Published in SETSS 92, 30th March to 1st April 1992, Florence

IMPLEMENTING A FUNCTIONAL LANGUAGE FOR HIGHLY PARALLEL REAL TIME APPLICATIONS

J. L. Armstrong, B. O. Däcker, S. R. Virding and M. C. Williams

Computer Science Laboratory, Ellemtel Utvecklings AB

Introduction

This paper describes a fast, highly portable implementation of the real time, symbolic, declarative programming language **Erlang** (1).

A previous paper (2) described our investigations into the suitability of different programming languages for use in telecommunications switching software. We concluded that declarative languages are highly suitable for the implementation of such systems and identified several properties which we regarded as essential in languages for programming robust, highly concurrent real time systems.

A subsequent paper (3) described the prototyping work which verified and enhanced these ideas. This resulted in a slow interpreter which implements a new declarative real time programming language (**Erlang**) (1).

Erlang has been used internally in Ericsson for prototyping switching systems and tens of thousands of lines of **Erlang** code have been written. Projects in which **Erlang** has been used have demonstrated at least an order of magnitude reduction in the work load of the specification - programming - testing cycle when compared to projects using conventional technology. The interpreter which was used to implement **Erlang** was, however, too slow for use in commercial applications.

This paper describes some of the implementation techniques used in our fast and highly portable implementation of **Erlang**. We present measurements of the execution speed of **Erlang** when used to implement a prototype control system for a PABX.

Summary of the Implementation

Our implementation of **Erlang** has been inspired by the techniques usually used to implement Prolog (4). **Erlang** code is compiled, by a compiler itself written in **Erlang**, into instructions for a virtual machine. These instructions are then interpreted by an emulator. In the virtual machine, each process has its own process header stack and heap (more about this later). The emulator is written in **C**. In the following section we describe the virtual machine by way of examples. Note that the instruction set used here has been simplified for pedagogic reasons.

Simple Sequential Erlang

Consider the following function which computes the factorial of a positive integer:

```
-module(test).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
```

We refer to this function as *factorial/1* i.e. the function factorial with one argument. This compiles into the following instructions:

```
        {info,test,factorial,1}            [1]
        {try_me_else, label_1}             [2]
        {arg, 0}                           [3]
        {getInt, 0}                        [4]
        {pushInt, 1}                       [5]
        ret                                [6]
label_1 try_me_else_fail                   [7]
        {arg, 0}                           [8]
        dup                                [9]
        {pushInt, 1}                       [10]
        minus                              [11]
        {call,local,factorial,1}           [12]
        times                              [13]
        ret                                [14]
```

Although the *factorial/1* function above has two clauses, the resulting code has one entry point at [1] above. We start by trying to match the first clause, *factorial(0) -> 1;*

Instruction [2] indicates where to start trying to match the next clause (*label_1*) if the first clause should not match. Instruction [3] pushes the first (and in this case only) argument onto the top of the stack. Instruction [4] tests if the top of the stack contains the integer 0. If it does contain the integer 0 the stack is popped, the return value of the function, the integer 1, is pushed onto the stack [5] and we return from the function [6].

If, however, instruction [4] finds something other than the integer 0 at the top of the stack, failure occurs. This results in a jump to *label_1*, as indicated by instruction [2]. The stack is also restored as it was at instruction [2]. Instruction [7] indicates there are no more clauses to match if failure occurs. Any subsequent failure will cause a run time error. Instruction [8] pushes the argument of the function on the top of the stack. Instruction [9] duplicates the top of the stack, [10] pushes the integer 1 onto the top of the stack [11] subtracts the top of the stack from the second location on the stack, pops the top two locations from the stack and pushes the result onto the stack. This corresponds to the calculation of *N - 1* .

The stack now contains a copy of the argument *N* in the second location and *N - 1* on the top. Instruction [12] calls the function *factorial/1* recursively. When this function returns the result of calculating *factorial(N - 1)* will be on the top of the stack and *N* as the next location of the stack (put there by [9] above). Instruction [13] multiplies these, pops them from the stack and pushes the result onto the stack and we return from the function [14].

