North-Holland
Microprocessing and Microprogramming 18 (1986) 567–580

A LANGUAGE FOR BEHAVIOR, STRUCTURE AND GEOMETRY

Gunnar Carlstedt

Hybridlaboratoriet Hylab AB, Träringen 47, S-416 79 Gothenburg, Sweden

A representation for behavior. It is a type of language specialised for synthesis tools. This report also outlines a syntax for documentational purpose.

The representation isolates different language issues as parallelism (spatial), time dependency, alternatives, hiding, rules, conditionals (guards), abstraction and links (datapaths).

A comparison is done with normal imperative, applicative and declarative languages. Different types of language constructs are outlined in the representational form.

Several types of hardware structures are shown. They range from graphics over nets to processors.

INTRODUCTION

This report describes a representation for behavior adapted for machine synthesis and the design of VLSI circuits.

Description of time dependance

Computer languages have been used for a long time. There are several examples of such languages, one is Pascal (JENS75). Most languages have been able to describe sequencies of action. Each step of such a sequence takes zero time. Consequently the sequence cannot describe time. The language is therefore unable to describe behavior. Except for the use of read and write clauses the language describes a mapping of values into an output structure of values.

CCS (MILN80) may define several parallel processes. They perform actions by transporting messages between each other. Because there are no means to describe time delays CCS cannot fully describe a behavior. It may only be used to describe event synchronisation.

Simula (BIRTW73) can describe parallel structures of processes. Each such one may depend on a global clock. The language has been used intensively to simulate real-time behavior.

The Ada language (ADA82) has processes defined that may be structured parallel to records or arrays. Ada can express time-delays. Ada is therefore capable of describing discrete time behavior.

The general behavior needs a possibility to describe waveforms of values. HHDL(HHDL83), VHDL (VHDL85) and VERITAS (HANN85) are the only languages known to be able to do this.

Parallelism

The Pascal language except for read and write clauses cannot describe any parallel behavior. Data structures may be formed into hierarkies of parallel objects. Pascal is therefore only able to describe parallel structure, but not actions.

Both Ada and Simula can describe structures of parallel processes. A process is described by a sequence of statements describing by effect the actions of a process (imperative parallelism).

Ella and CCS can also describe parallel processes. Each process is describe in a functional approach (applicative parallelism).

True hardware consists of (hierarchical) structures of processes, each acting dependent on values of other processes. Such parallelism can in a straight forward manner be modelled by Ada, CCS or Simula.

Synchronisation method

On lowest level hardware may be modelled as a system of non-linear equations. The resultant behavior are currents and voltages describing waveforms.

On register transfer level hardware can be modelled as finite state machines with many

states. The states may correspond to register values and microprogram sequence state.

On a high-level in a system components generally communicate by sending messages. On this level the Ada and CCS can model the behavior.

Most hardware description languages has adopted a synchronising method different from computer languages. They assume the existence of signals, that are time dependant values, i e waveforms. Signals are modelled by functions from values of signals. The function can express some sort of delay mechanism. The delay may be discrete steps. VHDL (VHDL85), ELLA (ELLA85) and Karl (HART77) are using this method.

Both Ada and CCS synchronise by transferring messages. Normal hardware do not synchronise in this way.

In order to be able to model all levels in a design a language must have a semantics that can describe non-linear continuous time behavior, finite state machines with functions controlled by the state and by interchanging messages with certain delays.

Language type

The language shall be used for automatic synthesis. All programming by "effect" should be avoided. The synthesis tool often has difficulty in modelling such actions into a manageable action.

Applicative languages are well suited for sythesis. The Ella language (ELLA85) is an applicative language for RTL level behavior. Other applicative languages are Sticks and stones adapted for topology.

Declarative languages like PROLOG (CLOCK81) may probably be synthesised easily. The description power of such languages are high. They generally by rules describe the properties of the result, but not the result.

The synthesis is used for finding the result. It is therefore no meaning to synthesise such programs. Instead it should probably be possible to introduce supplementary rules describing adequate properties in order to get comparative results.

The language should be of functional type, where all functions are performed in parallel, in order to make synthesis simple.

Abstraction mechanism

Common languages use types and subroutines to hide properties or implementation. They contain means to abstract behavior / value.

There is a need of abstraction in order to build large good systems. This abstraction shall not be mixed up with the abstract behavioral mechanism of a computer.

There are several types of abstractions in the hardware, e g:

RANGES
: use of a "0" when voltage less than 0.8 V, The language shall abstract a real range to a symbol. In this case true or false.

NUMBERS
: bitcombination "1101" corresponds to the value 13. Functions shall be used to map a number into an implementation. In this case a function "binary_code" shall construct a bit-vector of ones and zeroes.

ACTIONS
: the instruction "add" corresponds to several actions as reading two registers, transfering values to an ALU, adding in the ALU, storing the result in a register, and advancing to next instruction. A symbol shall abstract a behavior. In this case the add instruction code abstracts several microsteps of action.

DYNAMIC CHANGE OF ABSTRACTION
: an instruction set is an abstraction from code to behavior. A processor may load a new instruction set dynamically and then start to execute it. Therefore the language must be able to dynamically change abstraction.

