UNLIMITED

RSRE
MEMORANDUM No. 4441

ROYAL SIGNALS & RADAR ESTABLISHMENT

SPRITE-ELLA LANGUAGE ENHANCEMENTS

Authors: M G Hill, E V Whiting & J D Morison

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 4441

UNLIMITED

CONDITIONS OF RELEASE

0088206 BR-115852

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

Royal Signals and Radar Establishment

Memorandum 4441

Title: SPRITE-ELLA LANGUAGE ENHANCEMENTS

Authors: M G Hill, E V Whiting, J D Morison

Date: November 1990

Summary

Language enhancements carried out for the ESPRIT 'SPRITE' project 2260 are discussed. These enhancements include naming of output signals, multiple identifier declarations, multiple instantiation of functions and advanced connectivity of circuits. A new primitive has also been added to the language for extending the timing model. The syntactic and semantic definitions of the enhancements together with examples of text are presented.

THIS PAGE IS LEFT BLANK INTENTIONALLY

Contents

THIS PAGE IS LEFT BLANK INTENTIONALLY

1 Introduction

This document outlines language enhancements to the Hardware Description Language ELLA^{TM} that have been carried out for the ESPRIT project 'SPRITE'. The ELLA language was chosen at the commencement of the project alongside Silage [2] as an input language to the Cathedral Silicon Compilers [1]. ELLA was chosen for the project because its semantics, particularly its applicative style, are well matched to signal processing tasks and because it has a well developed support environment. However in order to fulfil all the functions required of the SPRITE design language it was recognised that a number of enhancements to the language were required. It is these enhancements that will be described in this document.

The enhancements include the ability to name function output signals and their subsequent 'joining' within the function body, the instantiating of multi-dimensional rows of functions with greater flexibility in their external connection. The other major enhancement to ELLA has been in the introduction of timescaled regions. These are regions which have internal clocks running either faster or slower than the enclosing function. This enhancement was implemented so that ELLA can have the ability to model the functionality of Silage Interpolate and Decimate functions.

All the features described in this document are available in the current release of the SPRITE-ELLA system.

The language enhancements will be introduced in turn with their collective syntax given in appendices.

2 Multiple Declarations

Previous versions of ELLA (see [3]) have only supported single identifier declarations e.g.

```
LET a = FUNC(input1, input2).
```

Enhancements to the language now permit multiple identifier declarations in the left hand side of such expressions, for example

```
LET (a1,a2) = TWO_OUT_FUNC(input1, input2).
```

In ELLA sequence clauses VAR's and PVAR's have been enhanced to allow similar formats, and sequential assignment statements enhanced to allow the following

```
(a1, a2) := (b1, b2);
```

Each of these declarations will now be considered with a complete definition of the syntax given in appendix A.

2.1 LET Declarations

Multiple declarations allow collaterals of identifiers to be grouped together, thus giving users the ability to assign different names to different parts of a 'unit' clause, (see [3] for description of ELLA units), for example

```
LET (a1,a2) = TWO_OUT_FUNC(input1, input2).
```

where TWO_OUT_FUNC is a function which has two output signals. The number of identifiers declared in such statements must equal the number of signals of the value delivering (unit) clause.

Such declarations can also help to 'tidy up' series of LET statements. For example with multiple LET's such a set of statements as

```
LET a = d, b = e, c = f.
```

can be written as

```
LET (a,b,c) = (d,e,f).
```

Multiple LET's can also be used to simplify ELLA text, for example

```
LET dummy = CASE (in1,in2)
                OF (t,t): (t,f,t),
                   (f,bool)|(bool,f): (f,t,f)
            ESAC.
LET a = dummy[1], b = dummy[2], c = dummy[3].
```

can now be written as

```
LET  (a,b,c) = CASE (in1,in2)
                   OF (t,t): (t,f,t),
                      (f,bool)|(bool,f): (f,t,f)
               ESAC.
```

In certain cases it may not be required, or desirable, to name all the outputs of a 'unit' clause. In this case null names may be used, however there must be at least one non-null name, thus

```
LET  ( ,a, ) = (in1, in2, in3).        # legal #
```

is valid but

```
LET  ( , , ) = (in1, in2, in3).        # illegal #
```

is not.

A restriction imposed on multiple LET's is that they can't be used to decompose a STRING (a STRING is a special form of packed array possessing the global unknown property, see [11]). For example the following is not allowed

```
LET (a, b, c) = STRING [3] bit'0.      # illegal #
```

Multiple LETs can also be used in the ELLA sequence clause and they follow the identical rules as in the functional part.