Representation of Data

All data objects in **Erlang** are represented by a word with a tag as follows:

The integer 1234, for example, is represented as:

box; lw(0.6i) | lw(1.6i). TAG VALUE

Atoms are represented as:

box; lw(0.6i) | lw(1.6i). INTEGER 1234
box; lw(0.6i) | lw(1.6i). ATOM ATOM TABLE IN-

The atom table index is an index in a hash table which stores the strings of ascii characters corresponding to each atom. In the following we represent atoms (for example the atom *subscriber_no*) as follows:

DEX

So far we have been able to represent all data objects in one tagged word. Such objects can be stored in the stacks of processes. More complex structures are stored on the heaps of processes.

The **Erlang** tuple {*position, 23, 96*} is stored in consecutive heap locations:

box; lw(0.6i) | lw(1.6i). ATOM subscriber_no
box; lw(0.6i) | lw(1.6i). SIZE 3 _ ATOM position

The first item on the heap is an object with the tag *SIZE* indicating how many objects are in the tuple i.e. how many of the subsequent heap locations contain objects in the tuple.

References to this tuple (which may be stored in the stack or heap) are represented as:

_ INTEGER 23 _ INTEGER 96

Lists in **Erlang** are represented by linked lists of two consecutive heap locations:

box; lw(0.6i) | lw(1.6i). TUPLE pointer to heap

The Erlang list *[11, 22, 33]* could thus be represented as (the figures on the right represent heap addresses):

On the stack

box; lw(2.4i). Head of List _ Tail of List

And on the heap:

lw(0.6i) | lw(1.6i) l. _ LIST 501 _
_ _ lw(0.6i) | lw(1.6i) | r _ _ lw(0.6i) | lw(1.6i) | r _ _.

INTEGER	11	501
LIST	417	502

_ _ lw(0.6i) | lw(1.6i) | r _ _ lw(0.6i) | lw(1.6i) | r _ _.

INTEGER	22	417
LIST	287	418

_ _ lw(0.6i) | lw(1.6i) | r _ _ lw(0.6i) | lw(1.6i) | r _ _.

INTEGER	33	287
NIL		288

Building Complex Structures

The following fragments of code shows how we build a tuple when calling a function:

put_data_base(address, {joe, main_street, 23}),

This compiles into:

{*pushAtom,address*}	*[1]*
{*pushAtom,joe*}	*[2]*
{*pushAtom,main_street*}	*[3]*
{*pushInt,23*}	*[4]*
{*mkTuple,3*}	*[5]*
{*call,local,put_data_base,2*}	*[6]*

Instruction [1] above pushes the first argument of the call to *put_data_base/2* onto stack i.e the atom *address*. Instructions [2], [3] and [4] push the three components of the second argument, the tuple {*joe, main_street, 23*} onto the stack. Instruction [5] makes the top three locations of the stack into a tuple. It constructs the tuple on the heap and pops the three components of the tuple from the stack and pushes a tagged pointer to the tuple onto the stack. Instruction [6] calls the function *put_data_base/2*.

When building lists, we build one head and tail pair at a time, starting from the end of the list. Consider the the following call:

analyse_number([3,4,5]),

This compiles into:

pushNil	*[1]*
{*pushInt,5*}	*[2]*
mkList	*[3]*
{*pushInt,4*}	*[4]*
mkList	*[5]*
{*pushInt,3*}	*[6]*
mkList	*[7]*
{*call,local,analyse_number,1*}	*[8]*

Instructions [1] and [2] push the components of the last head and tail pair of the list onto the stack. The mkList [3] instruction creates the head and tail pair on the heap, pops the top two objects from the stack and pushes a tagged list pointer to the newly created pair onto the stack. Instruction [4] pushes the head of the next head and tail pair to be created onto the stack. The mkList [5] instruction will, as before, create a new head and tail pair on the heap. The tail element of this pair will be the newly created tagged list pointer to the last head and tail pair of the list, thus creating a linked list of pointers. Instructions [6] and [7] create the first head and tail pair of the list leaving a tagged list pointer to this on the stack. The call to *analyse_number/1* is made in instruction [8].