Some of these abstraction mechanisms are generally found in compiler-compilers. Denotation semantics GORD79b do also describe such abstractions.

In order to describe these abstractions the language must be able to dynamically alter an abstraction mechanism. The abstraction mechanism shall also be able to map patterns of symbols (codes) into behavior.

Existence in space

Several types of objects need a description of the existence of an object and the behavior within the limits of the object.

Typical such objects are graphic objects. They exist in a certain environment, the space. Each object exists within a certain volume where each

point may have a behavior. The behavior may be colour, density, material, potential, current, speed etc.

The space may be frequency, room etc.

Predicate logic have some similarities with this type of abstraction.

The language must be able to separate the notion of existence and the behavior for parts of the object.

Description of design objects

The design object may be described from mask-pattern to system behavior. The latter may be a man-machine interface.

The EDIF (EDIF85) format is a language for a medium to be interchanged between different design-tool users. It is by mode switching capable describing both structure, geometry, nets and behavior.

It is our belief that there really is no difference between these type of design objects.

The generally used meaning by "structure" and "behavior" is in our world the same. A structure is a behavior that is implementable. The generally used structure does not have any behavior but is bound to a set of parallel objects occupying area on silicon. In our opinion the general behavior may also be bound to some type of silicon area.

With this new type of structural object the behavior of the object is known Such structural object must be described by some other similar type of implementable behavior (object). The lowest level of objects are probably gates or wires and transistors.

This new type of structural object defines a graphic layout consisting of a mask pattern specifying the existence of wires and transistors. Each wire or transistor has a behavior changing currents and voltages.

There are several "aspects" of such a behavior: area, power consumption, fabrication mask level etc. Thus the concept behavior must contain a mechanism to describe such type of aspects.

Syntax

The purpose of this report is to outline a language capable of describing most levels of design objects.

In the following paragraphs a language is described. It consists of firm semantics. The language may be depicted by a hardware description language for human beings and as a data structure for a computer.

It is not the purpose of this report to find an optimal man-machine interface. Thus the syntax is just scetched. The description occurs only for the purpose of making communication with the reader simple.

SEMANTICS

The semantics describes behavior. Behavior is synonymous to process and activity. In this report the word behavior will be used.

A behavior is said to be time discrete if it exhibits itself only a certain infinitesimal points in time. The behavior is time continuous if it is exhibited all the time.

The behavior can exist between two points in time. If the time interval is zero, then the behavior is a pulse behavior. If the interval is finite then the behavior is terminable.

The elementary behavior is a behavior which does not show any outward signs of change, but still exists.

At every moment in time, the behavior shows a value which is an infinitesimally small part of a course of events. A behavior is constant if its value is unchanged during the time. Otherwise the behavior is variable.

The different issues of behavior may be discrete or continuous. The discrete types consists of quantified behaviors. The continuous types consists of an infinit infinitesimal small behaviors that together specifies a "linear" range of behaviors, compare points and a line.

The behavior as such may be controlled by the followning semantic issues:

Parallelism, (Structure)

```
par(b1, b2, ..., bn)
par_cont
```

Several separate behaviors b1 to bn are configurated to one new compound behavior. The external view of this behavior is ONE UNIT containing SEVERAL parallel discrete behaviors. A special continuous structure of parallel behavior is described further down.

Proposed syntax is:

```
b1 b2 ... bn
```

The syntax consists av a list of behaviors separated by blanks. A list may be enclosed

within parenthesis in order to describe one unit.

Time dependency, (Temporal)

```
seq(b1, b2, ..., bn)
seq_cont
```

Several terminable behaviors follow each other in the order of the list b1 .. bn. A new behavior starts when the preceeding behavior finishes. The external view of this behavior is ONE behavior that is CHANGED by time to new behaviors. A special continuous sequence of behavior is described further down.

Proposed syntax is:

```
b1 ; b2 ; ... ; bn ;
```

The syntax consists of behaviors separated by semicolons. A list may be enclosed within parenthesis in order to describe one unit.

Alternatives, (Sets)

```
alt(b1, b2, ... , bn)
alt_cont
```

Several behaviors are alternatives to a behavior. The external view of this behavior is ONE behavior, that may take the appearance of several ALTERNATIVE behaviors. A special continuous set of alternative behaviors is described further down.

Proposed syntax is:

```
b1 , b2 , ... , bn
```

The syntax consists of containing behaviors separated by commas. A list may be enclosed within parenthesis in order to describe one unit.

Hiding parts

```
hide(bh, bv)
```

A behavior consists of a hidden behavior bl and a visible behavior bv. The hidden behavior bh is hidden. An external view of this behavior is only the VISIBLE behavior bv.

Proposed syntax is:

```
bl1 ... bln ! bo1 .. bom
```

The syntax consists of two lists first, the local behavior and the visible behavior, separated by the sign !. The construct may be enclosed within parenthesis in order to describe one unit.

Rule (Lambda expressions)

```
rule(bp,bs)
```

A rule is a pair of pattern and substitute behavior.

