Modelling Garbage Collection Algorithms

Howard Bowman, John Derrick & Richard Jones.
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Phone: + 44 227 764000, Email: hb5@ukc.ac.uk, jd1@ukc.ac.uk. and rej@ukc.ac.uk)

February 8, 1995

Abstract

We show how abstract requirements of garbage collection can be captured using temporal logic. The temporal logic specification can then be used as a basis for process algebra specifications which can involve varying amounts of parallelism. We present two simple CCS specifications as an example, followed by a more complex specification of the cyclic reference counting algorithm. The verification of such algorithms is then briefly discussed.

Keywords: Concurrency, garbage collection, temporal logic, CCS.

1 Introduction

The memory management of simple *static* programming languages, such as Fortran, can be handled entirely by the compiler. The location of all variables can be fully determined at compile-time and no run-time support for memory management is necessary. However, such languages impose considerable restrictions on programming style, for example, recursive procedure calls are disallowed. High-level languages that allow recursion, on the other hand, demand some run-time support. Typically, the compiler determines statically the memory requirements of each procedure. When a procedure is invoked at run-time, a *frame* of sufficient size is allocated to the procedure on a *stack*. When this invocation of the procedure completes, the frame is popped off the stack. No action is required of the programmer.

Such a stack-based discipline is insufficient for many modern high-level programming languages. These languages require the use of a *heap*. Such languages may simply offer the programmer dynamic data structures, which are stored on the heap as their size cannot be determined at compile-time. Furthermore their lifetime may exceed the lifetime of the procedure which created them (e.g. Pascal [Wirth and Jensen, 1974]). Alternatively the language may support *closures*, functions paired with environments of values (e.g. [Turner, 1985]). If the binding of names to values in these environments is static rather than dynamic, then these closures must be kept on the heap until they are no longer in use.

In general, the extent of such heap-allocated structures cannot be determined by inspection of the program source. Either the programmer must explicitly allocate and de-allocate such objects or an automatic storage reclamation system, a *garbage collector*, must be employed to identify at run-time which objects may be in use now and in the future, and which objects cannot be used again. The space used by the latter can be re-cycled. The advantage claimed for the manual method is efficiency: the programmer 'knows' when objects are no longer in use. However this claim is increasingly being challenged (see for example the recent discussion in the Internet news group *comp.lang.c++* on the performance of Sun's memory allocator and de-allocator, malloc() and free(), compared with the Boëhm-Weiser garbage-collected GC_malloc() [Boehm and Weiser, 1988]).

Furthermore, there is considerable evidence that a high proportion of programmer time is spent chasing memory allocation bugs [Rovner, 1985]. Automatic storage reclamation offers the programmer the clarity of a higher level interface to the memory sub-system with guarantees that objects will never be prematurely deallocated (the 'dangling pointer' bug) nor that the same memory will be (incorrectly) allocated to more than one object. To maintain these guarantees, it is essential that garbage collection algorithms are demonstrably correct.

Proving the correctness of all but the simplest sequential algorithms for garbage collection is difficult. To take but one example, the authors know of no less than thirteen papers offering different strategies to prove the correctness of well-known pointer-reversal algorithms [Schorr and Waite, 1967]. The implementation of concurrent algorithms only increases this difficulty. David Gries [Gries, 1977] noted the fragility of concurrent algorithms in his paper proving the correctness of Dijkstra *et al*'s 'on-the-fly' algorithm [Dijkstra *et al.*, 1976]:

> 'When we write a procedure to be used in a sequential setting, once it is written and proved correct we can view it as a black-box operation and use it over and over again without having to look in the black box. We worry only about what it does. In a parallel setting, however, we must analyse the procedure each time we wish to use it to make sure that the parallelism does not disturb its proof of correctness. And each change in the other process forces us to re-analyse the procedure again. One can avoid this complexity by making the procedure an indivisible operation through the use of synchronisation and mutual exclusion primitives and by limiting the use of shared variables. Or one can summarise in an invariant for the procedure what a parallel process must leave true in order not to interfere.
>
> The on-the-fly garbage collector is very fragile and susceptible to such changes. Slight changes which would seem innocent in a sequential setting are disastrous in a parallel context.'