Pattern Matching

In **Erlang**, selection (choice) is done by pattern matching. Consider a function which has three clauses with the following heads:

```
destination(X) when integer(X) ->
        ......
destination(local) ->
        ......
destination({remote, case_1}) ->
        ......
```

This compiles into:

```
        {info,test,destination,1}         [1]
        {try_me_else,label_1}             [2]
        {arg,0}                           [3]
        {test,integer}                    [4]
        ......
label_1 {try_me_else,label_2}             [5]
        {arg,0}                           [6]
        {getAtom,local}                   [7]
        ......
label_2 try_me_else_fail                  [8]
        {arg,0}                           [9]
        {unpkTuple,2}                     [10]
        {getAtom,remote}                  [11]
        {getAtom,case_1}                  [12]
        .......
```

The entry point of the function is at instruction [1] above. We first try to match the first clause.
destination(X) when integer(X) ->
Instruction [2] indicates where to continue if we fail to match the head of the first clause. Instruction [3] pushes the argument of the function onto the stack and instruction [4] tests if the argument is an integer (*when integer(X)*). If this fails, i.e the argument is not an integer, we proceed to instruction [5] (*label_1* as in instruction [2]).

We now try to match the clause,
destination(local) ->
Instruction [5] tells us to continue at *label_2* if this clause does not match. We push the argument onto the stack [6] and test if it is the atom *local*. If this is not the case we proceed to instruction [8] (*label_2*).

We now try to match the last clause,
destination({remote, case_1}) ->
Instruction [8] tells us that this is the last clause and we will have a run time failure if the clause should not match. We push the argument onto the stack [9]. Instruction [10] tests if the arguments is a tuple of size 2 (i.e. with two elements). If it is we pop the stack and push the elements of the tuple onto the stack. We now proceed to test if the elements are the atom *remote* [11] and *case_1* [12].

Variables

Variables are introduced in **Erlang** programs simply by naming them. Consider the following program fragment in which we unpack a tuple.

```
var(A) ->
    {B,C} = A,
    C.
```

This is compiled into the following code:

```
{info,test,var,1}            [1]
try_me_else_fail             [2]
{alloc,2}                    [3]
{arg,0}                      [4]
{unpkTuple,2}                [5]
{storeVar,{'B',{var,0}}}     [6]
{storeVar,{'C',{var,1}}}     [7]
{pushVar,{'C',{var,1}}}      [8]
ret                          [9]
```

The function *var/1* introduces two variables, *B* and *C*. Space on the stack for these variables is reserved by instruction [3]. Instruction [5] unpacks the tuple contained in argument *A* and checks that it is of size 2. Instructions [7] and [8] put the components of the tuple into variables (*B* is variable 0 and *C* is variable 1). Instruction [8] pushes the return value (in variable 1) onto the stack.

Processes and Concurrency

So far we have just looked at sequential **Erlang**. **Erlang** is designed for use in highly concurrent real time systems and thus implements lightweight processes. Each process has its own stack, heap and process header. The process header stores the registers of the virtual machine when the process is not executing. Processes awaiting execution are linked into a scheduler queue. Processes are scheduled on a round robin basis. A process executes until either it consumes a time slice or until it awaits a message from other processes.

Sequential **Erlang** as shown above requires the following registers in the process header:

Stack Start of the stack.

Stack_top
Top of the stack (i.e. next free stack location).

Heap Start of the heap.

Heap_top
Top of the heap (i.e. next free heap location).

PC Program counter, a pointer to the instruction currently being executed.

Fail Address of the instruction to which to jump if failure occurs (i.e. during pattern matching). This is set to zero by *try_me_else_fail* instructions indicating that failure is to cause run time errors.

Stack_Save

This is set to the value of *Stack_top* by *try_me_else* instructions. When failure occurs and there are other alternatives to match (i.e. the *Fail* register does not contain zero) we reset the *Stack_top* register to the value of *Stack_Save*.

Processes headers are stored in a process table. Processes are referenced by special tagged data objects, process identifiers (or *PIDs*). One of the things contained by a PID is the index in the process table.

The following is an example of how a process could send a message to another process:

```
send(Analyser,Number) ->
    Analyser ! {analyse, Number}.
```

The function *send/2* above sends the tuple {*analyse, Number*} to the PID in *Analyser*. This compiles to the following instructions:

```
{arg,0}                [1]
{pushAtom,analyse}     [2]
{arg,1}                [3]
{mkTuple,2}            [4]
send                   [5]
ret                    [6]
```

The first argument to the function *send/2* is pushed onto the stack [1]. The tuple {*analyse, Number*} is created on the heap by [3], [4] and [5]. This leaves the stack with a tagged pointer to this tuple at the top and the contents of variable *Analyser* in the second stack location.