The rule is applied to an actual behavior ba. If the value of the actual behavior ba and a pattern behavior bp are equal the resultant behavior is the substitute behavior bs. The value of the pattern behavior when substituted is a formal behavior bf.

The equality depends on the structure of the two behaviours ba and bp:

elementary elementary
: The two values are equal. The formal behavior equals the actual behavior.

elementary parallel
: or vice versa. The two values are not equal.

parallel parallel
: The two values are equal if each corresponding subelement are equal. The formal behavior equals the actual behavior.

parallel alternative
: or vice versa. For each alternative the the rule may be assumed to to be duplicated in an alternative behavior form:

```
rule((bp1,bp2),bs)

equals

alt(rule(bp1,bs),
    rule(bp2,bs))
```

general hiding
: The equality corresponds to the equality where the visible part of the hiding construct replaces the hiding construct.

general rule
: The equality corresponds to the equality where the pattern behavior of the rule replaces the rule. The formal behavior equals the substitute behavior of the rule.

general behavior
: The "general" and the "behavior" are two behaviors, not values. The two are equal if the last part of the "general" equals the "behavior" for each point in time. The formal behavior equals the "behavior" behavior. This is generally the case only just when equality occurs.

The rules may be included into structures as parallel, sequence and alternative behaviors.

parallel rule matches an equally parallel structured actual behavior. The elements of the actual behavior must then equal the corresponding pattern behavior. The formal pattern value is substituted at each element.

In the sequence of rules and the alternative of rules the external behavior equals the behavior when the actual behavior is moved and duplicated to all the elements:

```
ba (r1, ... , rN)
ba (r1; ... ; rN)
```

equals:

```
(ba r1,  .. , ba rN)
(ba r1; ... , ba rN)
```

Proposed syntax is:

```
bp -> bs
```

The syntax consists of the pattern behaviors an arrow and the substitute behavior.

Existence of behavior, (Guard)

```
guard(ba,r)
```

The guard consists of an abstract behavior ba and a rule. The rule may be structured as parallel, sequence or alternative rule.

If the value of behavior ba and a pattern behavior bp of the rule are equal the resultant behavior is the substitute behavior bs. If several of the alternatives substitute, the result is a list of alternative behaviors. Otherwise no external behavior is existent.

The external view of this behavior is an ALTERNATIVE behavior SUBSTITUTED for the abstract behavior ba or NO behavior at all.

When a substitution takes place the construct is non-cutting, otherwise cutting.

Proposed syntax is:

```
ba r
```

or with alternative rules:

```
ba (r1,
    ...
    rN)
```

The syntax consists of the abstract and the rule behaviors. The construct may be enclosed within parenthesis in order to describe one unit.

Abstraction, (Reduction)

```
abstract(bd, ba)
```

An abstract behavior contains rules bd and abstract behavior patterns ba.

The abstract behavior pattern may contain nested behaviors eg parallel, sequence, alternative, hidden, and guard behavior type. For each element or list of elements of them a recursive replacement occurs.

Each such element, part of list or full list is an actual abstraction of a behavior.

The definition may be an unstructured rule or a structured rule. The definition may therefore define alternatives of rules. Assume that each alternative is one guard where the pattern and substitute have taken their corresponding places and the actual abstraction is used as abstract behavior.

If at least one of the alternatives is NOT CUT the substitution takes place. Many alternatives may be created. If CUT the actual abstraction remains.

The external view of the abstraction is a SUBSTITUTION of by rules defined behaviors.

Proposed syntax is:

```
def r
ba
```

or with alternative rules:

```
def r1,
    r2,
    ...
    rN
ba
```

The syntax consists of the symbol "def", rule/s and a substitute behavior. Optionally each rule may be preceeded by the symbol "def". If so abstract behavior patterns and rules may be mixed. The construct may be enclosed within parenthesis in order to describe one unit.

LINKS

```
bind(b, symb)
 ...
 .. use(symb) ..
```

A symbol symb is bound to a behavior b. The external view of the binding construct is the behavior b.

A symbol is a behavior. Generally it is static and just a sequence of characters.

```
´adam
"a row"
´(bp -> bs)
```

corresponds to

```
(a d a m)
(a _ r o w)
rule(bp,bs)
```

The symbol _ marks just here the space symbol. The rule is converted to a structure containing the rule designator and the pattern and substitute behavior.

OPERATORS

Operators are rules. Some of these are predefined.

Several classes of operators have one operator working on either of the constructs parallel, sequence or alternative. These types of operators generally alters the structure of the construct.

COMPARISON WITH OTHER LANGUAGES

Languages

The language describes syntax of normal languages in a stright forward method. Each language category is represented by a rule. The following rules are translation from the Pascal manual:

```
def int -> ... ,

def int:l ´<> int:r -> ... ,
    real:l ´<> real:r -> ... ,
    bool:l ´<> bool:r -> ... ,
    ...

def rel_op ->
  (´= , ´<> , ´< , ´<= ,
   ´>= , ´> , ´in)

def simpl_expr -> ...

def exp -> (
   simpl_exp:l -> l,
   simpl_exp:l rel_op:op simpl_exp:r
     -> T op r )
```

The "expression" category is defined in one definition containing two rules. The first rule specifies only a single simple "expression". The other rules contains two simple expressions separated by a relational operator.