2.2 Variable Declarations

Variable assignment declarations are used solely in the ELLA sequence clause and the introduction of assignment variables occurs through either a VAR or PVAR statement. These statements are enhanced in the same way as LET statements such that the following are all legal statements

```
VAR (a,b,c) := (t,f,x),
    (u,v,w) := (t, i/2, char'c);

PVAR (one, two, three) ::= (i/1, t, char'z),
     (unknown, value) ::=  (?bool, i/2);
```

In multiple VAR and PVAR assignments, as with LET statements, some but not all of the expected names may be null, for example

```
     VAR (a, ) := (i/2, t);            #  legal  #
PVAR (one, , ) ::=  [3]char'k;         #  legal  #

     VAR ( , ) := (i/2, t);            # illegal #
PVAR  ( , , )  ::=  [3]char'k;         # illegal #
```

2.3 Assignment Statements

Once a variable has been declared through a VAR or PVAR declaration it can be assigned to via an assignment statement. Like the variable declarations the assignment statement has been extended to allow collaterals of variables on the left hand side, for example

```
(one, two, three) := (a,b,c);

( , value, ) := (u, v, w);

(array[2..3][[in]], row[3], value) := (input1,input2,input3);
```

As with declarations, multiple assignments can have some but not all of the expected names given as null, for example

```
( , , a, ) := FOUR_OUT_FUNC(input);   # legal   #

( , , , )  := FOUR_OUT_FUNC(input);   # illegal #
```

where FOUR_OUT_FUNC is a function that delivers four output signals.

A further restriction which is placed on multiple assignments is that the same identifier cannot appear more than once in the left hand side. This is because multiple sequential assignment statements are treated as parallel statements within a sequence step. Thus there is no specified ordering of execution, and hence potentially ambiguous statements must not be allowed, for example

```
(a, b, c, d) := (in, a, a, b);              # legal   #

(array[2..4], b, c, array) := FOUR_OUT_FUNC(input);   # illegal #
```

However a statement of the form

```
(a, b) := (b, a)            # legal  #
```

is allowed and this swaps the contents of variables 'a' and 'b'.

3 Named Outputs

ELLA V4 only allowed functions with unnamed outputs, this meant that the output of a function could only be given through an OUTPUT statement. An enhancement carried

out for the SPRITE project allows the naming of outputs and their joining within the function body.

The format for an output type is similar to an input type, for example

```
FN A = (type:in) -> (type:out): ...
```

Of course unnamed outputs are still allowed in exactly the same format as before (see [3]).

By providing named outputs the user also has the choice of how the output may be joined, for example

```
FN A = (type:in) -> (type:out):
( ...
  ...
  JOIN in -> out.
).
```

or

```
FN A = (type:in) -> (type:out):
( ...
  ...
  OUTPUT in
).
```

These two examples show the only ways in which the output can be joined. Thus an output must be joined to by either an 'OUTPUT' statement or specific 'JOIN' statements. There can never be a mixture of part of an output signal being 'joined' by the OUTPUT statement and the rest by explicit JOIN's.

4 Enhancements to the Function Type Mechanism

Before continuing with the enhancements carried out for SPRITE an enhancement carried out for a UK IED (Information Engineering Directorate) project will be described. This enhancement is incorporated into the SPRITE-ELLA system since it is anticipated that it could provide extra functionality advantageous to transformational design. Only a brief overview will be given here, a complete account is provided in [10].

Function types in ELLA are an enhancement of the basic type mechanism and are used for defining signals that carry information in both directions. Before these enhancements function type signals were used in a limited way, as input signals to a function but never as output signals and hence function types required the use of function sets, which are an

extension to the ELLA function mechanism, in order to communicate. However it is now possible to have function types delivered as outputs, both in function specifications and value delivering clauses.

The definition of a function type has not changed and therefore a valid function type definition is

```
TYPE  enum1  = NEW (val1 | val2 | val3 ),
      enum2  = NEW (res1 | res2 | res3 ),
      enum1_to_enum2  = enum1 -> enum2.
```

where 'enum1_to_enum2' is a function type that has a signal 'enum1' in one direction and a signal 'enum2' in the other.

An example of a function with a named function type input and output is

```
FN B = (enum1_to_enum2:lhs) -> (enum1_to_enum2:rhs): ...
```

Where, internal to 'B', 'lhs' is a value delivering function type which is made up of a value delivering signal of type 'enum2' and a value requiring signal of type 'enum1'. Whilst 'rhs' is a value requiring function type which is made up of a value delivering signal of type 'enum1' and a value requiring signal of type 'enum2' (see [10]). Hence it is possible to connect the input signals of 'B' to its output signals, and this may be achieved is several ways. For example via an OUTPUT statement e.g.

```
FN B = (enum1_to_enum2:lhs) -> (enum1_to_enum2:rhs):
( ...
  OUTPUT IO lhs
).
```

or via JOIN statements e.g.

```
FN B = (enum1_to_enum2:lhs) -> (enum1_to_enum2:rhs):
( ...
  JOIN lhs -> rhs,
       rhs -> lhs.
).
```

or

```
FN B = (enum1_to_enum2:lhs) -> (enum1_to_enum2:rhs):
( ...
  JOIN IO lhs -> IO rhs.
).
```

or

```
FN B = (enum1_to_enum2:lhs) -> (enum1_to_enum2:rhs):
( ...
  JOIN IO rhs -> IO lhs.
).
```

Function types can now be used anywhere where an ordinary type is used providing it is meaningful. Thus replication of function types is not allowed since this would mean joining an input to a signal more than once. The syntax of function types is given in appendix B. For a complete description of function types and the consequences of their extension on the ELLA language the reader is referred to [10].