The *send* instruction is now executed. A check is first made that the second stack location is a valid PID. If it is we copy the contents of the top of the stack to the heap of the process denoted by the second stack location. Note that this means copying the entire data structure, i.e. if the top of the stack is a tagged pointer (e.g. a pointer to a list or a tuple) we copy the data to which it points. Thus we never have pointers from the heap of one process into the heap of another.

Each process may have a queue of messages which have been sent to it and are waiting to be processed. This queue is maintained as a list on the heap of each process. Having copied the message to the receiving process, we now link the message onto the the end of this list. To facilitate this, each process header has two more registers.

Message_queue

This contains the head of the message queue (i.e. the list).

Last_in_message_queue

This is a pointer to the last element in the message queue. This facilitates adding messages to the end of the queue.

When a message is delivered to a process, the receiving process is placed on the scheduler queue.

Erlang implements active message reception in a manner similar to SDL (5). When we receive a message, we try to match it against the alternatives in a receive statement. If there is no match, we retain the message on the message queue so that it may be processed later. Consider the following code fragment:

```
receive
        {analyse,Number} ->
                .......;
        {change,Number,Destination} ->
                .......
end,
```

This is compiled into the following code:

```
label_2  wait                                   [1]
         {try_me_else,label_3}                  [2]
         {unpkTuple,2}                          [3]
         {getAtom,analyse}                      [4]
         {storeVar,{'Number',{var,0}}}          [5]
         join_save_queue                        [6]
         ........
label_3  {try_me_else,label_4}                  [7]
         {unpkTuple,3}                          [8]
         {getAtom,change}                       [9]
         {storeVar,{'Number',{var,0}}}          [10]
         {storeVar,{'Destination',{var,1}}}     [11]
         join_save_queue                        [12]
         .........
label_4  save                                   [13]
         {goto,label_2}                         [14]
```

Instruction [1] (*wait*) causes the process to be suspended if there are no message awaiting reception. If there are one or more messages, the first message is pushed onto the stack. The *try_me_else* chain used (instructions [2] and [7]) work in the same way as in the pattern matching of the heads of functions. Instructions [2] - [5] are used to match the tuple {*analyse, Number*} and instructions [8] - [11] are used to match the tuple {*change, Number, Destination*}. If neither of these tuples match, the *try_me_else* chain causes a jump to instruction [13]. This instruction, *save*, saves the current message (which did not match). Instruction [14] jumps back to the start at instruction [1] (*wait*) either causing the process to be suspended if there are no more messages in the message queue, or pushing the next message on the message queue onto the stack.

If we match one of the tuples, we execute a *join_save_queue* at [6] or [12]. This instruction removes the message which matched from the message queue and replaces any messages which were saved by instruction [13] into the message queue.

To facilitate this there is another register for each process:

Save_point

This register points to the next message in the message queue which is a candidate for reception in the current message reception. In this way we need not start matching from the beginning of the message queue every time a message is received.

More Advanced Features

Erlang also has facilities for registering processes, timeouts in message reception and highly sophisticated mechanisms for handling run time errors. This also effects the virtual machine and the emulator. Discussion of this is beyond the scope of this paper.

Compilation and Linking

Erlang modules are the units of compilation. Each module is compiled to instructions (as described above). These instructions are then assembled, and an object code file is produced for each compiled module. Internal references within each module can be resolved at compile time. External references (i.e. functions in one module which call functions in other modules) are resolved at load time.

Each **Erlang** module contains an export clause defining the functions in the module which are accessible from *outside* the module.

Consider, for example, the factorial function we examined previously.

```
-module(test).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
```

We see that the function *factorial* with one argument (we write *factorial/1*) is exported from the module *test*. Within the module *test* we can call *factorial/1* with the argument 5 simply by writing: *factorial(5)* This will generate the instruction: {*call,local,factorial,1*} the addresses of which may be resolved at compile time.

