RSRE
MEMORANDUM No. 4462

ROYAL SIGNALS & RADAR ESTABLISHMENT

ELLA CONNECTIVITY CONTROL

Author: M G Hill

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

RSRE MEMORANDUM No. 4462

CONDITIONS OF RELEASE

0093680 BR-116643

********** 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 4462

Title: ELLA CONNECTIVITY CONTROL

Author: M G Hill

Date: February 1991

Summary

This document describes the enhancements to the internal data structure of the ELLA compiler for the holding and checking of the connectivity information within an ELLA design. The consequence of this work on the rest of the ELLA system is discussed.

INTENTIONALLY BLANK

Contents

INTENTIONALLY BLANK

1 Introduction

This document describes the internal data structure of the ELLA compiler for the holding and checking of the connectivity information within an ELLA design. A reader of this document may require some understanding of the ELLA language since only those aspects of the system which effect the connectivity control will be detailed. The work described in this document has been completed and a large amount of testing of the code has been successfully carried out. This work will be incorporated into version 6 of the commercial ELLA product.

With the enhancements to the ELLA language of multiple instantiations, greater freedom in connectivity and the wider applicability of function types (see [1]), the whole structure of how connectivity information is stored and checked within the compiler has changed. In the next two sections of this document the new join syntax and connectivity compiler mode (JOINUP) will be described (Appendix A gives a brief outline of the compiler modes used in ELLA V4). The implication of the changes on the rest of the compiler and intermediate language are discussed.

2 Syntax

The JOIN construct in the ELLA language has been extended from that available in ELLA V4 (see [2]) to the following

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".

Without the above extension to the JOIN syntax the ways in which circuits could be connected was very limited, and hence the need for sophisticated compiler modes to keep

track of the connectivity information was avoided.

The above syntax means that it is now possible to have expressions of the form

```
JOIN (IOfntype, a CONC b, IOfntype)[1]          # Complex JOIN #
      -> (IOfntype[2..3],  [INT k=1..3]d[k], IOfntype)[3].

JOIN true -> and[1].                            # Partial JOIN #

JOIN [4][3][5](true, false) -> multiple_fn.     # Total JOIN #
```

where 'fntype' is an ELLA function type signal which has both a value delivering and value requiring component. These enhancements to the language mean that it is no longer feasible to use a simple compiler 'join' mode (see appendix A) for the monitoring of whether instantiations have had all their value requiring components supplied with signals. In particular for the case of an instantiation of a function with function types in both its input and output there are value requiring signals that need to be supplied in both parts of the function specification. For a complete description of function types and their use within the ELLA language the interested reader is referred to [1].

A new set of compiler modes have been defined which can hold all the necessary information about whether identifiers have had all there value requiring signals supplied. At certain parts of the syntax it is possible to mark off signals as having been joined, and the places where this occurs are the following.

- Just before the → of a JOIN statement
- At the very end of a JOIN statement
- The input to an implicit function call
- At the end of a LET statement
- At the end of the OUTPUT statement
- At the end of a function when no OUTPUT present

At all other places a join mode will only be changed if a fault occurs or if the mode needs to have a check in the post assembler phase. The modes which have been defined to hold the join information are described in the next section.

3 New Modes

The new modes introduced in the compiler are

```
JOINED = STRUCT (INT j),
JOINST = STRUCT (JOINUP jst, REF JOINST rest),
```

```
  JOINFN = STRUCT (JOINED joined),
 JOINROW = STRUCT (REF VECTOR [] JOINUP jrow),
  JOINIO = STRUCT (JOINUP source, sink),
JOINCASE = STRUCT (JOINUP jcase, REF JOINCASE rest),
 JOINREP = STRUCT (JOINUP jrep, JOINVARSTACK jvarstack),
JOININST = STRUCT (JOINUP jinst, input, BOOL hasfnout),
  JOINUP = UNION  (REF JOINST, REF JOINFN, REF JOINROW, REF JOINIO,
                         REF JOINED, REF JOINCASE, REF JOINREP,
                               REF JOININST),

JOINVARSTACK = STRUCT (INTEGER lwb, upb, BOOL index, dynindex,
                                                  REF JOINVARSTACK rest),
JOINCASESTACK = STRUCT (REF JOINCASE jcase, REF JOINCASESTACK rest),
JOINMULTSTACK = STRUCT (JOINUP joinup, BOOL source, REF JOINMULTSTACK rest),

INTEGERSTACK = STRUCT (INTEGER i, REF INTEGERSTACK rest),

  DECIDS = STRUCT (INT decno, origin, REF INTEGERSTACK makeint,
                                      TYPE type, JOINUP jsource, jsink),
```