It is surely worth noting that many attempts to prove the correctness of the 'on-the-fly' algorithm, including Dijkstra's, were found to be lacking.

The aim of our research is to model and verify concurrent garbage collection algorithms using formal models of concurrency. In particular, we wish to turn our attention to the Cyclic Weighted Reference Counting Algorithm [Jones and Lins, 1993], a hybrid reference-counting/mark-scan garbage collection algorithm for distributed processors. The formal models used are Temporal Logic,[Manna and Pnueli, 1992], and the process algebra CCS, [Milner, 1989]. Both temporal logic and CCS have be en used successfully to model and verify a number of algorithms, [Sanderson, 1982; Larsen and Milner, 1992; Parrow, 1985; Parrow, 1987] and [Richier *et al.*, 1987; Cavelli and Horn, 1987; Clarke *et al.*, 1986]. We show here how garbage collection algorithms can be specified using temporal logic and process algebras. The verification of such algorithms is an area of ongoing research.

This paper starts by examining those simpler algorithms upon which the CWRC has been built, namely standard recursive marking [McCarthy, 1960] and Cyclic Reference Counting [Martinez *et al.*, 1990; Lins, 1990]. In Section 2 we consider garbage collection from an abstract point of view. These abstract requirements are formally described in temporal logic. Two specifications in CCS are then given, each involving different amounts of parallelism. In Section 2 we present the Cyclic Reference Counting Algorithm, and show how this can be specified in a process algebra. The verification of such algorithms is then discussed.

2 An abstract model of garbage collection

One abstract view of garbage collection is the following. The heap consists of a number of cells, one is named the **root** of the heap. When pointers between the cells are represented as arcs, the active data structure is a connected, directed graph. The cells which are in the transitive closure of the root (denoted $root^*$) are considered to be the active cells. Cells which are not active are considered garbage, and the purpose of garbage collection is to identify these cells and make them available for future use.

The garbage collector does so by maintaining a set of free cells (often called the free-list, although it need not be implemented as a list). Let us view the free-list as the transitive closure of a named cell **free**. Garbage collection is then a state change of the heap, which will alter the transitive closure of **free**. We denote this state change by the use of primes to represent the effect of garbage collection having taken place on the heap. This preliminary specification assumes that no mutator action happens concurrently.

Garbage collection should preserve the active cells:

$$root'^* = root^*$$

A cell which is neither active, or already on the free-list, should be placed on the free-list by the garbage collector:

$$\forall x \notin root^* \cup free^* : x \notin root'^* \wedge x \in free'^*$$

The free-list should be preserved or increased by garbage collection:

$$free^* \subseteq free'^*$$

Finally active cells should never be reclaimed as garbage, this amounts to saying that at all time the transitive closures of **root** and **free** are distinct:

$$root^* \cap free^* = \emptyset$$

These four properties make an abstract specification of garbage collection, against which sequential or concurrent implementations must be verified. We now formalize these requirements by giving a temporal logic specification.

2.1 Temporal Logic Specification

We will use Temporal Logic to express the correctness properties of garbage collection algorithms. This form of logic has been shown to be highly suited to abstract expression of program properties and in particular, expression of the properties of concurrent computations [Manna and Pnueli, 1992]. As first observed by Pnueli [Pnueli, 1977], both safety and liveness properties can be expressed with Temporal Logic, while only safety properties can be expressed with first order logic. We will use standard Manna and Pnueli Linear Time Temporal Logic [Manna and Pnueli, 1992] in this paper.

The following Temporal Logic propositions express, in the same order, the properties of garbage collection highlighted in the previous section.

$$root^* = S \rightarrow \Box(root^* = S)$$

i.e. if the transitive closure of the root equals S at the start of the garbage collection, then in *all future states* it will equal S.

$$\forall x \notin (root^* \cup free^*).\ \Diamond(x \notin root^* \wedge x \in free^*)$$

i.e. all cells neither in the active or free lists will *eventually* be placed in the free list. The re-expression of the other two conditions included in the above section is also straightforward. They can be expressed as follows:-

$$free^* = S \rightarrow \Box(free^* \supseteq S)$$

$$\Box(root^* \cap free^* = \emptyset)$$

2.2 Using CCS to model simple garbage collection algorithms

In this section we specify two simple garbage collection algorithms in CCS. One is a sequential version, the other uses the parallelism of garbage collection as much as possible.