Now consider the last rule. The pattern is "simpl_exp" "rel_op" "simpl_exp". The value of them ar bound by T, op and r respectively. The substitute is l op r.

The pattern is described in an abstract form. The simple_exp and rel_op is described elsewhere.

The category rel_op is defined as a pattern consisting of the alternative containing all alternative symbols = to in. The rule only defines the set of symbol.

There is a definition containing all rules for relational operators. They consist of two type patterns separated by a relational operator. The semantics, i e the substitute behavior, is not shown here.

Protocols

There is no big difference between a protocol and a language. On a channel where the protocol is transfered the sequence of symbols transferred are described by a language. In a computer language the symbols are in parallel but in a protocol in a sequence. In a language there is really no producer of symbols. In a protocol either of two parties sends the symbol.

The following rules are part of the CCITT s.62 standard:

```
def party id:pid (any:w) -> ...

def s62 -> ...

( party A ch ) :A
( party B ch ) :B
( s62 A B ) :ch
```

The description of a protocol contains two behaviors A and B, the two parties, and the activity s62 on the channel. Each party listen to the channel ch and contains the state for the party. The channel reads the parties A and B.

The channel shows a parallel structure containing the A and B party change of state.

Each party checks whether update shall occur or not. If so it updates the state in the party.

The channel may be defined as simplex or duplex. Let us now consider a simplex channel.

```
def s62 any:a any:b ->
    ((css; (rssp; session; cse; rsep),
           rssn) a b)

def css any:a any:b ->
      ((s_css a):ch (r_css ch b)),
    rssn ->
      ((r_rssn ch a) (s_rssn b):ch),
    rssp ->
      ((r_rssp ch a) (s_rssp b ):ch),
    cse ->
      ((s_cse a):ch (r_cse ch b)),
```

```
    rsep ->
      ((r_rsep ch a) (s_rsep b):ch)

def r_css party:b ->
      ( scan_cmd co_css (
          scan_pg co_sr (
            (scan_p co_tid ):tid ;
            (scan_p co_dat ):dat ;
            (scan_p co_asrn ):asrn );
          (scan_p co_sid ):sid ;
          (scan_pg co_nbtc (
            (scan_p co_grchs ):grchs ;
            (scan_p co_cchs ):cchs ;
            (scan_p co_pfmt ):pfmt ;
            (scan_p co_mtc ):mtc ))
        -> ...) ,

    r_rssn ... ,
    ...
    r_rsep ...

def s_css party:a ->
      ( fmt_cmd co_css (
          fmt_pg co_sr (
            fmt_p co_tid a.tid ;
            fmt_p co_dat a.dat ;
            fmt_p co_asrn a.asrn );
          fmt_p co_sid a.sid ;
          fmt_pg co_nbtc (
            fmt_p co_grchs a.grchs ;
            fmt_p co_cchs a.cchs ;
            fmt_p co_pfmt a.pfmt ;
            fmt_p co_mtc a.mtc ))) ,

     s_rssn ... ,
     ...
     s_rsep ...

def fmt_cmd byte:c any:p ->
      (c (size p) p)
    fmt_p ... ,
    fmt_pg ...

def co_css -> 13 ,
    co_sr -> 1 ,
    ...
    co_mtc -> 75
```

The first definition defines the s62 protocol. The s62 protocol is simplified but the main parts are shown. The s62 protocol consists of two variants, one with negative acknowledge and one with positive. All message are in a sequence. The messages are abstract.

The second definition contains the semantic definition of messages. Each message definition is a parallel behavior of a transmitter and receiver. They are connected by a link "ch" corresponding to the channel. The s_... is the transmitter and the r... the receiver.

The third and forth definition contains all specification of formats for individual commands and responses. The third rule is shown as the receivers action. The fourth rule is the senders action. The css command is shown.

The command is a structured sequence of bytes. They are formed by nested format constructors. Each such forms a sequence of bytes. All constructors form a sequence.

The fifth definition contains the three definitions of coding commands / responses, parameters and group parameters. The rules are not fully defined, because those details are not considered here.

The sixth definition contains all definition of codes for commands, responses and parameters. Each symbol is replaced with a code.

Statements

Statements are a sequence of actions. Typical Pascal and Ada statements are sequencies of such actions separated by ";". A sequence of S1 ; .. ; Sn statements is:

```
S1 ; S2 ; ... ; Sn
```

Type definitions

The type definition i Pascal and Ada specifies all possible values a variable ocr expression may take.

The type information is used as pattern behaviors in implementation of languages. The Pascal integer is defined as:

```
def type_id ->
      (´integer, ´boolean,
       ´real, ´record, ´array)

def int:v ->
      ((+ , -) nat)
```

An integer consists of a sign, +/-, and a natural number. It is a discrete alternative specified as a basic alternative.