If we call *factorial/1* from within another module we must write: *test:factorial(5)* This says, call *factorial/1* in the module *test*. This will generate the instruction: {*call,remote,test,factorial,1*} The addresses here can first be resolved at load time. To make this possible, the addresses of all exported functions are kept in an *export_table*. Each exported function is allocated a slot in this table either when the module exporting the function is loaded, or when a module making a call to the function is loaded. The index of this slot is patched into the object code at load time. Calls to exported functions are thus always made via this table. This means that the order in which modules are loaded is not important. It is also the basis for code replacement in a running system, loading a new version of a module means that we can replace the addresses of functions in the *export_table*. New calls to functions in the replaced module will thus use the new code.

Last Call Optimisation

Consider the following function which computes the sum of a list of integers:

```
sum([H | T]) ->
        H + sum(T);
sum([]) ->
        0.
```

In the above we cannot evaluate the "+" until both its arguments are known. This means that we must compute *sum(T)* before we can return from the function. This in turn means that we will not be able to compute the first *sum(T)* before we get to the end of the list (*sum([]) -> 0*). Each call of *sum(T)* creates a new stack frame (i.e. return value and arguments of the function). If the list is long this will require a large stack.

An alternative way to write the *sum/1* function is:

```
sum(X) ->
        sum(X, 0).
sum([H | T], Acc) ->
        sum(T, H + Acc);
sum([], Acc) ->
        Acc.
```

In this version we introduce an accumulator (*Acc*) set to 0 at the start and then passed as a parameter to *sum/2*. The *sum/2* function now simply returns the result of the recursive *sum(T, H + Acc)* call. The *Erlang* compiler recognises this and, instead of using a {*call,local,sum,2*}, uses a {*enter,local,sum,2*} instruction. This instruction does not set up a new call frame on the stack, but overwrites the current call frame. The function *sum/2* will not use up stack space.

This optimisation is called the last call optimisation since it is always the last call made by a function which can be treated in this way. It is vital that long lived processes which are common in real time systems are written this way. The following is an example of such a process

```
server(Data) ->
   receive
        {From, Info} ->
           Data1 = process_info(From, Info, Data),
           server(Data1);
        {From, Ref, Query} ->
           Reply = process_query(From, Query, Data),
           From ! {Ref, Reply},
           server(Data)
   end.
```

The process in the example above executes a loop which ends with a recursive call. Last call optimisation applied to *server(Data1)* and *server(Data)* ensures that the process will execute in constant stack space.

Garbage Collection

Space on the stack and the heap are allocated sequentially. Call frames, temporary variables, etc are pushed onto the stack, when function calls are made. All the stack space allocated by a function is reclaimed when the function returns. The space on the heap is not reclaimed until all the heap allocated to a process is used. At this time many of the objects on the heap are probably no longer in use (i.e. they are not referenced).

When the heap is "full", and we need more heap space, we call the garbage collector. The garbage collector is, in fact, not a garbage collector but a collector of valid data (i.e. data to which there are references). Before garbage collection, memory is allocated for a new heap.

Starting from the top of the stack, we scan the stack looking for references to the heap. Referenced objects are copied from the old heap to the new and references on the stack are adjusted to point to the new heap. When we have scanned the stack, the new heap may contain references to the old heap. We, therefore, scan the new heap looking for these references. When we find such references, we copy the objects to which they point onto the end of the new heap and adjust the pointers on the new heap. Garbage collection is finished when we reach the end of the new heap. We can now free the memory allocated to the old heap.

This method of garbage collection relies on the fact that there cannot be circular references on the heap since **Erlang** does not have destructive assignment.

Since **Erlang** processes are small and garbage collection is done on a per process basis, the time for each garbage collection is very small.

Dynamic Resizeing of Processes

If, when we have performed a garbage collection, we still require more heap space, we allocate a bigger heap and call the garbage collector again so the old heap is copied to the bigger new heap.

If we should require more stack space, we allocate memory for a bigger stack and copy the old stack.

In the same way if we find that a process has much more stack and heap space than it requires, we can shrink the stack and heap.

The size of an **Erlang** processes thus varies according to the space required. Typically a process is created with 200 words of stack and 400 words of heap.

Dynamic resizing of processes is very useful in switching system applications. For example a subscriber line is usually inactive, i.e. in most cases subscribers (hopefully) spend much more time not using the telephone than using it. A process representing a subscriber line is, in **Erlang**, just a few hundred words when the line is idle but may expand to a few thousand words when the line is in use. In this way we can make the **Erlang** process structure isomorphic to the concurrency inherent in the application without using an excessive amount of memory.