An invaluable introduction to CCS is given in [Milner, 1989]. CCS is a process algebra, and all objects within our system specification (free-lists, cells, garbage collectors) will be modelled as processes.

To model cells and pointers between them we consider the heap to consist of a fixed number of cells (or nodes) $P_0, \ldots, P_N$. A cell can then be referred to by its subscript without confusion. We assume that the dedicated root cell is P_0. To represent a cell as a process, we consider each process to have $N+1$ integer parameters which represent pointers to other cells. So in the process

$$P_i(x_0, \ldots, x_N)$$

x_0 is the number of pointers from P_i to P_0, x_1 is the number of pointers from P_i to P_1 etc.

We use the standard method to access and update a data-store in CCS by each parameter having two ports, in^i and $\overline{out^i}$. Each process P_i is defined by:

$$\begin{array}{rcl} P_i(x_0,\ldots,x_N) & = & in^i_0(x).P_i(x,x_1,\ldots,x_N) + \overline{out^i_0}(x_0).P_i(x_0,x_1,\ldots,x_N) \\ & + & in^i_1(x).P_i(x_0,x,x_2,\ldots,x_N) + \overline{out^i_1}(x_1).P_i(x_0,x_1,\ldots,x_N) \\ & + & \\ & \vdots & \\ & + & in^i_N(x).P_i(x_0,\ldots,x_{N-1},x) + \overline{out^i_N}(x_N).P_i(x_0,\ldots,x_N) \end{array}$$

We record the collection of active cells and free cells by maintaining two sets. The free-list (in fact it is a set in this instance) is then defined by the process:

$$Free(\emptyset) = add_F(x).Free(\{x\}) + empty_F.Free(\emptyset)$$

$$Free(X) = add_F(x).Free(X \cup \{x\}) + minus_F(x).Free(X \setminus \{x\}) \qquad X \neq \emptyset$$

The collection of active cells is modelled by the process *Active*, defined by:

$$Active = add_A(x).Set(\{x\})$$

$$Set(\emptyset) = empty_A.Active$$

$$Set(X) = add_A(x).Set(X \cup \{x\}) + \overline{out_A}(maxX).Set(X \setminus \{maxX\}) \qquad X \neq \emptyset$$

where *max* returns the maximum element of a finite set of integers.

The basis method of a simple garbage collection is to start with an empty active collection, run a recursively defined algorithm which checks which cells are still active, then finally defines the free-list to be all cells which are not active. The initial prefix of an action g allows the garbage collector to be fired into action.

$$GC = g.(Active := \{0\}).gc(0).(Free := \{0, \ldots, N\} \setminus Active).GC$$

Written formally this becomes:

$$GC = g.\overline{add_A}0.gc(0).\overline{add_F}0. \ldots .\overline{add_F}N.Y.GC$$

$$Y = out_A(j).\overline{minus_F}(j).Y + \overline{empty_A}.0$$

The definition of the recursively defined algorithm is as follows:

$$gc(i) = \prod_{j=0}^{N}(out_j^i(x_j).\text{if } x_j > 0 \wedge j \notin Active \text{ then } \overline{add_A}j.gc(j))$$

This works as follows: for an active process P_i, *gc(i)* will, in parallel, check each parameter x_j. If this parameter is non-zero, then the cell P_j is connected to P_i and hence active. We add the reference j to the active set, and recursively look at the process P_j. To ensure termination we only perform this recursion when the process P_j has not already been discovered to be active, we do this by checking whether $j \notin Active$. The definition of this as a CCS expression is omitted.

The whole system (heap plus garbage collector running on it) is then represented by the expression

$$(Active|Free|P_0|\ldots|P_N|GC) \setminus \mathcal{L}$$

where $\mathcal{L}$ is the set of all sorts of the processes. Prefixing the garbage collector by the action g allows us to fire the process into action. Thereafter system will evolve silently, all the communications between GC and the cells being internal transitions, until termination.

