

# V-CHARTS: A Visual Formalism for VHDL

David E.W. Mercer and Douglas R. Smith  
Department of Electrical and Computer Engineering  
Royal Military College of Canada  
Kingston, Ontario, Canada, K7K 5L0

## ABSTRACT

This paper presents a graphical methodology for developing and documenting hierarchical VHDL models at all levels of abstraction that include both behavioural and structural elements. The methodology is a modification and refinement of Buhr diagrams developed by Dr. R.J.H. Buhr of Carleton University for describing concurrency in Ada. Because there is a well defined transform between the diagrammatic elements of this methodology and VHDL code, we believe it would lend itself well to automated means of generating VHDL from diagrams and diagrams from VHDL.

## INTRODUCTION

VHDL is Ada like in its syntax and in that it supports; concurrency, separate compilation, generics (albeit not to the extent Ada does), and interprocess communication. It also possesses some features that particularly suit the description of electronic systems. One of the main features of the language is its ability to integrate both structural and behavioural elements in a single model. Furthermore, different architectures and configurations of the same entity may be simultaneously instantiated into a model.

In order to instantiate a VHDL entity, the designer must have knowledge of the entities interface. This includes its generic and port maps of which the name, position, type, and kind of the elements must be known. Also, the designer must have knowledge of the architectures and configurations available for the entity and must be able to determine which is appropriate. Previously, this has demanded that a designer search code and read the comments that accompany entities. We feel that a pictorial method of representing entity specifications would be beneficial to the design process, and furthermore, that this same method can be extended to aid in the visualization of internal structural and behavioural elements.

In 1984 Dr. R.J.H Buhr published System Design with Ada [1] which described a pictorial, "Black Box" approach to the design of Ada systems. The motivation behind these Buhr Diagrams was that "improved communication and enhanced intuition will lead to superior design quality in actual projects." [1] and that a picture is indeed worth a thousand words (or lines of code). The methodology was extended and enhanced in 1990 in Buhr's Practical Visual Techniques in System Design: With Applications to Ada [2] and the diagrams are now called Machine Charts.

Machine Charts describe system partitioning in terms of boxes and interactions between the boxes. Initially, a system, or system component, is viewed as a single box or set of boxes with connecting lines called events. Events are the means by which the boxes

interact. As design progresses, boxes are partitioned into more boxes which are in turn partitioned again until each box represents a single system component.

As the interaction between boxes is formalized, they are assigned fingers and buttons with which to initiate and respond to interaction. Buttons are analogous to publicly visible procedure calls in packages (the box representing the package) or to task entries. Events are formalized into fingers pushing buttons and may provide data paths between boxes. Within each of these boxes or buttons may be an engine (a box with diagonally connected corners) which performs the work and has no inherent place structure (may not be further partitioned).

## A VISUAL FORMALISM FOR VHDL

Due to their inherent concurrency, we view VHDL entities as comparable to Ada tasks and Machine Chart actors, they are therefore drawn as parallelograms. The major differences between entities and actors are:

1. Entities may have multiple engines while an actor only one.
2. Communication between entities and processes is through signals and ports rather than fingers and buttons. Fingers are only used to communicate with subprograms.
3. VHDL entities may not be dynamically instantiated as may be Ada tasks and Machine Chart actors.

Signals and ports provide interaction paths between entities as fingers and buttons do between actors. They are drawn as lines and no direction of data flow is indicated as signals merely provide a place to put a value and anybody connected to it may read or write. The main differences between signals and fingers are:

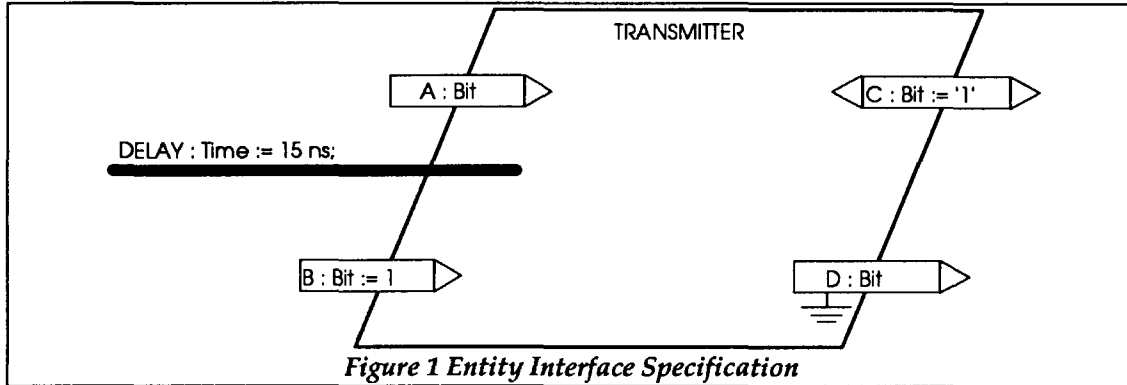
1. Interactions with entities are passive and do not lock the entity initiating the interaction whereas a push on a button of an actor locks the initiator until the push is serviced or it times out.
2. Ports are not analogous to actor buttons because they merely provide pathways to engines and do not perform work.
3. The ports of entities may be dynamically connected to and disconnected from signals whereas fingers between actors are fixed.
4. Signals may be used for data storage while fingers may not.
5. Both signals and ports may be driven and may have multiple drivers while only actor buttons in Machine Charts may be driven.

Processes are analogous to engines and are thus illustrated as boxes with diagonal corners connected. This notation may also be used to illustrate groups of concurrent signal assignments and non guarded blocks (guarded blocks are a special case and have a slightly different notation). Differences between processes and engines include:

1. Processes may only access variables declared within the process while engines may access variables anywhere within the actor.

- Engines may push (Ada task or subprogram call) and pull (Ada ACCEPT) buttons while processes may only drive signals or ports, call subprograms, or wait on signal transactions.

We now describe by example how one may illustrate VHDL constructs. We begin with entity interface declarations and progress through structural models, behavioural models, and mixed models.