Applications

Our **Erlang** system has been installed in over thirty sites worldwide and has been used to control hardware in many different applications including optical cross connect systems, switches for cordless telephony, multimedia experiments and PABX systems.

Erlang has proved highly suitable for building a support system for **Erlang**. The compiler and debugger for **Erlang** has been written in **Erlang**. An interface to the X-Windows system (6) has been written and is widely used both in the **Erlang** support system and as command and simulation interfaces in various applications.

Erlang object code is very compact. The basic **Erlang** operating system including the compiler, the code loading system, the interfaces to the file system and to X-Windows requires less that 60 Kbytes of object code. Under UNIX, the process runing this requires less than 2Mbytes of memory.

The system has also proved highly portable. Versions of the system have been ported to VAX 11/750, DS3100 (MIPS processor), SUN-3 (68020) and SUN-4 (SPARC) running various forms of UNIX and to a Force CPU-30 processor running VX-Works (7).

Measurements

The systems described below were made using an Ericsson MD 110 PABX which has been modified (2) so that it can be controlled by a standard work station, in our case a SUN SPARC Station 2 ®.

A large prototyping experiment for a new PABX architecture resulted in a PABX control program with the following:

Basic calls and ten features, both internal and over a network.

Basic operator features

Digital extension lines

Operator lines

ISDN Trunk lines

The system also includes a sophisticated dynamic feature loading mechanism in which telephony features may be loaded independently of each other.

Little attempt was made to optimise this as regards execution efficiency. The system comprises about 25 000 lines of **Erlang** source code. Setting up a simple (POTS) call requires 160ms CPU time.

A more simple demonstration system (which was made in conjunction with the ISS-90 conference) implements:

Basic calls and ten features

Digital extension lines

Analog extension lines

This system comprises about 5 000 lines of **Erlang** source code. Setting up a call requires 35ms CPU time.

Processes were used very freely in the above two systems. For example, each extension line is controlled by its own individual process, and each call in the system is controlled by two half-call processes. The telephone operating system (switch control, data base, number analysis, etc) also comprise a multitude of processes.

A very primitive system was designed with the objective of finding out how fast we could set up a telephone call. This system includes no features at all and comprises 2 500 lines of **Erlang** source code. Each call requires about 5ms CPU time.

In all three cases garbage collection accounts for about a quarter of the CPU time. This is much larger than in most applications in which Lisp or Prolog is used. This disparity is due to garbage collection in **Erlang** being done on a per process basis, and that processes in **Erlang** are kept small.

In all three cases a single processor was used. **Erlang** programs can be run in a distributed environment with little or no change to application programs. The capacity of the three systems above could thus be greatly increased by the use of more processors.

Conclusions

There are many examples (for instance (8)) of how the use of declarative languages can reduce the work load required to implement large software systems by at least an order of magnitude. Experience of using **Erlang** has indicated that similar productivity gains can also be made in the sphere of real time programming. Moreover our implementation of **Erlang** shows that it is also possible to execute such software sufficiently efficiently to be used in commercially viable systems.

References

(1) Armstrong J. L. and Virding R., *Erlang - An Experimental Telephony Programming Language*. International Switching Symposium, Stockholm, May 27 - June 1, 1990.

(2) Däcker B., Elshiewy N., Hedeland P., Welin C-W. and Williams M. C., *Experiments with Programming Languages and Techniques for Telecommunication Applications* 6th Int Conf. on Software Engineering for Telecommunication Switching Systems, Eindhoven, 14-18 April 1986.

(3) Armstrong J. L. and Williams M. C., *Using Prolog for Rapid Prototyping of Telecommunication Systems* Software Engineering for Telecommunication Switching Systems, Bournemouth, July 3-6, 1989.

(4) Warren, David H.D. *An Abstract Prolog Instruction Set. Technical Note 309, SRI International*, Menlo Park, CA, October 1983.

(5) CCITT Recommendation Z.100 *Specification Description Language. (SDL)*

(6) Scheifler R. W. and Gettys J. *The X Window System.* ACM Transactions on Graphics, Vol 5, No 2, April 1986, Pages 79-109.

(7) VX-Works Programmers Guide, Wind River Systems Inc.

(8) Reintjes, P. B., *A Set of Tools for VHDL Design* 8th International Conference on Logic Programming , Paris, June 24-28 1991.