This algorithm can obviously be written without so much parallelism. For example we can define a further garbage collector GC', by replacing the algorithm gc by:

$$\begin{aligned} gc'(i) \quad = \quad & out_1^i(x_1).(\text{if } x_1 > 0 \wedge 1 \notin Active \text{ then } \overline{add_A}1.gc'(1)). \\ & out_2^i(x_2).(\text{if } x_2 > 0 \wedge 2 \notin Active \text{ then } \overline{add_A}2.gc'(2)). \\ \ldots & \\ & out_N^i(x_N).(\text{if } x_N > 0 \wedge N \notin Active \text{ then } \overline{add_A}N.gc'(N)) \end{aligned}$$

A simple test for a formal specification technique is then to try and prove the equivalence of the two algorithms so given.

2.3 Verifying the Algorithms

Having modelled the requirements of garbage collection using temporal logic, and given simple specifications using CCS, we would like to verify the process algebra specifications against the temporal logic properties. One approach is to use model checking algorithms to perform this verification. This avenue is currently being explored by the authors.

Even without the temporal logic requirements, it is true that the functionality of the two CCS systems is equivalent in some sense. Proving equivalence of the two algorithms amounts to showing that for a given heap, the two algorithms will produce the same transitive closures of **root** and **free**. The complete system, i.e. heap and the garbage collector running on it, is modelled in the usual fashion by parallel composition of the processes P_i with the active set and the garbage collector GC itself. One method of showing equivalence is to show that the following two processes are bisimilar, [Milner, 1989]:-

$$(Active|P_0|\ldots|P_N|GC) \setminus \mathcal{L} \qquad (Active|P_0|\ldots|P_N|GC') \setminus \mathcal{L}$$

The restriction by the sorts $\mathcal{L}$ ensures that the only non-silent communication are those communications with the process *Free*. The nature of the processes GC and GC' mean that this amounts to showing the active-set will be the same in both systems. Unfortunately the equivalence technique of deriving bisimulations requires that the order of events is identical in the two systems under discussion. For comparisons between sequential and parallel algorithms, this is too strong a requirement, since by its nature the parallel algorithm does not preserve the order of the sequential version.

3 Cyclic Reference Counting

Lazy Cyclic Reference Counting combines reference counting with lazy four-colour mark-scan garbage collection. Cells are allocated and references are copied in a manner similar to the standard reference counting algorithm [Collins, 1960], as is the deletion of the last reference to an object. However, if the target of a deleted pointer is shared then it may be part of an isolated, and hence garbage, cycle. In the lazy algorithm [Lins, 1990], a reference to the cell is placed on a *control queue* but no further action is taken until either the free-list becomes empty or the control queue is full. In the original algorithm [Martinez *et al.*, 1990], the cell is examined immediately to determine whether it is garbage.

In either case, cells in the transitive closure of this cell are eventually marked and scanned to find any references from cells external to this subgraph. If none are found the subgraph is garbage and is returned to the free-list. Garbage collection proceeds in three phases. In the first phase all cells in the transitive closure are painted *red*. Each time a cell is visited its reference count (RC) is decremented. On completion only those cells that are the target of an external reference will have non-zero RCs. The task of the second phase is to discover any such cells. They and their descendants are re-painted *green* and their RCs are corrected. All other cells are painted *blue*. The third and last phase returns blue cells to the free-list.

We denote a pointer from a cell S to a cell T by $< S, T >$.

```
New(R) =
    U := pop(free-list)
    RC(U) := 1
    colour(U) := green
    make-pointer(<R,U>)

Copy(R,<S,T>) =
    RC(T) := RC(T) + 1
    make-pointer(<R,T>)

Delete(<R,S>) =
    delete-pointer(<R,S>)
    RC(S) := RC(S) - 1
    if RC(S) = 0 then
        for T in Sons(S) do
```

```
            Delete(<S,T>)
        colour(S) := none
        push(S,free-list)
    else
        Mark_red(S)
        Scan(S)
        Collect_blue(S)

Mark_red(S) =
    if colour(S) ≠ red then
        colour(S) := red
        for T in Sons(S) do
            RC(T) := RC(T) - 1
            Mark_red(T)

Scan(S) =
    if colour(S) = red then
        if RC(T) > 0 then
            Scan_green(S)
        else
            colour(S) := blue
            for T in Sons(S) do
                Scan(T)

Scan_green(S) =
    colour(S) := green
    for T in Sons(S) do
        RC(T) := RC(T) + 1
        if colour(T) ≠ green then
            Scan_green(T)

Collect_blue(S) =
    if colour(S) = blue then
        colour(S) := none
        for T in Sons(S) do
            Collect_blue(T)
            delete-pointer(<S,T>)
        push(S,free-list)
```