where the old modes of DECID and MAKEID (see appendix A) have been replaced by the new mode DECIDS.

Each of these modes will now be considered.

3.1 DECIDS

```
  DECIDS = STRUCT (INT decno, origin, REF INTEGERSTACK makeint,
                                      TYPE type, JOINUP jsource, jsink)

      INTEGERSTACK = STRUCT (INTEGER i, REF INTEGERSTACK rest)
```

This mode now holds the information of all lower case identifiers where

```
origin  ==  inputtok,      # an input signal          #
            outputtok,     # an output signal         #
            lettok,        # a let identifier         #
            maketok,       # a make identifier        #
            fntok,         # a function set terminal  #
```

The field 'decno' is the declaration number, 'makeint' is a structure of integers which correspond to the integer sizes of multiple make statements (this field is set to NIL for single makes), 'type' is the type of the identifier and 'jsource', 'jsink' are the joinup information fields which will subsequently be ticked off when the identifier has been joined to. The

'jsource' field corresponds to the value delivering (source) part of the identifier and 'jsink' corresponds to the value requiring (sink) part.

3.2 JOINED

```
JOINED = STRUCT (INT j)
```

This mode is the bottom mode of JOINUP, that is all other modes eventually end at this mode. The 'int' in JOINED can take on the following values

```
jtrue    (j=1)   i.e. has been joined
jfalse   (j=2)   i.e. has not been joined
jcheck   (j=3)   i.e. macro parameter needs checking
jfault   (j=4)   i.e. a fault has occurred
jvoid    (j=5)   i.e. a void identifier
jcase    (j=6)   i.e. used in marking of a CASE clause
```

The 'jvoid' is used for a void type and is ignored by the system, thus a void may or may not be joined one or more times. At present the compiler syntax prevents widespread use of void, but if void is allowed greater freedom then this field might need changing.

As an example of a JOINED mode consider the following function specification

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

then 'in' has the mode JOINED with the int field set to 'jtrue', and 'out' has the mode JOINED with the int field set to 'jfalse'. Thus only 'out' can be joined to and when that has occurred the int field will go to 'jtrue'. Examples of the other int fields are given in the following example

```
FN A = ([2]bool:inp) -> ():                  # 'in' set to 'jtrue'        #
( ...                                        # output set to 'jvoid'      #
  ...
  MAC B {INT n} = ([n]bool:in, bool:c)       # 'in' set to 'jtrue'        #
                        -> ([n]bool:out):    # 'out' set to 'jcheck'      #
  BEGIN
     MAKE [2]F: func_f                       #  FN F = (bool)->bool:      #
     JOIN CASE c OF
             true: in[1],                    # 'jcase' used in joinup     #
             false: in[2]                    #   to mark all cases        #
          ESAC
            -> IO func_f.                    #  'jfault' due to error:    #
  END.                                       # since CASE delivers bool   #
  ...                                        # but IO implies bool->bool  #
).
```

3.3 JOINST

```
JOINST = STRUCT (JOINUP jst, REF JOINST rest)
```

This mode reflects the structure mode of TYPE and holds collaterals of JOINUPs. A JOINST mode can terminate with the following terminator

```
REF JOINST niljoinst = NIL
```

An example of a JOINST mode with four elements is

```
JOIN (a[1..2], b, IOf, d[2]) -> ...
```

where each element of JOINST will correspond to the individual parts of the collateral.

