrow \mathbf{stop} \\ \llbracket e \ \mathbf{catch} \ e_1 \ \mathbf{with} \ h_1 \ || \ \dots \ || \ e_n \ \mathbf{with} \ h_n \rrbracket &= \\ &((\ \mathbf{fn} \ x_1 \Rightarrow \dots \ \mathbf{fn} \ x_n \Rightarrow \\ &\quad \llbracket e \rrbracket \ \{ \zeta\#x_1/\chi\#x_1, \dots, \zeta\#x_n/\chi\#x_n \} \\ &\quad < \ (\zeta\#x_1?m_1 \Rightarrow \llbracket h_1 \rrbracket \ m_1 \ \backslash / \dots \ \backslash / \ \zeta\#x_n?m_n \Rightarrow \llbracket h_n \rrbracket \ m_n) \\ &\quad) \ \llbracket e_1 \rrbracket \ \dots \ \llbracket e_n \rrbracket) \ \{ / \zeta \} \end{aligned}$$

4.2 Operational semantics

A system of core-LCS processes may be seen as a tree with nodes labelled by combinators $/\backslash$, $\backslash/$, $<$, $\{a\}$, $\{\phi\}$ (ϕ is a relabelling or aliasing), and the leaves constituting the threads. That tree evolves as the result of progress of its leaves and their interactions; this is essentially the presentation of [2] and the classical “observational” presentation. For implementation purposes, it would be better to handle a system of threads as a set or multiset, as in the Chemical Abstract Machine [4]. A simple method to achieve this is to distribute the structure information represented by the nodes of the previous tree into the leaves themselves. This can be done simply by associating each thread with the path from the root of the previous tree to that thread. Clearly, both representation carry exactly the same information. An operational semantics for core-LCS using this presentation is shown in Table 4.1.

Table 4.1 defines an evaluation relation \Rightarrow for a set of “located” threads from a reaction relation \rightarrow_{Π} operating on its subsets (labels Π are sets of paths; \rightarrow stands for \rightarrow_{\emptyset}). Paths are built from the empty path ε and constructors for compositions, restriction and relabelling. The rules make use of two auxiliary predicates \mathcal{P} (read *Persistent*) and \mathcal{M} (read *Matches*), soon to be described. There are three kinds of reactions:

- *Reductions* (figured by a single rule; the exact details are left out) implement the usual applicative order evaluation for functional expressions, extended to handle behavior expressions. Behavior expressions reduce to behavior values, similar to function values.
- *Expansions* (all other reactions except the last two) set up a structured system of threads by interpreting the composition, restriction and renaming behavior constructions.

Termination and collection of results

Computation is over when no pixel processor computed a different region number than currently assigned to it. This information is propagated using the `jinp/jout` synchronization chain. The synchronizer initially sends the value `false` on `jinp`; each processor propagates the value received, or `true` if its region changed. The boolean received on port `jout` by the synchronizer is thus `true` iff any of the pixel processors updated its region number. Collecting the results consists of scanning all pixel processors and building a list from the regions values they computed. That list is then delivered to the client process (on port `out`), formatted to its needs (by function `format`, left undefined). Note that all pixel processors have stopped by then.

```
agent collect_results = rinp![]=> rout?l=> out!(format l)=> stop
```

The network of pixel processors

The main process `compute_regions` is a pipeline of located pixel processors implementing pixels $(0,0)$ to $(k-1,k-1)$, in parallel with the synchronizer. Functions `combine` and `range` are library functions; `combine` makes a list of pairs from a pair of lists, and `range 0 n` returns the list $[0,1,\dots,n]$:

```
agent compute_regions pict =  
  let val k = length pict  
    val ind = range 0 (k-1)  
    agent pixline (i,line) =  
      PIPE (fn (j,v)=> pixel(v,k*i+j) at (i,j,k)) (combine(ind,line))  
  in (synch /\ PIPE pixline (combine (ind,pict)))  
    {/global finp fout jinp jout rinp rout}  
end;
```

4. Operational Semantics, Implementations

The operational observational semantics of LCS is expressed abstractly in [2] from a hierarchy of three relations named *Reduction*, *Expansion* and *Communication*. We give here a less abstract description of the operational semantics of LCS, suitable as an implementation basis, in terms of a compiler and an abstract machine.

For simplicity, we focus on the concurrency and communication aspects; the implementation of the functional subset of the language is just overviewed, and we will omit to discuss the imperative aspects (stores, stream input/output and timers).