Variables

In imperative languages there are variables. Each variable is a parallel process. There is also one central protocol, the statements, controlling the variables:

```
def var type_id:type id:ad ->
   ( ad := type:v -> v,
     (not id := type) -> v_ad ) :v_ad

(cntrl (var integer ´id1 )): id1
(cntrl (var integer ´id2 )): id2
```

```
(... ; name := expr ; ...):cntrl
```

The variable is defined as a process with feedback. The rule for a variable has as parameter the variable type and identifier.

The variable is a process listening to the control. When it shows an identifier, an assign symbol and a correct value the process takes the value from the parameter. In other cases the value is kept by a feedback.

To each variable process and identifier is bound. They may be used at several placess in the control sequence.

The variable definition is schematic because a master slave function must be included in order to allow the update of a variable that also is read.

When there are several activation records, the identifier for a variable has to be supplemented by an activation record identifier.

Conditionals

The generally used case-clause can be described as:

```
expr (pat1 -> S1,
      pat2 -> S2,
      ...
      patN -> SN)
```

There is one expression expr calculating the control variable. There are N rules defined. Each rule contains the choser pat and the statement S. The expr select the alternatives, zero, one or more if patterns overlap.

The if-clause is implemented as:

```
expr:cond true -> Strue,
     cond false -> Sfalse
```

Two symbols true and false are defined. A boolean expression expr is a behavior and bound to the symbol cond. A guard choses either the of the statements Strue or Sfalse.

Function calls

A function is an abstraction in Pascal, Ada and Ella. It is implemented as:

```
def f_id ptype1:p1 ... ->
      (.... ! expr )


   ... ; f_id expr ... expr ; ...
```

The function is defined by a rule. The pattern behavior consists of the function identifier followed by the type pattern behaviors for the parameters. Each parameters has an identifier bound to its value.

The substitute behavior consists of a hiding constructs. The visible part is the result expression.

The procedure call is more complex, because it has the ability to write in imported variables.

Lamda functions

The lambda functions are available in Lambda Calculus and in most functional languages. The guard and the lambda function performs approximately the same semantics. The guard uses a pattern matching but the lambda function uses direct replacement.

The following lambda expressions:

```
lambda x.f ap
lambda x.y.f(x,y) apx apy
```

are implemented as:

```
ap any:x -> bs
apy apx any:x -> (any:y -> f x y)
```

The last example is an high-order function. The example shows two curried operands. Each function is implemented as a rule. The any behavior is an alternative stating any type of structure.

Inference - Prolog

The reduction mechanism is directly applicable to implement Prolog. Thus the following Prolog program:

```
rule1(lit,v) :- pred1(v), pred2(v)
ruel2...

goal(lit,v)
```

is implemented as:

```
def rule1 lit:v1 t_v:v2 ->
      (pred1 v) any ->
      (pred2 v) any -> (v1,v2)
    rule2...

goal lit any:v
```

The predicate is implemented as a guard with the pattern equal to any ("true"). Several nested guards form a sequence of predicates. Parameters are bound in the conventional way.

In the goal each variable is substituted by the any behavior. The reduction mechanism will reduce this behavior to cases that are implemented. The result consisting of (lit,v), where lit is the specified literal and v is the searched variable. The result may consist of several alternatives.

Message passing

Message passing occurs in the Ada rendezvous, in CCS and CSP. They are in their basic function similar but from user point of view different.

Such systems are finite-state machines and may be implemented as:

```
(process1 p2 ):p1
(process2 p1 ):p2
 ...
(process2 pm ):pN

def proc_id m_types:m ->
      ((state m ):state
      ! transmit)

def state1:s m_types:m ->
     (m (pat1 -> S1,
         pat2 -> S2,
         ...
         patK -> SK,
         else -> s),

    state2:s ...
```

The state machine consists of N processes. They may read other processes.

Each process rule has a pattern behavior describing the process id and the message channel m.

The substitute behavior constitutes the behavior of the process. It is a hidden construct. The visible part are the messages transmitted. The hidden part contains the states.

The different states are described by an own rule. It consists of a guard with a set of rules, one for each possible message and one for none and all other messages. The last guard contains a feedback, that preserves the current state.

Wave-form

In several cases a waveform is needed. An oscillator with frequency 1/2*tph is:

```
def osc real:tph ->
       ((0 o> tph ; 1 o> tph ) ;
        osc tph )
```

The operator o> keeps a value constant over a time. The use of operators are essential when designing waveforms. Im portant operators are the apply and the value to sequence operators.

DESCRIPTION OF DIFFERENT OBJECTS, EXAMPLES

Processors and controllers

Below is a very simple processor shown:

```
(... ; sub_a ; ...): program
def sub_a -> (... ; op_b ; ...)

def op_a reg:p1 -> (micro cntrl_opa, p1),
    op_b -> (micro cntrl_opb ())

def micro rac:c reg:p1 ->
    (c (cntrl_opa ->
           (aTu p1 (c11p1 c12p2)),
        cntrl_opb ->
           (aTu () (c11reg1 c12clock)))

def alu reg:p1 (ra1:c_u1p1 ra2:c_u1p2) ->
    ((adder c_u1p1 (c11p1 -> p1,
                    c11reg1 -> reg1)
            c_u1p2 (c12u2 -> u2,
                    c12clock -> clock)):u1
     (adder reg1 clock):u2
     (ram 64 bit16):reg1
     (...):clock
     ! u1)

def ra1 -> (c11p1, c11reg1),
    ra2 -> (c12u2, c12clock),
    rac -> (cntrl_opa, cntrl_opb),
    reg -> (0 .. 65535)

def adder reg:p1 reg:p1 -> ...
```

The first construct is a program. It is a sequence of statements. A subroutine call is shown as one step.