3.4 JOINFN

```
JOINFN = STRUCT (JOINED joined)
```

This mode is concerned with a function type. A JOINFN mode is setup after an identifier has been sourced or sinked, thus this mode only requires one field. The fact that the function type could be made up of other function types does not matter since by the time this mode is made up the whole function type it refers to is going to be joined (i.e. no other source /sink can occur before joining). Whenever a function type is assigned to a new identifier a new JOINFN is constructed from the type information.

For example a JOINFN mode will occur with the following function, first in the specification and then in the JOIN statement.

```
FN C = ( bool->(bool->[2]bool):in ) -> ( bool->(bool->[2]bool):out ):
( ...
  JOIN in -> out.
  ...
).
```

In the function specification the JOINUP mode of 'in' is in two parts, the source and the sink. The source field gets set to JOINFN, representing the function type 'bool→ [2]bool', and the sink field gets set to 'jtrue'. The reverse holds for 'out'. In the JOIN statement, the value delivering part ('in') will result in a JOINUP mode of JOINFN with the 'joined' field set to 'jfalse'. Once the '→' is encountered the 'joined' field of JOINFN gets marked off and hence the source field on 'in' gets marked as having been joined. A similar process occurs for 'out' on the right hand side of the statement.

3.5 JOINROW

```
JOINROW = STRUCT (REF VECTOR [] JOINUP jrow)
```

This mode reflects the ROW mode of TYPE. It is given as a row of joinups since elements of a row can be joined at different times and the joinup mode must be able to monitor this. An example of a JOINROW mode comes from the following

```
FN C = ([4](bool->bool):in) -> ():...
```

where 'in' has the mode JOINROW with the size of row being 4 and each element of the row being set to JOINFN with joined field 'jfalse'.

3.6 JOINIO

```
JOINIO = STRUCT (JOINUP source, sink)
```

This mode is created whenever an IO is used in an ELLA function. It has two fields which are called source/sink and these represent the value delivering /value requiring parts of the IO. For example

```
FN D = ( [2]bool->bool:in) -> ():
( ...
  JOIN IOin ->...                    # JOINIO made here #
  ...
).
```

where the JOINIO mode is made when 'IOin' is found. For the above example this leads to

```
JOINIO = (JOINED(jtrue), JOINROW ([JOINED(jfalse), JOINED(jfalse)]))
          # source #                  # sink #
```

3.7 JOINCASE

```
JOINCASE = STRUCT (JOINUP jcase, REF JOINCASE rest)

JOINCASESTACK = STRUCT (REF JOINCASE jcase, REF JOINCASESTACK rest)
```

A JOINCASE mode terminates with the following

```
REF JOINCASE niljoincase = NIL
```

The JOINCASE mode is created whenever a CASE statement is found. A separate mode is needed here since if a CASE statement is subsequently indexed all the arms of the case statement must be indexed, and by using a dedicated mode this is easy to do. An example of such a situation where this can arise is

```
JOIN ( CASE choice OF
         a: IOout,
         b: IOin,
         c: IOftype
       ESAC )[3] -> ...
```

where the JOINCASE mode is made at the ESAC position and then each element is indexed when the '[3]' is encountered.

The JOINCASESTACK mode is used to stack CASE statements which occur inside other CASE statements. Thus each element of the structure represents a different CASE statement.

3.8 JOINREP

```
JOINREP = STRUCT (JOINUP jrep, JOINVARSTACK jvarstack)

JOINVARSTACK= STRUCT (INTEGER lwb, upb, BOOL index, dynindex,
                                          REF JOINVARSTACK rest)
```

This mode represents the joining of replicators. If an identifier is indexed by a known integer then the appropriate field of the joinup mode can be selected. However in the case of replicators the indexing is not done until the end of the replication statement. Consider the following cases which are assumed to occur within a replicator statement

```
 i)     uc[index]

 ii)    uc[index][2..3]

 iii)   uc[index][[num]]
```

where 'index' is the replicator variable. Then the joinup modes are

```
 i)   JOINREP ( JOINUP of 'uc', (index, index, TRUE, FALSE, NIL) )

 ii)  JOINREP (JOINUP of 'uc', (2, 3, FALSE, FALSE,
                                (index, index, TRUE, FALSE, NIL)) )
```

```
iii)    JOINREP (JOINUP of 'uc', (num, num, FALSE, TRUE,
                                  (index, index, TRUE, FALSE, NIL)) )
```