5 Multiple Instantiations

In ELLA V4 it was only possible to MAKE either a single instantiation or a row of instantiations of a function, for example

```
FN FUNC = (enum1:in) -> enum2: res1.

FN USE_FUNC = (enum1:in) -> enum2:
( ...
  MAKE FUNC:    func,
       [2]FUNC: func2.
```

With the SPRITE-ELLA system it is possible to instantiate rows of rows of functions and the number of rows of rows that can be instantiated is not restricted. Thus it is possible to say

```
MAKE [5][3][4]FUNC: multfunc.
```

and the type of 'multfunc' is [5][3][4]enum1_to_enum2, which is 'a row of row of row of function types'. The value delivering part of 'multfunc' is then of type [5][3][4]enum2, and the value requiring part of type [5][3][4]enum1.

In ELLA V4 it was necessary to supply each input to a row of makes separately. Clearly for the case of multiple rows joining each element separately would be too restrictive. Thus

the whole question of joining up ELLA circuits has been considered and the SPRITE-ELLA system allows inputs to multiple rows of makes to be supplied all at once, or separately, or in groups. The next section describes in detail the enhancements that have been made to ELLA's join mechanism.

6 Connectivity Enhancements

Earlier versions of ELLA only allowed a single identifier after the '→' of a JOIN statement, possibly followed by up to two indices (one for rows of makes and one for function set rows). This form of 'join to' structure was felt to be too limited in the light of the function type work and multiple makes. Thus the structure of the JOIN statement has been enhanced to the following form

JOIN unit → joinval.

where

```
joinval  :-  joinval1
             joinval CONC joinval1

joinval1 :-  joinval2
             [INT name = integer .. integer] joinval1

joinval2 :-  name
             IO name
             joinval2 [integer]
             joinval2 [integer .. integer]
             ( joinval <<, joinval>> )
```

with the 'unit' clause remaining unchanged, and <<, joinval>> representing zero, one or more repetitions of ", joinval".

A JOIN statement takes the value delivering part of the 'unit' and connects it to the value requiring part of the 'joinval'. Only in the particular case of joining IOidentifier1 to IOidentifier2, where both identifier1 and identifier2 are function types, is it irrelevant which way round the JOIN is done i.e. IOidentifier1 → IOidentifier2, or IOidentifier2 → IOidentifier1. Examples of join statements are

```
JOIN something -> (name[3] CONC name[2], name[4], name1[2..3]).

JOIN somethingelse -> [INT k=2..5](func_a[k], func_b[k]).
```

In addition to supplying extra syntactic constructs the JOIN mechanism has been extended to allow 'complete joins', and 'individual joins'. Complete joins allow all input or output signals of an identifier to be joined in one statement. Thus multiple makes no longer require separate JOIN statements for each row e.g.

```
FN PART = (enum1: in, enum2: ip) -> (enum2, enum1): (res2, val1).

FN MULT_PART = ([5][3][4](enum1, enum2):in) -> [5][3][4](enum2, enum1):
( MAKE [5][3][4]PART:part.
  JOIN in -> part.
  OUTPUT part
).
```

Individual joins are in a sense the opposite of this since they allow identifiers to have their signals joined one at a time. Thus single makes which used to require all their inputs to be supplied in one statement can now have them spread over several statements e.g.

```
FN PART = (enum1: in, enum2: ip) -> (enum2, enum1): (res2, val1).

FN USE_PART = (enum1:in) -> ((enum2, enum1):out):
( MAKE PART:part.
  JOIN in    -> part[1],
       res2 -> part[2],
       part[1] -> out[1],
       part[2] -> out[2].
).
```

Individual joins and complete joins are really extremes of the join mechanism, between them there is a variety of partial joins which allow signals to be connected in different ways, and it is up to the user to make the connection decisions.

6.1 Connectivity Transformation Example

The example in this section illustrates how a circuit using the new connectivity constructs can be transformed into a circuit which would be valid in earlier versions of ELLA.

Consider the following circuit

```
TYPE ty = NEW (t1|t2|t3).

FN F = (ty:in) -> ty: ...

FN G = (ty:in) -> ty: ...

FN DEL = (ty) ->ty:DELAY(?ty,1).

FN MAIN = (ty:in) -> (ty:out):
( MAKE F:f,
       [5]DEL: del,
       G:g.
  JOIN (in, f CONC del, g) -> (f, del CONC g, out).
).
```

where function MAIN merely joins its input to some function F then delays the output of F through five unit delays before passing the signal into function G. The output of MAIN being taken as the output of G. Transforming the function MAIN to remove the extended joins, and hence produce a function which would be valid in ELLA V4, gives

```
FN MAIN = (ty:in) -> ty:
( MAKE F: f,
       [5]DEL: del,
       G: g.
  JOIN in -> f.
  JOIN f -> del[1].
  JOIN del[1] -> del[2].
  JOIN del[2] -> del[3].
  JOIN del[3] -> del[4].
  JOIN del[4] -> del[5].
  JOIN del[5] -> g.
  OUTPUT g
).
```

It can be noted that the statements with connections from del[1] to del[4] could have been put into a multiple join statement of the form

```
FOR INT i = 1..4 JOIN del[i] -> del[i+1].
```

however at present the transformations in ELLA replace higher level constructs with lower level ones.

6.2 Regular Array Example

In this example a basic function called CELL is taken and combined to form an array of cells. Such an array can provide the basis for a systolic array. This example makes use of the extensions to the MAKEs, JOINs and function types. A particular example is illustrated in figure 1.