The first definition specifies the called subroutine. The subroutine contains a sequence of operators. The operator op_b is shown.

The second definition contains definitions for operators. Two operators, op_a and op_b, are shown. The first operator has a parameter, the other none. Each operator is specified by one microinstruction. It has two parameter fields, one for a control code and one parameter.

The third definition specifies the microinstruction. It has as earlier mentioned two parameters. The microinstruction uses the control-field c to select among two different alu operations. The alu is controlled by one literl parameter and two control symbols.

The fourth definition specifies the alu. It has two parameters as mentioned before. The alu consists of two "hardw" units u1 and u2, unit reg1 and clock. All units except u1 are hidden.

The unit u1 is a hardw unit having two parameters. By a guard a multiplexing structure is used for both of them.

The fifth definition specifies the range of four types ra1, ra2, rac and reg. The last type is specified to contains a 65536 different states.

The sixth definition specifies the pure physical unit for the alu. It is not fully shown here.

ALU:s

An arithmetic unit may be parametrised in order to perform different operations. This switchable function is specified further down in the description of a processor.

The arithmetic unit uses a code for the operands and performs a mathematic function. Both have to be specified:

```
def binary nat:le ->
     (nat:v ->
        (0..le):bit *
        (v mod (2 ** bit)))

def add nat:l nat:r -> l + r
```

and an implementation:

```
def add_binary nat:bit
     (binary bit):l (binary bit)y:r ->
       (0..le):bit *
       (half_add l.bit r.bit
          (carry (bit-1):lec
           l.(0..lec) r.(0..lec)))

def half_add bool:l bool:r bool:c ->
       (l r c) ((0 0 0) -> 0,
                (0 0 1) -> 1,
                (0 1 0) -> 1,
                (0 1 1) -> 0,
                (1 0 0) -> 1,
                (1 0 1) -> 0,
                (1 1 0) -> 0,
                (1 1 1) -> 1)

def carry nat:bit
     (binary bit):l (binary bit)y:r ->
     ((l.bit r.bit)
      ((0 0) -> 0,
       ((0 1), (1 0)) ->
            (carry (bit-1):lec
             l.(0..lec) (r.(0..lec))
       (1 1) -> 1))
```

The add operation is specified in the second definition. It is a general description for all positive values.

The first definition specifies how the binary code is generated. By use of a special high-order function this definition may be converted into a add_binary definition.

The third definition specifies binary add of bit long words. An alternative containing 0 to le is specified. Each alternative corresponds to one bit of the binary word. The apply operator * converts this alternative to a parallel structore containing a half adder. It has as parameters the corresponding bit from the l and r operand and a carry calculated by all less significiant bits.

The half adder uses three boolean values. According to a guard containing 8 rules a result consisting of 0 or 1 is substituted.

The fifth definition specifies the carry. It has three parameters, a word length bit and two words l and r of this length. A guard selects three cases. Two of them substitutes 0 or 1. The third case substitutes a carry calculated from the least significiant bits.

This adder is not optimal, because it contains a carry chain for each bit. Simple modifications removes this problem.

Memories

A conventional read-write random access memory is defined as:

```
def cellop -> (nop, read, write)

def cell type:ty nat:addec ->
       (cellop:rw nat:ad ty:di -> (
          (rw (nop -> vout,
               read -> vout:do,
               write ->
                  ad addec -> di,
                  ad (<> addec) -> vout)
          ):vout
        ! ad addec -> do))

def ram nat:si type:ty ->
       ((0 .. si):ad * (cell ty ad))

(op address di) (ram 1024 bit16 )
```

The first definition specifies the operations for a memory.

The second definition specifies a memory cell. The cell is characterised by two parameters, the first ty specifies the type of the stored word, the second specifies the cell address.

The cell is a rule for a guard. The cell is controlled by three parameters, the

celloperation rw, the address ad and the data input di.

The celloperator rw specifies that the cell output is stable by a feedback. During the write operation the cell is written if adress ad equals the celladdress addec. Otherwise the cell is stable.

All the mentioned behavior is hidden. When the adress ad equals the cell address addec the output of the cell equals do. It is only defined during the read operation.

The ram array is specified by giving the size si and the type ty for the memory cell. An alternative structure containing the values 0 to si is defined. An apply operator * uses each alternative value and replaces it with an abstract cell of type ty and an address equal to the value. The apply operator alters the structure from alternative to parallel structure.