3.1 CCS Specification of CRC

In a manner similar to before we show how we can model CRC in CCS. With CRC each cell now contains two pieces of additional information, the first the reference count for a cell, the second the colour of the cell. The colour will be green, red or blue. We extend our definitions of processes representing cells by adding parameters for the reference count, and the colour, and two ports per parameter (one to update it, one to access the value).

$$\begin{array}{rcl}
P_i(n,c,x_0,\ldots,x_N) & = & inref_i(m).P_i(m,c,x_0,\ldots,x_N) + \overline{outref_i}(n).P_i(n,c,x_0,\ldots,x_N) \\
& + & incol_i(d).P_i(n,d,x_0,\ldots,x_N) + \overline{outcol_i}(c).P_i(n,c,x_0,\ldots,x_N) \\
& + & in^i_0(x).P_i(n,c,x,x_1,\ldots,x_N) + \overline{out^i_0}(x_0).P_i(n,c,x_0,x_1,\ldots,x_N) \\
& + & in^i_1(x).P_i(n,c,x_0,x,x_2,\ldots,x_N) + \overline{out^i_1}(x_1).P_i(n,c,x_0,x_1,\ldots,x_N) \\
& + &
\end{array}$$

$$\vdots$$
$$+ \quad in^i_N(x).P_i(n,c,x_0,\ldots,x_{N-1},x) + \overline{out^i_N}(x_N).P_i(n,c,x_0,\ldots,x_N)$$

Without loss of generality we refer to the process P_i by its subscript, and henceforth all processes will range over integer parameters. With these definitions the process representing the task New is:

$$New(R) = pop_F(v).\overline{inref_v}(1).\overline{incol_v}(green).out^R_v(x).\overline{in^R_v}(x+1).0$$

The communication $pop_F(v)$ will obtain a new cell from the free-list (which we call v). (We assume that the free-list is modelled as before, with ports $\overline{pop_F}$ and $push_F$ to perform the required communication.) $\overline{inref_v}(1)$ will set the reference count in v to 1, then $out^R_v(x).\overline{in^R_v}(x+1)$ will create an additional pointer from R to v.

To increase clarity we use the following abbreviation in the process definitions:

$$SONS^S_T(P) = \prod_{T=0}^{N} out^S_T(x).\{\text{ if } x \geq 1 \text{ then } P\}$$

The remaining processes are:

$$Copy(R, <S,T>) = outref_T(n).\overline{inref_T}(n+1).out^R_T(x).\overline{in^R_T}(x+1).0$$

$$\begin{aligned} Delete(<R,S>) \quad = \quad & out^R_S(k).\overline{in^R_S}(k-1).outref_S(x).\overline{inref_S}(x-1). \\ & \text{if } x = 1 \text{ then } \{SONS^S_T(Delete(<S,T>))\}.\overline{incol_S}(none).\overline{push_F}(S).0 \\ & \text{else } \{Mark\_red(S).Scan(S).Collect\_blue(S).0\} \end{aligned}$$

$$\begin{aligned} Mark\_red(S) \quad = \quad & outcol_S(c).\text{ if } c \neq red \text{ then} \\ & \{\overline{incol_S}(red).SONS^S_T(outref_T(y).\overline{inref_T}(y-1).Mark\_red(T)).0\} \end{aligned}$$

$$\begin{aligned} Scan(S) \quad = \quad & outcol_S(c).outref_S(n).\text{ if } c = red \text{ then} \\ & \{\text{ if } n > 0 \text{ then } Scan\_green(S).0 \text{ else } \{\overline{incol_S}(blue).SONS^S_T(Scan(T)).0\}\} \end{aligned}$$

$$\begin{aligned} Scan\_green(S) \quad = \quad & \overline{incol_S}(green). \\ & \{SONS^S_T(outref_T(n).\overline{inref_T}(n+1).outcol_T(c).\text{ if } c \neq green \text{ then } Scan\_green(T).0)\} \end{aligned}$$

$$\begin{aligned} Collect\_blue(S) \quad = \quad & outcol_S(c).\text{ if } c = blue \text{ then } \{\overline{incol_S}(none). \\ & \{SONS^S_T(Collect\_blue(T).\overline{in^S_T}(0)).\overline{push_F}(S).0\}\} \end{aligned}$$