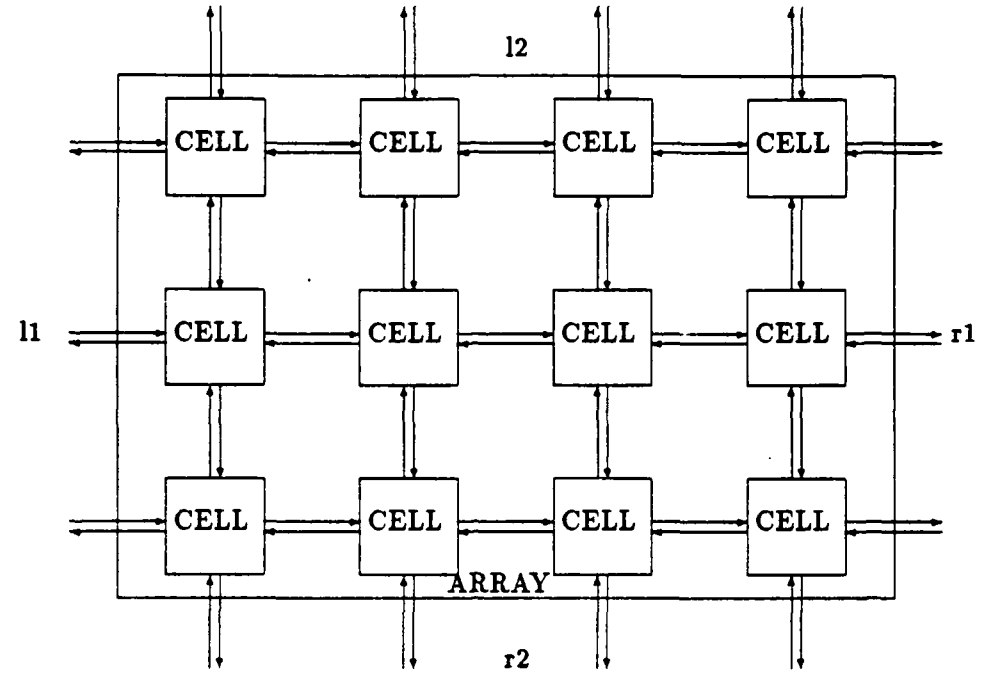


Figure 1: Array with m = 3, n = 4

```
TYPE a = ...
     b = ...

INT  m = ...
     n = ...

FN CELL = ((a->a): xin, (b->b): yin) -> ((a->a): xout, (b->b): yout):
( JOIN (IO xin, IO yin) -> (IO xout, IO yout). ).

FN ARRAY = ([m](a->a):l1, [n](b->b):l2) ->
                                    ([m](a->a):r1, [n](b->b):r2):
( MAKE [m][n]CELL:cell.

  #  INTERNAL JOINS  #
  FOR INT j = 1..m-1 INT i = 1..n-1
        JOIN (cell[j][i][1],cell[j][i][2])
                -> (cell[j][i+1][1],cell[j+1][i][2]).
  FOR INT i = 1..n-1 JOIN cell[m][i][1] -> cell[m][i+1][1].
  FOR INT j = 1..m-1 JOIN cell[j][n][2] -> cell[j+1][n][2].

  #  EDGE JOINS  #
  JOIN Il1 -> [INT j = 1..m]cell[j][1][1],
       Il2 -> [INT i = 1..n]cell[1][i][2],
       [INT j = 1..m]cell[j][n][1] -> Ir1,
       [INT i = 1..n]cell[m][i][2] -> Ir2.
).
```

This example shows how partial joins can be used to connect the array. However the joining of the cells within the array requires several statements. By redefining the specification of CELL the internal connections of array can be made more succinct. This is achieved by defining CELL as a function-set of two elements i.e.

```
FN CELL_2 = ((a->a)->(a->a):x, (b->b)->(b->b):y): (JOIN x->x, y->y.).
```

where 'x' describes the horizontal information and 'y' the vertical. Then the function ARRAY can be re-written as

```
FN ARRAY_2 = ([m](a->a):i1, [n](b->b):i2) ->
                                        ([m](a->a):r1, [n](b->b):r2):
(  MAKE [m][n]CELL_2: cell.
   LET vert  = [INT j = 1..m][INT i = 1..n] IO cell[j][i][2],
       horiz = [INT i = 1..n][INT j = 1..m] IO cell[j][i][1].
   JOIN
      IOi1 CONC horiz -> horiz CONC IOr1,
      IOi2 CONC vert  -> vert  CONC IOr2.
).
```

where ARRAY_2 has exactly the same functionality as ARRAY yet its internal connectivity is described in a much simpler way. A complete description of the enhancements to the function type/set mechanism for describing such functions as CELL_2 can be found in [10].

7 Timescaling

The timescaling enhancements for ELLA come in two parts. First the introduction of a new language primitive, second, the introduction of hierarchic time regions. The language primitive is a sample-and-hold construct which will allow synchronous descriptions to use ELLA time to describe clock periods other than one per time tick. The hierarchic approach will allow users to wrap up ELLA functions into regions which operate with clock periods either faster or slower than the surrounding region. These two features are related by a transformation and the simulator makes use of this fact.