4.1 Core-LCS

The operational semantics of LCS is obtained from the semantics of a more abstract but richer “core” language including lambda-expressions, the usual primitives, and behavior expressions. Core behavior expressions include those in Table 2.1, except

Cooperation of the processors

An image is represented by a matrix of light intensities. That matrix is distributed on a matrix of `pixel` processes. For simplicity, we assume that pictures are toroidal: The right and left pixels in each line are assumed neighbors, as well as the top and bottom pixels in each column. This way, all pixel processes have four neighbors.

Parameterized communication ports are used to implement the connections between the north to west ports of a process and the `me` ports of its four neighbors. The `me` port of the processor handling pixel (i, j) is mapped to the communication port `global#(i, j)`; we can deduce from this the `global` ports mapped to its north to west ports. The following `at` combinator renames ports `me`, `north`, ..., `west` of its left parameter (a pixel processor) into `global` ports, according to the pixel index (i, j) of the processor and the size `k` (in pixels) of the picture.

```
infix at;  
agent b at (i, j, k) =  
  let fun ++ i = (i+1) mod k  
    and -- i = (i+(k-1)) mod k  
  in b {global#(i, j)/me,  
        global#(--i, j)/north, global#(i, ++j)/east,  
        global#(++i, j)/south, global#(i, --j)/west}  
end;
```

Synchronization

We will use barrier synchronizations to force the pixel processors to operate synchronously on elements of the distributed region matrix. The pixel processors are pipelined through the ports `finp` and `fout` on one hand and through `jinp` and `jout` on another hand (and through `rinp` and `rout` as well, for production of results). To build the pipeline of processors, we define the process combinators `^^` (read pipe) and `PIPE` (iterated `^^` on a list of parameterized processes) as follows:

```
infix ^^;  
agent a ^^ b =  
  (a {rtmp/rout, ftmp/fout, jtmp/jout} /\  
   b {rtmp/rinp, ftmp/finp, jtmp/jinp}) {/rtmp, /ftmp, /jtmp};  
agent PIPE a [h] = a h | PIPE a (h::t) = a h ^^ PIPE a t;
```

Synchronizations are initiated by an external `synch` process shown below. At each iteration, `synch` first offers an output on port `finp`. When that synchronization signal reaches back the synchronizer (through `fout`), it starts another synchronization loop on port `jinp` (ending up at `jout`). That second loop of synchronizations is also used to detect termination. Depending on the boolean received on `jout`, either a new iteration is started, or results are collected.

```
agent synch =  
  finp! => fout? => jinp! false => jout? b =>  
  if b then synch else collect_results
```

3. Programming with LCS, A complete example

Region Labelling

LCS programming is illustrated in this section by a simple image processing example known as region labeling. The description of the problem is adapted from [6]. An image is a matrix (assumed square here) of pixels, each being assigned a light intensity. A region is a set of neighboring pixels having the same light intensity. The region labelling problem is that of identifying the regions of a picture and assigning a region label to each of its pixels.

There is a natural iterative algorithm to solve the problem. Each pixel initially receives a unique region number (e.g. in a picture of $k \times k$ pixels, pixel (i,j) , for i, j in $0..k-1$, is assigned $k*i+j$). At each iteration, the region number of a pixel is set to the largest among those assigned to its neighbors (including itself) having the same light intensity than itself. Processing is complete when region numbers are stable.

For the purpose of exercising LCS constructions, we propose a solution assigning one process per pixel. This is not a “production” solution however: Though we could run it satisfactorily on images of size up to 100×100 pixels on a typical workstation, that solution is quite space consuming. A “production” version would preferably implement a hierarchical version of the above algorithm, as described in [6], in which a few sequential cooperating processes compute regions on different parts of the picture.

Pixel processors

The behavior of an individual pixel processor is described below; it maintains a local state through its parameters constituted of a pair (light intensity x region number). It is built as the disjunction (\setminus) of two processes: one for collecting the final results, activated on reception of some list of regions on port `rinp` (collection of results will be discussed later), and another implementing labelling.