3.2 Correctness of the algorithm

To show that this algorithm is correct we have to show that the following temporal logic properties remain true under application of the garbage collector. Remember active cells are those in the transitive closure of the root cell. We define the predicate *active* on cells by: *active(n)* is true iff $(\exists < root, n >) \vee (\exists < x, n > \wedge\ active(x))$

I1. Safety:

$$\Box \forall n. \neg(active(n) \wedge free(n))$$

I2. Comprehensive ie there are no space leaks:

$$\Diamond \forall n.(active(n) \vee free(n))$$

I3. Equivalence:

$$\Box(free(n) \Leftrightarrow RC(n) = 0)$$

I4. Invariance of reference count:

$$\Box(RC(n) = \mathrm{card}\{< x, n >: active(n)\})$$

We believe that the use of temporal logic to express the desired properties of garbage collection will be of particular value when we consider more advanced algorithms. Typically, such algorithms yield highly complex concurrent behaviour arising from the simultaneous interaction of the garbage collector and a mutator process.

4 Conclusions

We have shown how abstract requirements of garbage collection can be captured using temporal logic. The temporal logic specification can then be used as a basis for process algebra specifications which can involve varying amounts of parallelism. We presented two simple CCS specifications as an example, followed by a more complex specification of the cyclic reference counting algorithm. The verification of such algorithms was then briefly discussed.

References

[Boehm and Weiser, 1988] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[Cavelli and Horn, 1987] A. R. Cavelli and F. Horn. Proof of specification properties by using finite state machines and temporal logic. In *Protocol Specification, Testing and Verification, VII*, pages 221–233. North-Holland, 1987.

[Clarke *et al.*, 1986] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[Collins, 1960] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[Dijkstra *et al.*, 1976] Edsgar W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[Gries, 1977] David Gries. On believing programs to be correct. *Communications of the ACM*, 20(1):49–50, January 1977.

[Jones and Lins, 1993] Richard E. Jones and Rafael D. Lins. Cyclic weighted reference counting without delay. In *PARLE93*. Springer-Verlag, 1993.

[Larsen and Milner, 1992] K.G. Larsen and R. Milner. A complete protocol verification using relativised bisimulation. *Journal of Information and Computation*, 1992.

[Lins, 1990] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. Technical Report 75, The University of Kent at Canterbury Computing Laboratory, The University, Canterbury, Kent, July 1990. to appear in Information Processing Letters.

[Manna and Pnueli, 1992] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[Martinez *et al.*, 1990] A.D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[McCarthy, 1960] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[Milner, 1989] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Parrow, 1985] J.G. Parrow. *Fairness Properties in Process Algebra*. PhD thesis, Department of Computer Systems, Uppsala University, Sweden, 1985.

[Parrow, 1987] J.G. Parrow. Verifying a csma/cd-protocol with ccs. Technical Report Report ECS-LFCS-87-18, Computer Science Dept, University of Edinburgh, 1987.

[Pnueli, 1977] A. Pnueli. The temporal logic of programs. *Foundations of Computer Science*, 18:46–57, 1977.

[Richier *et al.*, 1987] J.L. Richier, C. Rodriguez, J.Sifakis, and J. Voiron. Verification in XESAR of the sliding window protocol. In *Protocol Specification, Testing and Verification, VII*, pages 235–248. North-Holland, 1987.

[Rovner, 1985] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL–84–7, Xerox PARC, July 1985.

[Sanderson, 1982] M.T. Sanderson. *Proof Techniques for CCS*. PhD thesis, Computer Science Dept, University of Edinburgh, 1982.

[Schorr and Waite, 1967] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[Turner, 1985] David A. Turner. Miranda — a non-strict functional language with polymorphic types. In Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computerr Architecture*, pages 1–16, 1985. Springer Verlag LNCS 201.

[Wirth and Jensen, 1974] N. Wirth and K. Jensen. *The Pascal User Manual and Report*. Springer-Verlag, 1974.