For example, a hierarchically faster region running at four times the rate of the outer region, say, would be transformed into a region which has its inputs held constant for four time units and its output sampled ever four time units. The outer region would then have its delay times multiplied by four. Thus both regions would appear to operate within the same time frame, however only the inner region would change each time unit (the outer region effectively changing only every four time units). Thus users have at their disposal a new timing primitive which can be accessed either explicitly or implicitly. For a complete description of the enhancements and the implications on the ELLA system the reader is referred to [5-8].

7.1 Sample Primitive

The new sample-and-hold primitive occurs in the same syntactic position as the DELAY primitive, that is it is the sole contents of a function body. The syntax of the new primitive is of the form [6]

SAMPLE(interval_size, initial_value, skew)

or

SAMPLE(interval_size)

where if ommitted 'initial_value' and 'skew' default to ?type and zero, respectively. The 'skew' value determines the sample point and its value must be less than the interval_size.

The semantics of this function are given informally as

- At a time 't' if (t-skew) is some multiple of the interval size then take the current input value, otherwise the output remains unchanged.

A formal definition of the semantics of SAMPLE may be found in [8].

The functionality of SAMPLE may be demonstrated by considering the following example

```
TYPE ty = NEW (t1|t2|t3|t4|t5|t6|t7|t8|un).

FN SMP = (ty) -> ty: SAMPLE(4,un, 2).
```

In this case SMP samples the input signal every 4 units with the sample point occurring after 2 units. A typical simulation run then gives the following result

```
TIME   :  0   1   2   3   4   5   6   7   8   9  10  11

input  :  t1  t2  t3  t4  t5  t6  t7  t8  t1  t2  t3  t4
output :  un  un  t3  t3  t3  t3  t7  t7  t7  t7  t3  t3
```

7.2 Hierarchical Timing

Hierarchical timing is obtained by the use of the FASTER and SLOWER constructs. Both constructs appear in the same syntactic position as the SAMPLE primitive, i.e. the sole contents of a function body, and they instantiate a function to run at a simulation rate faster or slower (respectively) than the enclosing region.

The syntax for the FASTER construct is

FASTER(function_name, interval_size, initial_value, skew)

or

FASTER(function_name, interval_size)

where, 'function_name' is the name of the function which is to have its clock rate set at 'interval_size' times faster than the surrounding region. The parameter 'initial_value' determines the initial output value of the faster region and 'skew' sets the input sample position, if omitted 'initial_value' and 'skew' default to ?type and zero respectively. A similar syntax follows for SLOWER i.e.

SLOWER(function_name, interval_size, initial_value, skew)

or

SLOWER(function_name, interval_size)

again if omitted 'initial_value' and 'skew' default to ?type and zero respectively. The restriction of the size of 'skew' for FASTER and SLOWER is the same as for SAMPLE.

7.3 An Example Transformation

In this section a very simple circuit is shown to demonstrate how the FASTER and SLOWER features are transformed to circuits with sample primitives. Consider the following ELLA circuit which is shown in figure 2.

```
TYPE a = NEW (a1|a2).

FN DEL = (a) -> a:DELAY(?a,2).

FN F = (a:in) -> a: DEL in.

FN S = (a:in) -> a: DEL in.

FN FAST = (a) -> a: FASTER(F,4,a1,2).

FN SLOW = (a) -> a: SLOWER(S,3,a2,1).

FN MAIN = (a:in) -> a: DEL(SLOW(DEL(FAST in))).
```

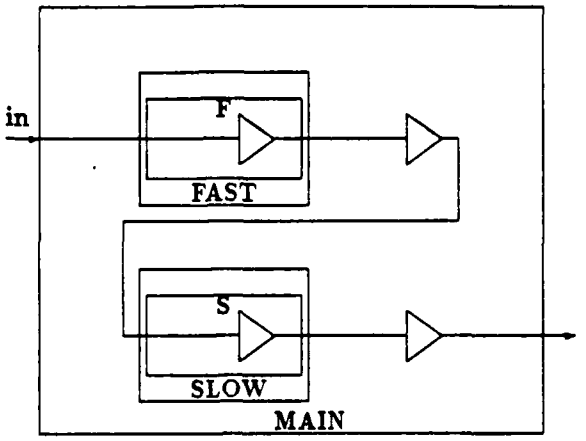


Figure 2: Faster and Slower

This circuit has little practical use and is given merely to illustrate the transformation procedure. The functions FAST and SLOW are transformed to remove the FASTER and SLOWER constructs using one of ELLAs in-built transformations. The resulting circuit can then be described in the following way.