In the case of an index or trim occurring before a replicator variable then this can be done without the need of stacking the integer on 'joinvarstack'. For example

```
uc[2][index]
```

where 'index' is a replicator variable, has the following joinup mode

```
JOINREP ( JOINUP of uc[2], (index, index, TRUE, FALSE, NIL) )
```

In the case where a JOINREP mode occurs within a collateral it becomes an element of JOINST just like any other joinup. Thus

```
(uc[index], uc CONC uc)
```

has the following mode at the end of the structure

```
JOINST( JOINREP( JOINUP of uc, (index, index, TRUE, FALSE, NIL)),
          JOINUP of 'uc CONC uc',
            niljoinst )
```

Once the joinup mode of the replicator clause has been made its replication can proceed. Consider the following example

```
[INT index = 1 .. 5] (uc[index], uc CONC uc)
```

In this example the resulting type is of the form [5](type, type2), and the joinup mechanism reflects this in order that such a joinup can be indexed at some point further on. For the typing information the new type is created in the compiler action <varmult3>, which occurs at the end of a replicator statement, and the new joinup mode is therefore constructed at the same place. The new mode made in <varmult3> is a JOINROW mode with a size equal to the number of replications. Thus in the last example the resulting joinup type after completing <varmult3> is

```
JOINROW ( [ JOINST(jst1[1], (jst2, niljoinst)),
            JOINST(jst1[2], (jst2, niljoinst)),
            JOINST(jst1[3], (jst2, niljoinst)),
```

```
    JOINST(jst1[4], (jst2, niljoinst)),
    JOINST(jst1[5], (jst2, niljoinst))
   ] )
```

where

```
jst1[1] == JOINUP of uc[index = 1]
jst1[2] == JOINUP of uc[index = 2]
jst1[3] == JOINUP of uc[index = 3]
jst1[4] == JOINUP of uc[index = 4]
jst1[5] == JOINUP of uc[index = 5]

jst2    == JOINUP of 'uc CONC uc'
```

and hence jst2 points to the joinup information of 'uc CONC uc' five times which means that when the joins are marked off an error will occur, which is the correct result for the above example. Note also that the JOINREP mode has been replaced. This is because when the JOINROW mode was made up the value of 'index' is known for each element, and hence the 'index' behaves as a known value which can be evaluated.

If the replicator variables range is unknown, i.e. due to macro parameters, then the JOINREP mode is marked off with 'jcheck' and information is output into the intermediate language for use by the post assembler system.

There is one other situation in which replicator checks are passed to the assembler. This is the case where a replicator contains a BEGIN...END clause in which there are locally declared MAKEs or LETs. This proved necessary since if the checks were to be done in the compiler multiple declarations would be needed, which would then be indexed by the replicator variable. For example consider the following

```
MAKE [5]A: a1.
JOIN [INT i = 1..5](MAKE A: a2.
                    JOIN a1[i]->a2, a2->a1[i].
                    OUTPUT ()) ->...
```

This example is of valid ELLA which should compile. In order to construct the correct form of JOINREP mode it would need to be transformed to

```
MAKE [5]A: a1.
MAKE [5]A: a2.
JOIN [INT i = 1..5](
                    JOIN a1[i]->a2[i], a2[i]->a1[i].
                    OUTPUT ()) ->...
```

This is clearly not an acceptable way to proceed and hence either the join mode needs to be extended so that locally declared makes/lets within replicators can be treated, or the compiler abandons the checks to the assembler. In order to do the checks in the compiler a mode similar to the assembler UNIT mode would be needed in order to ensure all scoping rules and occurrences had been taken into account. A decision was therefore taken to pass the checks to the assembler since a mechanism for doing such checks in the assembler already exists.