After synchronizations on ports `finp` and `fout`, the pixel processor forks a sub-process that sends its light intensity and region number to its four neighbors on a port labelled `me`. In parallel, it queries its four neighbors through ports `north` to `west`; the new region of the pixel is computed from those received by function `maxr` (definition omitted). That new value becomes effective once synchronizations on `jinp` and `jout` have been accepted (the messages passed on `jinp` and `jout` concern termination detection).

```
agent pixel (lr as (light,region)) =
  rinp? l => rout! (region::l) => stop
  \ / finp? => fout! =>
    (me!lr=> me!lr=> me!lr=> me!lr=> stop /\
     north?nn=> east?ee=> south?ss=> west?ww=>
      let val region' = maxr lr [nn,ee,ss,ww]
      in jinp?b=> jout!(b otherwise (region<>region'))=>
        pixel (light,region') end);
```

2.3 Behavior compositions

There are three basic forms of compositions: parallel ($/\backslash$), choice ($\backslash/$) and **catch**.

Processes composed in parallel may communicate values. The single communication and synchronization primitive is synchronous rendez-vous on communication ports. As in CCS, the first process in a choice composition that performs an action makes the alternative process(es) in the composition terminate.

Events, together with the event raising (**signal**) and event trapping (**catch**) expressions, implement an exception mechanism at process level. Events implement a particular kind of communication ports, obeying their own scope rules; they are built from event constructors, they may have parameters, and carry messages. Raising an event by **signal** offers the message associated with the event on a communication port identified by the event itself. Handling an event by **catch** is a communication between the process that offers the event and its handler. In addition to passing a message, this communication has the effect of terminating all processes on the left side of **catch**.

The communication ports defined by events have particular dynamic scope rules. An event can only be caught by a handler if issued from the behavior on the left side of the **catch**, and its scope stops at the innermost handler found for it. In short, events propagate like exceptions in SML, except that signalling an event has no effect when no surrounding handler may catch it.

Beside an exception mechanism, events are often used to force termination of a system of processes once a significant result has been produced. Such a usage is illustrated by the combinator `watch` below; `watch a b` behaves like `a` until some value is output on port `out`, and then behaves like `b`.

```
fun watch a b =  
  let event DONE in (a /\ out?x=> signal DONE) catch DONE=> b end
```

2.4 Other issues

Each process is associated with a store. That store may be private, or shared with some other processes, depending upon the process compositions used (e.g. process composed by $/\backslash$ share their store; composition \backslash/\backslash assigns a private copy of the current store to its left process). Other imperative features supported by LCS include timers, stream input and output, and windows.

In addition to the commands provided by an ML user interface, LCS interfaces offer commands for process management. An important one is the **start** command, that creates a process from a behavior value. Other commands are provided to force termination of all user processes, and to build stand-alone applications.

Behaviors, like other values, are submitted to a polymorphic type discipline extending that of ML. Behaviors are assigned types which make precise, for each possible communication label, the type of the values that may be passed on ports referring to that label. Details about the type system of LCS and its implementation can be found in [2, 3].

constructs include all those of CCS. The construct “=>” prepends an action (or interaction capability) to a behavior. Actions may be either that of performing a preemptive move (do, similar to the τ action of CCS), that of proposing a value on a communication port (output or signal), or that of accepting a value on a port (input or event handling). LCS is higher order in that one can pass behavior values as messages, or declare functions operating on behaviors.

Consistency of communication is enforced by an extension of the ML polymorphic typing discipline. Evaluation of an expression typing as a behavior does not produce a process but rather a closure, similar to a function value. Creation of a process may only result from a process creation command applied to a behavior value at top-level or, recursively, from an already created process performing a composition.

2.2 Naming schemes

In contrast with the approach retained in CML, Poly/ML and FACILE, as well as PICT, that make use of first-class channels created by a generative primitive, LCS follows closely the CCS approach of naming. The communication links of a process are of a structural nature; a process has communication labels a, b, c, etc. like a record has fields labelled a, b, c, etc. Label passing capabilities are approximated by using parametric labels (their theory is discussed in [1]).

LCS communication ports have two components: a label and an extension, e.g. the port $p\#(425, \text{true})$ has label p and extension $(425, \text{true})$. All ports have an extension (the default extension is the value $()$, of type `unit`). Ports have global scope, unless otherwise specified. Three constructions control the scope of ports:

- The *restriction* of p in b (written $b\{p\}$) has the effect of delimiting the scope of ports labelled p occurring within b to process b .
- The *relabelling* ($b\{p/q\}$) restricts label p in b and makes ports labelled q within b appear to the context as having label p . The restrictive effects of LCS relabellings make then slightly different from CCS relabellings. Like restrictions, relabelling apply to all ports with the labels involved.
- Finally, *aliasing* allows to rename individual ports, rather than sets of ports identified by their labels. E.g. $(b\{p\#1/q\#(2,5), p\#2/r\#\text{true}\})$ is an aliasing, renaming simultaneously port $q\#(2,5)$ into $p\#1$, and port $r\#\text{true}$ into $p\#2$; simultaneously, all ports labelled p are restricted in b .