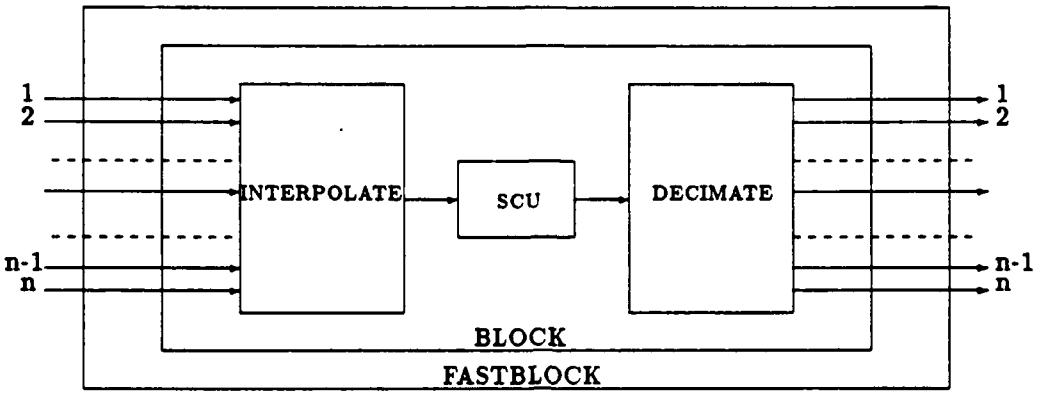
```
MAC NEW_DEL{INT period} = (a) -> a: DELAY(?a, period*2).

FN F_SAMPLE = (a) -> a: SAMPLE(4, a1, 2).

FN S_SAMPLE = (a) -> a: SAMPLE(12, a2, 4).

FN NEW_FAST = (a:in) -> a: F_SAMPLE(NEW_DEL{1}in).

FN NEW_SLOW = (a:in) -> a: NEW_DEL{12}(S_SAMPLE in).

FN NEW_MAIN = (a:in) -> a:
              NEW_DEL{4}(NEW_SLOW(NEW_DEL{4}(NEW_FAST in))).
```

The functions FAST and SLOW have now been transformed so that they have a SAMPLE function at the end and beginning of their respective new forms. Because SAMPLE is a core primitive, functions F_SAMPLE and S_SAMPLE are needed to instantiate particular instances. It can be seen that the delays external to the timescaled regions have also been transformed. If function NEW_MAIN is now simulated its 'clock' will be that of the common timebase.

7.4 Retiming Example

In this section we apply the new retiming constructs to a simple 'parallel to serial to parallel' circuit.

Consider the following circuit diagram.



which corresponds to the following ELLA description

```
TYPE word = ...

MAC INTERPOLATE = ([INT n]word:input) -> word:
( SEQ
     TYPE intcount = NEW ic/(1..n);
     FN INC = (intcount:in) -> intcount:
                ARITH IF in = n THEN 1 ELSE in+1 FI;
     PVAR count ::= ic/1;
     LET out = input[[count]];
     count :=  INC count;
     OUTPUT out
).


MAC DECIMATE {INT n} = (word:newvalue) -> [n]word:
( SEQ
     TYPE intcount = NEW ic/(1..n);
     FN INC = (intcount:in) -> intcount:
                  ARITH IF in = n THEN 1 ELSE in+1 FI;
     PVAR count ::= ic/1;
     PVAR out ::= [n]zeroword;
     LET pastout = out;
     out[[count]] := newvalue;
     count :=  INC count;
     OUTPUT pastout
).

FN SCU = (word:input) -> word: ...
```

```
FN BLOCK = ([n]word:input) -> [n]word:
( LET inter = INTERPOLATE input.
  OUTPUT DECIMATE {n} (SCU inter)
).

FN FASTBLOCK = ([n]word: in) -> [n]word: FASTER(BLOCK, n).
```

where the body of function 'SCU' has been left unspecified, this function represents some form of Serial Computation Unit. The keyword 'FASTER' in the function FASTBLOCK signifies entry into a region where the internal clock of the function 'BLOCK' is changing 'n' times faster than the outer clock. The circuit is thus modelling a parallel to serial (INTERPOLATE), a serial computation unit (SCU), and a serial to parallel (DECIMATE) function, where 'n' parallel signals are transformed into 'n' sequential signals which then pass through a computation unit running at 'n' times the outer rate, before being re-grouped in parallel. This circuit follows the format of the Silage [2] interpolate and decimate constructs.

The effect of the faster region on the way signals are handled is shown in the following example where SCU is taken as

```
FN SCU = (word:input) -> word:input.
```

Consider the case where n=4, and 'word' is an enumeration type defined by

```
TYPE word = NEW (a1 | a2 | a3 | a4 | b1 | b2 | b3 | b4 | bn ).
```

Then simulating the function 'BLOCK' with the input

```
input = (a1, a2, a3, a4)     for t = 0..3

input = (b1, b2, b3, b4)     for t = 4..7

input = (bn, bn, bn, bn)     for t = 8
```

gives the following result

```
 TIME      BLOCK          inter (internal node of
   0     ?  ?  ?  ?         a1       function BLOCK)
   1     a1 ?  ?  ?         a2
   2     a1 a2 ?  ?         a3
   3     a1 a2 a3 ?         a4
   4     a1 a2 a3 a4        b1
   5     b1 a2 a3 a4        b2
   6     b1 b2 a3 a4        b3
   7     b1 b2 b3 a4        b4
   8     b1 b2 b3 b4        bn
```

Simulating the function FASTBLOCK with the input

```
input = (a1, a2, a3, a4)     for t = 0  (i.e t_inner = 0..3)

input = (b1, b2, b3, b4)     for t = 1  (i.e t_inner = 4..7)

input = (bn, bn, bn, bn)     for t = 2
```

gives the following result

```
TIME        FASTBLOCK
  0         ?  ?  ?  ?
  1         a1 a2 a3 a4
  2         b1 b2 b3 b4
```

The results show that FASTBLOCK behaves in the same manner as BLOCK however from the user point of view FASTBLOCK only requires two simulation time units. In the transformations such regions of hierarchical timing will be transformed to a common time frame and sample-and-hold constructs would be appropriately placed in the circuit. A user could of course have written the circuit with sample-and-hold constructs from the outset, however the use of the hierarchical timing means that such detail can be hidden.