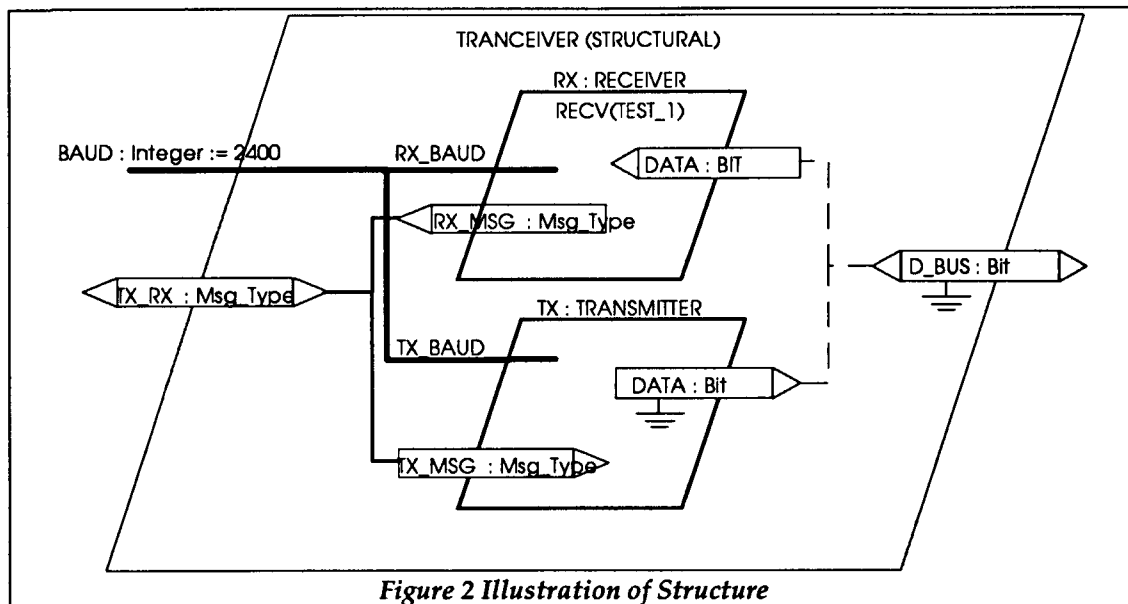


At their most basic level, V-Charts are an excellent vehicle with which to document entity interfaces. Entities are represented as parallelograms with the details of ports and generics shown in fig 1.

Ports are shown as rectangles with points indicating their mode. The interior of the rectangle indicates the port name, data type and default value. Additionally, the 'ground' symbol may be attached to indicate ports of kind bus. Generics are illustrated as heavy lines crossing the edges of the entity and must explicitly state the name of the generic, its data type, and default value. Generics may be differentiated from signals by their heavy line weight.

The entity in fig 1 has two ports of mode in; A will assume the value Bit'left and B will assume the value '1' when open or not driven. C is of mode inout (or buffer) and will assume '1' when open or not driven. D is of mode out and kind Bus (as indicated by the ground symbol). This entity also has a generic called DELAY that will default to 15 ns if not associated with an actual in an association list. The entity illustrated in fig 1 translates to the following VHDL:

```
entity TRANSMITTER is
  generic ( DELAY : Time := 15 ns );
  ports      (
    A : in Bit;
    B : in Bit := 1;
    C : inout Bit := 1;
    D : out Bit bus );
end TRANSMITTER;
```



The logical extension of interface illustration is the illustration of VHDL structure. Structural models are composed of the interconnection of one or more entities in an architecture or configuration of a higher level entity. Fig 2 illustrates an architecture named STRUCTURAL of an entity named TRANSCEIVER. The entity declaration comprises 2 ports (TX\_RX, and D\_BUS) and a generic (BAUD). Internally, the architecture (STRUCTURAL) of TRANSCEIVER is composed of two components; RX : RECEIVER and TX : TRANSMITTER. Furthermore, it should be noted that RX is bound (to RECV(TEST\_1)) and that TX remains unbound. Both entities are connected to D\_BUS with a dashed line to indicate the fact that TX/DATA is of kind bus and may not always be driving D\_BUS and that /D\_BUS is also of kind bus and may not always drive RX/DATA. This would translate into the following VHDL:

```

architecture STRUCTURAL of TRANSCEIVER is
  component RECEIVER
    generic (
      RX_BAUD : Integer );
    port (
      RX_MSG : out MSG_TYPE;
      DATA : in Bit );
  end component;
  component TRANSMITTER
    generic (
      TX_BAUD : Integer );
    port (
      TX_MSG : in MSG_TYPE;
      DATA : out Bit bus );
  end component;
  for RX : RECEIVER use entity RECV(TEST_1);
begin
  RX : RECEIVER
    generic map (
      RX_BAUD => BAUD )
    port map (
      RX_MSG => TX_RX,
      DATA => D_BUS );
  TX : TRANSMITTER
    generic map (
      TX_BAUD => BAUD )
    port map (
      TX_MSG => TX_RX,
      DATA => D_BUS );
end STRUCTURAL;