3.9 JOINMULTSTACK

```
JOINMULTSTACK = STRUCT (JOINUP joinup, BOOL source, REF JOINMULTSTACK rest)
```

A JOINMULTSTACK mode terminates with the following

```
REF JOINMULTSTACK niljoinmultstack = NIL
```

This mode is used in the following join statement

FOR INT i = 1 ..3 JOIN 'unit' → 'joinval'.

where it stacks the unit and joinval with the 'source' field set to true for the unit and false for the joinval. If there are several join statements within this multiple join then they are all stacked until the end of the complete multiple join statement. Once the multiple join statement has ended then each element of JOINMULTSTACK can be marked off in a similar way to that of the field of the JOINREP mode.

3.10 JOININST

```
JOININST = STRUCT (JOINUP jinst, input, BOOL hasfnout)
```

This mode is used whenever an explicit function instance is called. Such a mode is necessary to ensure a) a function instance with function type inputs is joined to once and once only b) a function instance with function type outputs is not indexed so that a function type does not get joined to (a problem anywhere). The boolean field reflects the information necessary to ensure the later point. An example of the creation of this mode is shown in the following example

```
JOIN FUNC inp -> ....      # JOININST created #
```

when a JOININST is created the input to the function instance, 'inp', is passed to the field 'input', and the function is interrogated to see whether is has function type outputs (the results being placed in the boolean field).

The joinup information from the input to a function instance is passed to the JOININST mode rather than being marked off since the function may turn out not to be used, e.g.

```
JOIN (FUNC inp  input)[2] -> ....
```

In this case if 'inp' contains a function type and it is not joined to elsewhere then a compiler error will occur since the JOININST mode will not have marked off the 'input' field.

4 Additions to the Intermediate Language

There has been very little change to the Intermediate language. Only the following new tokens have been added

TOKEN	IL VALUE
bjoinend	147
tvoid	148
vvoid	149

where 'bjoinend' has been added as a terminator to a JOIN clause, 'tvoid' is a void type and 'vvoid' is a void clause. The token 'bjoinend' has not been needed before since previously a JOIN could only have a very limited right hand side and hence getting a one track syntax was not a problem. By allowing 'unit' like expressions on either side of a JOIN a terminator was needed. Thus the Intermediate Language for the right hand side of a JOIN statement now looks like

```
bjoin val bjoinend
```

where 'bjoin' is a language token and 'val' a value delivering expression.

Another addition to the Intermediate Language has been the outputting of identifiers that need their joins checked in the post assembler phase. This information has been added at the end of a function body. Thus a function body now appears as

```
fnormacstart in in in          # libv_body, fnno, number_names #
             in in in          # stack_size, number_replicators, is_fnset #
             in macro          # number_mac_params, macro_params #
             inid  fnspec      # fn_name, fn_specification #
             fncase            # fn_body_unit #
             in inloop        ## number_checks, checks_list ##
             fnend
```

where 'in', 'inid' represents an integer and identifier, respectively. The addition to the Intermediate Language is shown by the ##....## line.

The function specification has altered slightly to reflect the changes in the function set information. There are three types now put out into the Intermediate Language. That is, the parameter and answer type of the function plus, in the case of function sets, the function type (a null type is put out for non-function sets).. This has been necessary since function set rows would not be correctly held if only the parameter and answer type are known. For example

```
FN FS1 = ( [5]([3]bool->[3]bool):one ):...
```

this function must retain the fact that it is a row 5 of function types and not a function type of [5][3]bool → [5][3]bool. Thus a function specification in the Intermediate Language now looks like

```
kfn inid in  in                              # fn_name, libv spec, is_fnset #
    in in in                                # import, ints_size, types_size #
    in in                                          # consts_size, fns_size #
    extctypes exttypes extints extfns extfns     # External types etc.   #
    type type type                              ## param, ans, FNSETTYPE ##
    in inloop ,                                  # inds_size, inds_list  #
```

where 'fnsettype' is the addition to the Intermediate Language.