8 Conclusions

This document has described language extensions to ELLA that have been carried out for the SPRITE project. All the language features described are available in the current release of the SPRITE-ELLA system. These features include multiple LETs, multiple MAKES, enriched JOINs, named output signals and the inclusion of timescaled regions. In addition to this SPRITE also has access to the enhancements of the function type mechanism carried out under the IED (Information Engineering Directorate) ELLA Behavioural Synthesis project.

9 Acknowledgements

The authors would like to acknowledge the support of the ESPRIT 'SPRITE' project 2260. The authors would also like to acknowledge Praxis Electronic Design for their contribution to the timescaling enhancement.

10 References

1. ESPRIT II Technical annex SPRITE-2260, "Interactive Silicon Compilation for High Performance Integrated Systems", November 1988.

2. C. Sheers, "User Manual for the S2C Silage to C Compiler" Internal Report, IMEC 1988.

3. Praxis Electronic Design plc, 20 Manvers Street, Bath, BA1 1PX, UK, The ELLA Language Reference Manual, 1989.

4. Praxis Electronic Design plc, 20 Manvers Street, Bath, BA1 1PX, UK, The ELLA Tutorial, 1989.

5. P. Rouse, "Requirements Specification for Timescaling", Praxis document ref. no. E.N0045.20.16 (also appeared as Sprite deliverable D3.3/Praxis/Y1-M6)

6. P. Rouse, "Specification for Timescaling", Praxis document ref. no. E.N0045.50.12

7. M.G. Hill, J.D. Morison, "Timescaling in ELLA", ESPRIT Project 2260 deliverable reference number D3.3/RSRE/Y1-M6, 1989.

8. M.G. Hill, E.V. Whiting, J.D. Morison, "Formal Semantic Definition of ELLA Timing", RSRE Memorandum 4436, 1990.

9. M.G. Hill, E.V. Whiting, "ELLA Language Extension", ESPRIT Project 2260 deliverable reference number D3.1/RSRE/Y1-M12, 1989.

10. M.G. Hill, E.V. Whiting, J.D. Morison, "Bidirectionality, Connectivity and Instantiations in ELLA", RSRE Memorandum 4421, 1990.

11. M.G. Hill, N.E. Peeling, I.F. Currie, J.D. Morison, E.V. Whiting, C.O. Newton, "Real Number Arithmetic for Mixed Behavioural and Structural Descriptions", To appear in IEE Proceedings-G, Paper CDS 1382

THIS PAGE IS LEFT BLANK INTENTIONALLY

A Syntax of Multiple Declaration

The full ELLA syntax will not be described in this appendix, only those parts which have changed as a result of the work detailed in this document are given. For a complete description of the syntax the reader is referred to the ELLA language reference manual [3].

In this appendix the following convention and notation will be adopted

```
<< statements >> ==  zero, one or more 'statements'
     name        ==  a let, make or function terminal name
  declaration    ==  a TYPE, INT, CONST, FN or MAC declaration
   printitem     ==  a set of output strings or names
   faultitem     ==  a set of output strings or names
     unit        ==  a value delivering ELLA clause
   nullname      ==  a blank character
   integer       ==  a non-tagged integer
    value        ==  an ELLA CONSTant value
    step         ==  a step in a BEGIN ... END clause
 sequencestep    ==  a step in a BEGIN SEQ ... END clause
  seqchoice      ==  Sequential CASE chooser with value delivering clause
```

A.1 Syntax

applicative statements:

```
step          :-  LET letitem <<, letitem>>
                   declaration
                   MAKE makeitem <<, makeitem>>
                   << FOR multiplier >> JOIN joinitem <<, joinitem>>
                   PRINT printitem <<, printitem>>
                   FAULT faultitem <<, faultitem>>

letitem       :-  nameslist  = unit

nameslist     :-  name
                   ( nameornull, nameornull << , nameornull>> )

nameornull    :-  name
                   nullname
```

sequential statements:

```
sequencestep  :- LET letitem <<, letitem>>
```

```
                VAR varitem <<, varitem>>
                STATE VAR statevaritem <<, statevaritem>>
                PVAR pvaritem <<, pvaritem>>
                statement
                declaration
                PRINT printitem <<, printitem>>
                FAULT faultitem <<, faultitem>>

varitem       :- nameslist := unit

statevaritem  :- nameslist INIT values

pvaritem      :- nameslist ::= values

statement     :- varnameslist := unit
                 ( <<statement;>> statement )
                 [INT name = integer..integer] statement
                 IF boolean THEN statement  FI
                 IF boolean THEN statement ELSE statement FI
                 CASE unit OF seqchoice <<ELSEOF statement>> ESAC
                 CASE unit OF seqchoice <<ELSEOF statement>>
                                          ELSE statement ESAC

varnameslist  :- varname
                 ( varnameornull , varnameornull <<, varnameornull >> )

varnameornull :- varname
                 nullname

varname       :- name
                 varname[integer]
                 varname[[unit]]
                 varname[integer..integer]

values        :- value
                 ( value, value <<, value >>)
```

A.2 Semantics

The following two types of multiple statements are not allowed semantically

```
 i)  LET (a, b, c) = STRING [3]bit'0.

ii)  (array[1], array[2], array[1]) := THREE_CUT_FUNC(input);
```

In i) a STRING and a structure are considered to be different objects and so such an equality cannot be made. This approach is consistent with that adopted for the REFORM operator.