```

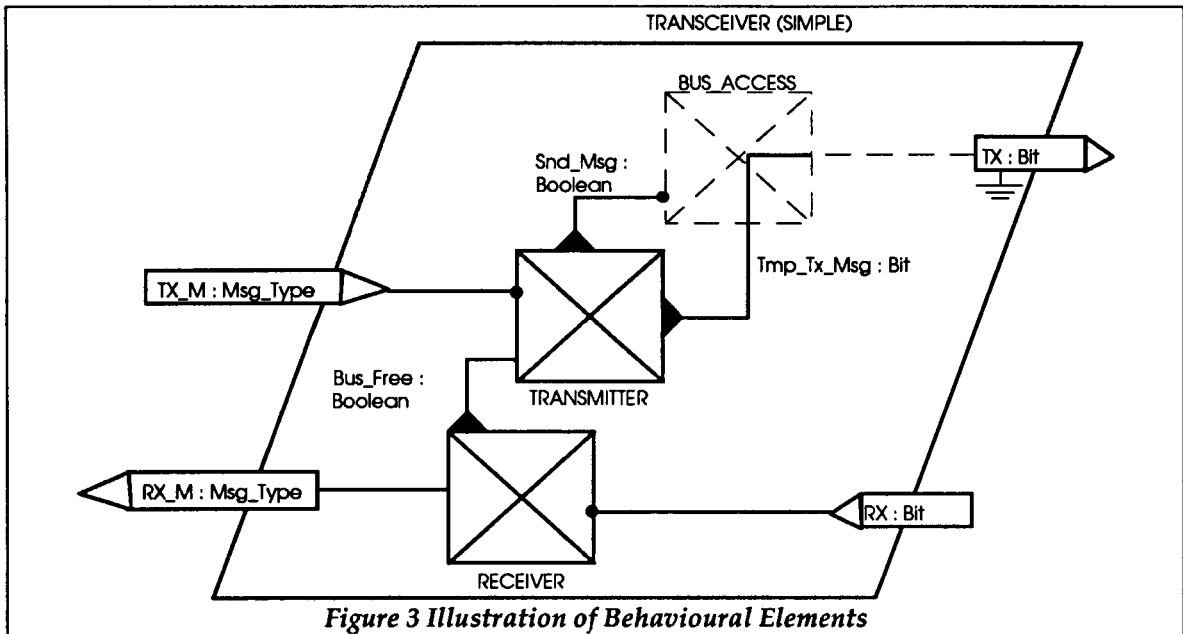


Figure 3 Illustration of Behavioural Elements

The illustration of behavioural elements is the most complex usage of the methodology and requires significant extensions to Buhr's symbology. Processes, concurrent signal assignments, and non guarded blocks are shown as boxes with diagonal lines through them (like Buhr's engines). Process signal or port sensitivity is shown as a dot at the signal/process boundary. This indicates that the process is sensitive to events or transactions on the signal. Solid triangles are used to illustrate that a signal is read, written, or both.

Guarded Blocks are also illustrated as boxes with diagonal crosses but are drawn with dashed lines to distinguish them. Signals evaluated in the guard condition statement are identified with dots at the signal/block boundary. Guarded signals, or connections from non guarded signals to ports, are illustrated with dashed lined to indicate they may not always be driven.

The internal details of processes or blocks may be described in any way the designer chooses; Flow charts, state charts, or pseudo code are among the possibilities.