In order to allow for multiple makes the Intermediate Language has been changed to the following

```
bmake in multint          ## is_fnset, multiple_ints ##
      type type            # input_type, output_type #
      in in itlist         # is_macro, numb_mac_params, mac_paprams #
      attribute idloop     # attribute_data, make_names #
      zero

multint ::= integer multint,   ## multiple make integers ##
            zero;
```

In the case of a function set the 'output_type' is set to 'tnull', and the 'input_type' takes the type of the function set.

5 Compiler Procedures

There have been a number of new procedures added to module COMPILEPROC because of the joinup work. The main ones are

- SOURCE: Get source of TYPE
- SINK: Get sink of TYPE
- SETIOTYPE: Get IO of TYPE
- JOIN_SETUP: TYPE to JOINTYPE
- CHECK_JOINED: Is mode JOINUP fully joined?
- CHECK_IF_JOINED: Does mode JOINUP have any joined fields?
- HAS_JOIN_CHECK: Does mode JOINUP have any 'jcheck's?
- CHECK_JOINS: Checks that all identifiers have been joined
- IS_FNTYPE_JOINED: Does mode JOINUP have any joined fntypes?
- HAS_FNTYPE: Does mode JOINUP have any fntypes?
- CHECK_IF_OUTPUT_JOINED: has function output already been joined?
- CHECK_OFF: Main procedure for marking joins in mode JOINUP
- JOINTRIM: Trim or index JOINUP
- GETVARMULT: Expand JOINREP to make JOINROW mode
- GETREPLICATE: Used in begin..end within replicator for marking joins

The 'setiotype' is not a null procedure since the IO TYPE depends on the context for determining its source and sink.

In addition to these procedures 'makedecs' has been altered to take account of the new DECIDS mode and is used to make up all the declarations.

6 Compiler Actions

The main changes to the compiler actions have been in the setting up of the JOINUP mode. Basically a new joinup mode is created whenever an identifier is used, this occurs in <findidpr> and <findioidpr>. These actions get their joinup information from the modes which hold the identifier declarations. Action <idlpr> and <ioidlpr> set up the identifiers TYPE information by using procedures 'source' and 'sink'.

A key point is that copies of the joinup mode are NOT taken, instead pointers are used so that when a 'check_off' occurs the pointers are already looking at the identifiers mode, and hence check-off occurs in the identifiers declaration mode.

Action <fn1> has only changed slightly. Where there were three possible choices for 'typeoffn' there are now only two, namely 'normalfntypetok' and 'fnsetfntypetok'. There are now two 'decs' which need to be made up, one for the inputs and one for the outputs. Everything else has remained unchanged.

Action <fn2> has had a large addition to its beginning. This is to do the checking of output names to see whether they have all been joined to, plus the outputting of information on join-checks. The rest has remained with little alteration.

Actions <index>, <trim> and <dynindex2> have had extra pieces added to allow for indexing/trimimg the joinup mode. In addition a JOINVARSTACK is used to stack the index/trimimg information in the case of replicators. This mode is of the form

```
JOINVARSTACK= STRUCT (INTEGER lwb, upb, BOOL index, dynindex,
                                                REF JOINVARSTACK rest)
```

where 'lwb', 'upb' are the range of index/trimimg, 'index' is TRUE for indexing, and 'dynindex' is TRUE for dynamic indexing. This mode is the used by action <joinvar> to create the JOINREP mode.

Action <conc> has been extended to handle the three possible types of concatenation of the JOINUP mode, that is

```
row CONC single_item
single_item CONC row
   row CONC row
```

Action <make1> has been altered for the new way in which multiple makes are passed out to the Intermediate Language.

Actions <checkoutputtype> and <checkoutputscope> have been added to check and mark the joinup mode when coming out of a begin..end clause, where <checkoutputtype> is used when an OUTPUT statement is present and <checkoutputscope> is used when it is not.

So far function types have not been allowed to stray very far into the sequential part of ELLA. LET names can have fntypes but since there is not much that can be done with them in sequences they are of little use. However if in future function types do go into sequences then further work will be needed.