In ii) the ordering of the left hand side assignments cannot be assumed as they are treated as parallel statements within the assignment statement. Thus the result of this statement is undefined. A restriction was therefore placed on such assignment statements not to allow the same identifier more than once on the left hand side. In some cases this would be too strong a restriction, for example

```
(array[1], array[2]) := TWO_OUT_FUNC(input);
```

However to avoid any potential ambiguity that could arise the compiler implements the stronger restriction. Of course such statements, as shown above, can always be written without the use of such assignment statements.

THIS PAGE IS LEFT BLANK INTENTIONALLY

B Syntax of Function Types, Makes and Joins

The full ELLA syntax will not be described in this appendix, only those parts which have changed as a result of the work detailed in this document are given. For a complete description of the syntax the reader is referred to the ELLA language reference manual [3].

In this appendix the following convention and notation will be adopted

```
<< statements >> ==  zero, one or more 'statements'
     name        ==  a let, make or function terminal name
    fnname       ==  an upper case function name
     type        ==  any ELLA type
    fntype       ==  an ELLA function type
     unit        ==  a value delivering ELLA clause
   integer       ==  a non-tagged integer
```

B.1 Type

```
type          :-  name
                  [integer]type
                  (type <<, type>>)
                  fntype

fntype        :-  type -> type
```

B.2 Function Specification

```
  FN fnname = input -> output:          # An ordinary ELLA function #
  FN fnname = fnsetspec:                # An ELLA function set      #

input         :-  ()
                  (decs <<, decs>>)

output        :-  type
                  ()
                  (decs <<, decs>>)

decs          :-  type :names
                  type

names         :-  name <<name>>

fnsetspec     :-  ( fntypespecs <<, fntypespecs>> )
```

```
fntypespecs   :-  fnsetdecs : names
                  fnsetdecs

fnsetdecs     :-  [integer] fnsetdecs
                  ( fnsetdecs <<, fnsetdecs>> )
                  fntype
```

SEMANTIC NOTES ON FUNCTION SPECIFICATION

In a function set the 'fntype' can be a structure of function types, or rows of function types or a combination of the two. Unlike previous versions of ELLA there is no restriction on the sort of components a function set structure may have. Also function sets do not now need to be explicitly 'MADE' before they are used.

The input/output type given as '()' is a 'void-type' which indicates that there is no input/output terminal present. At present 'void-type' can only appear in a function specification as shown or as the input to an implicit function call or as the sole contents of an 'OUTPUT' clause. Extensions of the use of 'void-type' are being considered.

B.3 Multiple Makes

```
makeitem      :-  <<[integer]>> fnname : names
```

B.4 Partial Joins

```
joinitem      :-  unit -> joinval


joinval       :-  joinval1
                  joinval CONC joinval1

joinval1      :-  joinval2
                  [INT name = integer .. integer] joinval1

joinval2      :-  name
                  IO name
                  joinval2 [integer]
                  joinval2 [integer .. integer]
                  ( joinval <<, joinval>> )
```

C Syntax of Timescaling

The full ELLA syntax will not be described in this appendix, only those parts which have been added as a result of the timescaling are given.

In this appendix the following convention and notation will be adopted

```
 fnspec      ==  an ELLA function specification
 fnname      ==  an upper case function name
integer      ==  a non-tagged integer
constant     ==  an ELLA constant
```

C.1 Sample

```
FN fnname = fnspec: body.          # An ELLA function #

body          :-  SAMPLE(integer, constant, integer)
                  SAMPLE(integer)
```

where in the case of no constant or second integer being supplied they take the default values of unknown type and zero. If a second integer is present then its value must be between + or - the value of the first integer.

C.2 Faster and Slower

```
FN fnname = fnspec: body.          # An ELLA function #

body          :-  FASTER(fnname, integer, constant, integer)
                  FASTER(fnname, integer)
                  SLOWER(fnname, integer, constant, integer)
                  SLOWER(fnname, integer)
```

where in the case of no constant or second integer being supplied they take the default values of unknown type and zero. If a second integer is present then its value must be between + or - the value of the first integer.

THIS PAGE IS LEFT BLANK INTENTIONALLY

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....Unclassified.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4441	NOVEMBER	1990

Originators Name and Location

RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title

SPRITE-ELLA LANGUAGE ENHANCEMENTS

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
HILL, M G; WHITING, E V; MORISON, J D	27

Abstract

Language enhancements carried out for the ESPRIT 'SPRITE' project 2260 are discussed. These enhancements include naming of output signals, multiple identifier declarations, multiple instantiation of functions and advanced connectivity of circuits. A new primitive has also been added to the language for extending the timing model. The syntactic and semantic definitions of the enhancements together with examples of text are presented.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED

THIS PAGE IS LEFT BLANK INTENTIONALLY