The entity above is illustrated behaviorally. It is composed of an entity declaration composed of four ports of various modes, data types, and kinds. The architecture (SIMPLE) of the entity is also illustrated. It would translate to VHDL similar to the code below:

```

architecture SIMPLE of TRANSCEIVER is
    signal BUS_FREE, SND_MSG : Boolean;
    signal TMP_TX_MSG : Bit;
begin
    RECEIVER : process ( RX )--note that RX could
    begin
        ??? <= RX;
        RX_M <= ???;
        BUS_FREE <= ???;
    end process RECEIVER;

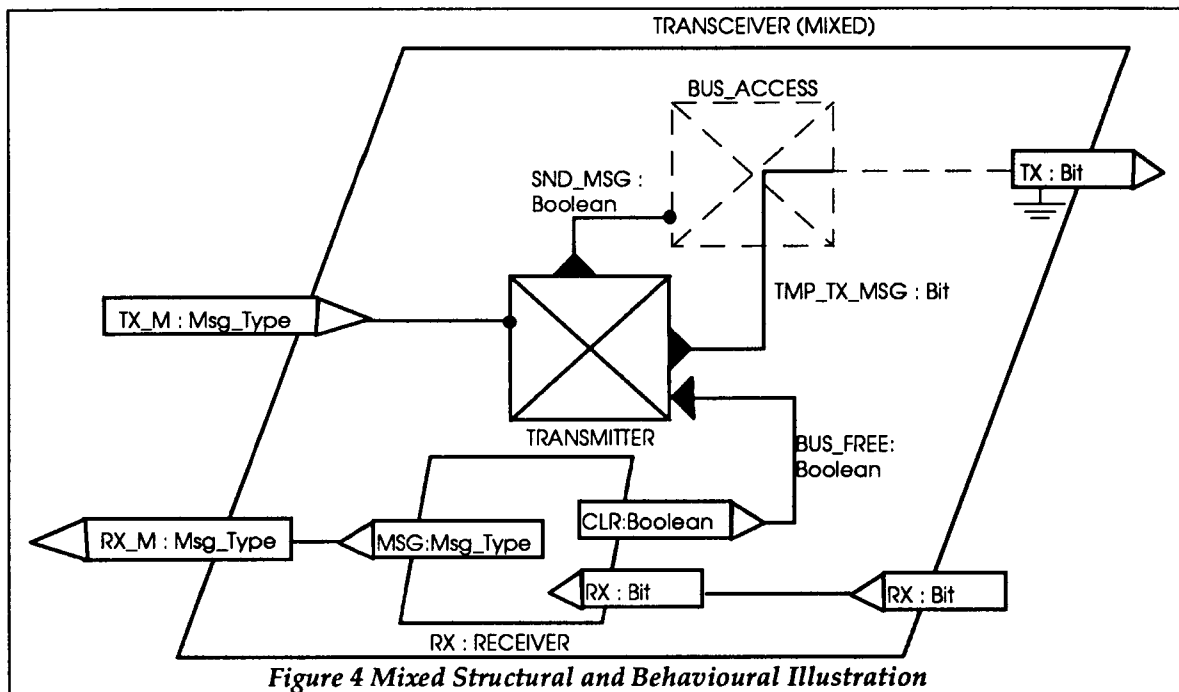
```

```

TRANSMITTER : process
begin
    wait on TX_M;           --note that TX_M could
    ??? <= BUS_FREE;       --have been in the
    SND_MSG <= ???;        -- process sensitivity
    TMG_TX_MSG <= ???;     --list instead.
end process TRANSMITTER;

BUS_ACCESS : block ( SND_MSG )
begin
    TX <= guarded TMP_TX_MSG;
end block BUS_ACCESS;
end SIMPLE;

```



The methodology is also well suited to mixed modeling, in which structural and behavioural elements are combined in a single entity. In the example above, an entity called TRANSCEIVER is declared and an architecture called MIXED is illustrated. MIXED contains an unbound component RX : RECEIVER, a process called TRANSMITTER which is sensitive to transactions or events on TX\_M, and a guard block called BUS\_ACCESS which selectively drives TX based on the value of SND\_MSG. The architecture MIXED would translate to:

```

architecture MIXED of TRANSCEIVER is
    component RECEIVER
        port (
            MSG : out Msg_Type;
            CLR : out Boolean;
            RX : in Bit
        );
    end RECEIVER;
    signal BUS_FREE, SND_MSG : Boolean;
    signal TMP_TX_MSG : Bit;

```

```

begin
  RX : RECEIVER
    port map      (      MSG =>  RX_M,
                    RX      =>  RX,
                    CLR     =>  BUS_FREE );

  TRANSMITTER : process (TX_M)
  begin
    ??? <= BUS_FREE;
    SND_MSG <= ???;
    TMP_TX_MSG <= ???;
  end process TRANSMITTER;

  BUS_ACCESS : block (SND_MSG)
  begin
    TX <= guarded TMP_TX_MSG;
  end block BUS_ACCESS;
end MIXED;