Gate-level, NAND, NOR etc

All simple gates are easily described as:

```
def s -> (0, 1)

def nor3 s:a s:b s:c ->
     ((a b c) ((0 0 0) -> 0,
               (1 s s) -> 1,
               (s 1 s) -> 1,
               (s s 1) -> 1))

def half_add_gate s:l s:r s:c ->
  nor4 nor3 (not l):lb (not r):rb c
       nor3 lb r (not c):cb
       nor3 l rb cb
       nor3 l r c
```

The "def s" defines an alternative containing the possible signals.

The "nor3" defines a nor gate. The description is a simple guard containing a rule for each case. The "s" is used for alternative 0 or 1, ie don't care.

The "half_adder_gate" is an implementation of a halfadder. It contains 3 not and 4 nor3 and 1 nor4 behaviors. All are implemented as gates.

The above shown description of the halfadder can be altered to separately specify all gates and then connect them by links:

```
def half_add_gate s:l s:r s:c ->
  ((nor4 w1 w2 w3 w4):wfu
     (nor3 wlb wrb c): w1
        (not l):wlb
        (not r):wrb
     (nor3 lb r wcb):w2
        (not c):wcb
     (nor3 l rb cb):w3
     (nor3 l r c):w4
  ! wfu)
```

Extra links denotated by wx have been added. The substitute has to be hidden in order to specify the same function.

Nets and topology

The logic abstraction is a functional abstraction, where the result is a substitution of the parameters.

This description can be turn into an equivalent description where all connections, input as well as output, are connected to net components. Each operator is translated to an element of a parallel behavior:

```
def half_add_unit
  (s:l s:r s:c s:fu) ->
  (nor4unit wl1 w21 w31 w41 wfu
     nor3unit wl1 wr1 wc1 wl2
       notunit wl2 wlb1
       notunit wr2 wrb1
     nor3unit wlb2 wr3 wcb1 w22
       notunit wc2 wcb2
     nor3unit wl3 wrb2 wcb3 w32
     nor3unit wl4 wr4 wc3 w42

   net (wl1 wl2 wl3 wl4 l)
   net (wr1 wr2 wr3 wr4 r)
   net (wc1 wc2 wc3 wc4 c)
   net (wfu fu)
   net (wlb1 wlb2)
   net (wrb1 wrb2)
   net (wcb1 wcb2 wcb3)
   net (wl1 wl2)
   net (w21 w22)
   net (w31 w32)
   net (w41 w42)

   connect:wl1
   connect:wl2
   ...
   connect:w42)

def connect -> (0, 1)
```

The topology consists of components, nets and connections. The connection may take only logical values.

Transistor and wire

The abstraction mechanism specifies new levels that are independent of earlier levels. This fact is very important to consider when discribing nets.

Generally all components do depend on other components. Because of this each component cannot

be considered alone. Larger portion of a net structure must therefore be flattened out into one level.

On each such level a component is described by linear or non-linear equations. All components together define a system of equations. By solving this system for voltages and currents the behavior on the conductors can be evaluated.

```
def half_add_unit
  (v:vcc v:gnd s:l s:r s:c s:fu) ->
  (nor4unit v1 g1 wl1 w21 w31 w41 wfu
     nor3unit v2 g2 wl1 wr1 wc1 w12
       notunit v3 g3 wl2 wlb1
       notunit v4 g4 wr2 wrb1
     nor3unit v5 g5 wlb2 wr3 wcb1 w22
       notunit v6 g6 wc2 wcb2
     nor3unit v7 g7 wl3 wrb2 wcb3 w32
     nor3unit v8 g8 wl4 wr4 wc3 w42

   wire C1 (wl1 wl2 wl3 wl4 l)
   wire C2 (wr1 wr2 wr3 wr4 r)
   wire C3 (wc1 wc2 wc3 wc4 c)
   wire C4 (wfu fu)
   wire C5 (wlb1 wlb2)
   wire C6 (wrb1 wrb2)
   wire C7 (wcb1 wcb2 wcb3)
   wire C8 (wl1 wl2)
   wire C9 (w21 w22)
   wire C11 (w31 w32)
   wire C12 (w41 w42)
   power (v1 v2 v3 v4 v5 v6 v7 v8
          vcc)
   power (g1 g2 g3 g4 g5 g6 g7 g8
          gnd)

   plug:wl1
   plug:wl2
   ...
   plug:w42)

def plug -> (0.5..2.4 0.0..0.001)
```

A network consists of components, wires and plug. A plug may take any voltage and current within the capability of the circuits.

Wires are special components having several components connected. All their terminals have the same potential and the sum of their currents is zero. Wires do have a capacitance.

Components have other characteristics.

Geometric object

Geometric objects are described by guards. The guard uses a pattern checking the coordinates to be within certain boarders. For points inside the border the value of the geometric objects is specially defined:

```
def geom_obj ->
        (p:pos border -> (point pos)
```

The border behavior is a static continuous alternative behavior. It contains all points inside the border of the geom_obj.

The border object is generally built by constructors. Such ones may be ABUT, VECTOR, TURN, MIRROR, SCALE, PRIORITY, POLYGONE, COORDINATE, ARC, PAD, WIRE, BRANCH, END or CUT. There may be several systems of constructors. It is not the purpose of this report to define them.