7 Optimisation

Any identifier which does not have a function type in the source position has the joinup mode for the source set to 'jtrue'. Thus a complex structure can be reduced to a simple joinup mode. An identifier with a complex function type specification is only ever split up at its top level source/sink parts. Any further splitting is done when the identifier is passed to another identifier, for example through a LET statement.

The joinup mode does little checking for type mismatch, incorrect sizing etc., all these checks are done by the type checking procedures of the compiler.

8 Conclusion

This document has presented the mechanism by which connectivity information is handled within the ELLA compiler. The implications of this work on the rest of the ELLA system has been discussed. The work described in this document has been completed and a large amount of testing of this code has been successfully carried out. This work will be

incorporated into version 6 of the commercial ELLA product.

9 Acknowledgements

The authors would like to acknowledge the support of the ESPRIT 'SPRITE' project 2260 and the IED Behavioural Synthesis project.

10 References

1. M.G.Hill, E.V.Whiting, J D.Morison, "Bidirectionality, Connectivity and Instantiations in ELLA", RSRE Memorandum No. 4421, 1991.

2. Computer General Electronic Design, Henry Street, Bath, BA1 1JR, UK, The ELLA Language Reference Manual, 1989.

INTENTIONALLY BLANK

A Connectivity Modes for ELLA V4

In this section the join mechanism used in ELLA V4 is outlined, and all comments relate to ELLA V4.

The compiler modes used for the join checks are

```
   DECID = STRUCT (TYPE type, BOOL explicit, INT decno)
  MAKEID = STRUCT (INT decno, INTEGER makeint, TYPE param,ans,
                   JOINED joined)

   JOINED = STRUCT (BOOL check, JOINUNION j)
JOINUNION = UNION (BOOL, REF VECTOR [] BOOL,
                   REF VECTOR [] REF VECTOR [] BOOL)
```

where DECID and MAKEID hold the information about identifiers, and JOINED holds the joinup information. Since ELLA has no function type outputs there is no need to have any joinup information in the DECID mode. It is only the MAKEID mode, which is used for implicit function makes, that needs to have a joined field.

The JOINED mode uses the JOINUNION mode which only has three possible forms and they are used for the following cases

```
                BOOL:   a single MAKE

      REF VECTOR [] BOOL: a multiple MAKE or function row

REF VECTOR [] REF VECTOR [] BOOL: a multiple MAKE of a function row
```

By having these forms of joined fields the following restrictions were imposed on the language.

- A single MAKE must have all of its inputs supplied at the same time.
- A multiple MAKE needs to have its inputs supplied one at a time.
- All output components must be supplied in one OUTPUT statement.
- A Function Set Row can only have a single dimension.
- A multiple MAKE can only have a single dimension.
- LET names cannot be function types.
- Function Sets have to be 'made' before use.

The only other places in the language where joins are done are where 'IO identifier' is present. This can occur as an element in a unitary clause. Consider the following extract of ELLA syntax

```
io_part_of_unit ::= IO ioid

    JOIN unit  -> ioid
```

where 'io_part_of_unit' is only that part of a unitary clause (unit) represented by 'IO identifier', and

```
ioid        ::= name ind1ety <ioid>;     # MAKE name #

ind1ety     ::= [integer] ind2ety,       # multiple MAKE #
                $;

ind2ety     ::= [integer],               # multiple MAKE and FNSET row #
                $;
```

The Action <ioid> does the join marking. Note that the only identifiers that are allowed to be joined in this way are those that have come from a MAKE. IO of an input function type is allowed because it was treated as a MAKE.

If a replicator has a begin..end clause as its replication statements and this begin..end clause contains a JOIN statement, then action <ioid> ensures that the MAKE was local to the begin..end otherwise it faults with "You have already joined to an element here".

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 4462	FEBRUARY	1991

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
ELLA CONDUCTIVITY CONTROL

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	18

Abstract

This document describes the enhancements to the internal data structure of the ELLA compiler for the holding and checking of the connectivity information within an ELLA design. The consequence of this work on the rest of the ELLA system is discussed.

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

Descriptors

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

S80/48

INTENTIONALLY BLANK