Relabelling and aliasing are useful to connect a process defined in isolation with a system of processes obeying a different naming scheme. A typical example of use of restriction and relabelling is the pipe combinator, defined below (\wedge). Other examples will be given in Section 3.

```
fun a  $\wedge$  b = (a {tmp/out} /\ b {tmp/inp}) {/tmp};
```

Combinator \wedge connects the `out` ports of its first parameter (if it has any) with the `inp` ports of its second parameter. The connection is made using an auxiliary label `tmp`.

Section 2 describes the features of the language LCS and discusses its language design decisions. Programming in LCS is illustrated by a complete example in Section 3. Section 4 discusses compilation and abstract machines for sequential and parallel implementations of the language. It is assumed throughout the paper that the reader has some knowledge of the essential features of both the language ML and the CCS formalism.

2. The design of LCS

2.1 Structure, behavior expressions

An important decision when designing a concurrent language is that of the respective role of functions and processes. LCS stands between the group of languages Poly/ML [12], CML [18], FACILE [7], in which concurrency and communication primitives are provided as functions, and PICT [14], that exemplifies the opposite “functions as processes” approach. LCS is basically a process language: The highest units of computation are the processes, but abstraction and application are primitives and may be used to build process scripts. Practically, LCS adds to Standard ML a language of behavior expressions, and commands to create processes from behavior values. Table 2.1 shows the syntax of LCS expressions.

exp ::= <sml expression>	SML expressions
stop	null behavior
{ do e } => exp	commitment
port ? {pat} => exp	input
port ! {exp} => exp	output
signal exp { with exp }	event raising { with message }
exp (\wedge \wedge \wedge \wedge) exp	parallel compositions
exp (\vee \vee \vee \vee) exp	choice compositions
exp catch erules	event handling
exp "{ /lab ₁ , ... ,/lab _n }"	restriction
exp "{ 'lab ₁ /lab ₁ , ... ,lab _n '/lab _n }"	relabelling
exp "{ 'port ₁ /port ₁ , ... , port _n '/port _n }"	aliasing
erules ::= erule { erules }	event handlers
erule ::= exp => exp'	(eqv. exp with _ => exp')
exp with match	event handler rules
port ::= label { # exp }	communication ports
match ::= patt => exp { match }	SML matches
patt ::= <sml pattern>	SML patterns

Table 2.1. LCS expressions

Behaviors are built from primitive behavior expressions and functional constructions (recursion, conditional, abstraction, application, etc.). The primitive behavior

Process Calculi at work - An account of the LCS project

Bernard Berthomieu

LAAS/ CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex, France.
e-mail: Bernard.Berthomieu@laas.fr

Abstract. LCS is an experimental high level parallel programming language aimed at exploring design, implementation and use of programming languages based upon process calculi. LCS extends Standard ML with primitives for concurrency and communication based upon a higher order extension of the CCS formalism. The paper discusses language design, illustrates programming disciplines, and investigates abstract machines for sequential and parallel implementations of the language.

1. Introduction

Over the last decade, there has been considerable research interest in process calculi (CCS [15], CSP, the pi-calculus [16], and a number of related formalisms). Process calculi bring for concurrent programming what the functional paradigm brought to sequential programming, and above all the capability of building complex concurrent systems from meaningful smaller pieces. The central role in process calculi is played by communications: a process describes interaction capabilities with other processes. The semantics of these formalisms are given in terms of an observation relation characterizing their interaction capabilities. The approach has been undoubtedly fruitful in the areas of specification and analysis of concurrent systems; it is inherently parallel, nondeterministic, and nonterminating systems are naturally accommodated.

Concurrently, a number of work were aimed at designing and implementing programming notations based on these paradigms, hopefully benefiting of their theoretical advances in terms of formal programming techniques. LCS is one of these programming languages designed to experiment with process calculi for concurrent programming; it implements a higher order extension of Robin Milner's Calculus of Communicating Systems (CCS) with process passing and parametric channels, embedded in Standard ML.

Several implementations have been investigated. A sequentially running version has been used several years as a workbench for language design, compilation techniques and experiments. Implementations on concurrent targets (networks of machines and parallel machines) are in progress. An architecture of virtual machine suitable for distributed targets has been designed, and several of its important components have been developed, including memory management with distributed garbage collection [9, 13] and distributed scheduling [10].