The geometry of the halfadder is:

```
def half_add_geom
  (v:vcc v:gnd s:l s:r s:c s:fu) ->
((x y)  nor4geom v1 g1 pl1 p21 p31
                 p41 pfu
 (x y)    nor3geom v2 g2 pl1 pr1
                 pc1 p12
 (x y)      notgeom v3 g3 pl2 plb1
 (x y)      notgeom v4 g4 pr2 prb1
 (x y)    nor3geom v5 g5 plb2 pr3
                 pcb1 p22
 (x y)      notgeom v6 g6 pc2 pcb2
 (x y)    nor3geom v7 g7 pl3 prb2
                 pcb3 p32
 (x y)    nor3geom v8 g8 pl4 pr4
                 pc3 p42

        wire me (pl1 pl2 pl3 pl4 l)
        wire me (pr1 pr2 pr3 pr4 r)
        wire me (pc1 pc2 pc3 pc4 c)
        wire me (pfu fu)
        wire me (plb1 plb2)
        wire me (prb1 prb2)
        wire me (pcb1 pcb2 pcb3)
        wire me (p11 p12)
        wire me (p21 p22)
        wire me (p31 p32)
        wire me (p41 p42)
        power (vcc1 vcc2 vcc3 vcc4
               vcc5 vcc6 vcc7 vcc8
               vcc)
        power (gnd1 gnd2 gnd3 gnd4
               gnd5 gnd6 gnd7 gnd8
               gnd)

        position:wl1
        position:wl2
        ...
        position:w42)

def position -> (xrange yrange)
```

A geometry consists of components, wires and positions.. Power is a special wire. The position may take arbitrary coordinate value.

Wires are special components having several components connected in points p. Wires are implemented in a certain layer (me).

Components have a size and interface specifying coordinates for connections. Components are placed in certain positions (x y).

A wire may be accurately described by eg:

```
wire width
     (line l1
      branchT2 (line l2 wireend)
               (line l3 cut wireend)
```

CONCLUSION

A language has been specified. It has a simple and strong semantics. Two types of syntaxes has been shown, one for computers and one for human beeings. The last one is just scetched in order to simplify communication with the reader.

The main target for the language is as a representation for design objects. Especially such a simple and strong semantics is necessary for automatic synthesis.

The strength of the language has been shown by examples. All important levels of a hardware design has been shown by examples, ie the full processor function, ALU and memories, topology and geometry. Important parts of imperative, functional and declarative languages has been shown written in the language.

The language will be named "H".

REFERENCES

ADA82
"Reference manual for the Ada Programming language", United States Department of Defense, 1982.

BIRT73
Graham Birtwistle et al, "Simula BEGIN", Auerbach, 1973.

BJOR78
Dines Björner, Cliff Jones, "The Vienna development method: The meta-language", LNCS 61, Springer-Verlag, 1978.

CARL84
Gunnar Carlstedt, "Proposal for the NMP design-system -8401, that compiles functions described in a high-level language to hardware and VLSI-masks", Hybridlaboratoriet Hylab AB, 1984.

CLOCK81
William Clocksin, Christopher Mellish, "Programming in Prolog", Springer-Verlag, 1981.

EDIF85
"EDIF. Electronic Design Interchange Format. Version 100", 1985.

ELLA85
"The ELLA language reference manual", Praxis Systems Ltd, England, 1985.

GOLD83
Adele Goldberg, David Robson, "Smalltalk-80, The Language and its implementation", Addison-Wesley, 1983.

GORD79a
Michael Gordon, "The denotational description of programming languages", Springer-Verlag, 1979.

GORD79b
Michael Gordon, Arthur Milner, Christopher Wadsworth, "Edingburgh LCF", LNCS 78, Springer-Verlag, 1979.

HAIL82
Brent Hailpern, "Verifying concurrent processes using temporal logic", LNCS no 129, Springer-Verlag, 1982.

HANN85
F Keith Hanna, N Daeche, "Specification and verification using high-order logic", Proceedings Computer Hardware Description Languages and their Applications, 1985, pp 418-433.

HART77
Reiner Hartensten, "Fundamentals of structured hardware design", North Holland, 1977.

HERB85
John Herbert, "The application of formal specification and verification to a hardware design", Proceedings Computer Hardware Description Languages and their Applications, 1985, pp 434-451.

HHDL83
"HHDL Language Reference Manual", in Helix Reference Manual, 1983.

JENS75
Kathleen Jensen, Niklaus Wirth, "Pascal user manual and report", Springer-Verlag, 1975.

MILN80
Robin Milner, "A calculus of communicating systems", LNCS no 92, Springer Verlag, 1980.

MILN85
George J Milne, "Simulation and verification: related techniques for hardware analysis", Proceedings Computer Hardware

Description Languages and their Applications, 1985, pp 404-417.

PIL083
R Piloty et al, "CONLAN Report", LNCS no 151, Springer Verlag, 1983.

VHDL85
"VHDL Reference Manual", Texas Instruments, IBM, .