```

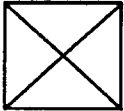
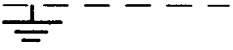
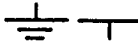


## CONCLUDING REMARKS

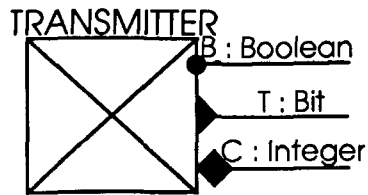
In this paper we have presented a visual methodology for developing and documenting VHDL models at all levels of abstraction. Although our approach is functional in its own right, it is by no means complete. As work in this area progresses, we intend to consider the following:

- How to document Global signals
- How to specify generics and ports for positional assignment in component declarations and configurations.
- How to graphically illustrate types, files, constants and aliases and differentiate between those publicly visible and invisible.
- Changes to VHDL introduced by VHDL 92.

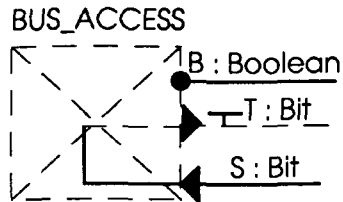
An avenue of research which has potential is the development of an automated tool set to generate V-Charts from VHDL and VHDL skeleton code from V-Charts. Such a tool set could be based on Case Works RT by Micro Processor Toolsmiths Inc or similar tools that use Buhr Diagrams or Machine Charts. We believe that such a toolset would facilitate the retro-documentation of existing VHDL libraries as well as the development and documentation of large VHDL models. At present, however, we are not perusing this avenue.

## SYMBOLOLOGY SUMMARY

PICTURE	MEANING	SAMPLE VHDL
<p>RECEIVER</p> 	<p>Process, Concurrent Signal, or non Guarded Block.</p>	<pre>RECEIVER : process begin end process RECEIVER;</pre> <pre>RECEIVER : T &lt;= 7;</pre>
<p>SEND : Integer := 10;</p> <hr style="width: 100%;"/> <p>TX : Std_Logic := W;</p> 	<p>A non-guarded signal of type integer and default value of 10</p> <p>A guarded signal of kind bus which defaults to W when not driven.</p>	<pre>signal SEND : Integer := 10;</pre> <pre>signal TX : Std_Logic bus := W;</pre>
	<p>Indicates a port or signal of kind bus</p>	<pre>port ( A : out Bit bus );</pre>
	<p>Indicates a signal of kind register</p> <p>Indicate weather a block or process reads, writes, or reads and writes a signal. Shown at engine/signal boundary.</p>	<pre>signal A : Bit Bus;</pre> <pre>signal A : Bit Register;</pre>
	<p>Used to indicate the a process is sensitive to a signal or the signal is part of the guard condition of a guard block. Shown at engine/signal boundary.</p>	



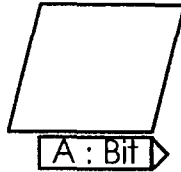
PROCESS sensitive to B, driving T, and both reading and writing C



GUARD BLOCK using B in the guard statement and driving signal T of kind register with the value of S.

TRANSMITTER

An entity named Transmitter



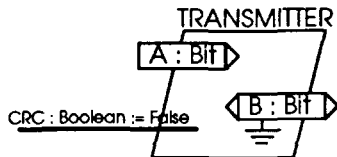
A port whose mode is in the direction it is pointing. If it points into an entity, its mode is in, etc.



A port of mode inout or buffer

MEM\_SIZE : Integer := 1024;

A generic called MEM\_SIZE of data type integer and default value 1024. Generics may be distinguished from signals as they are lines of heavier weight and cross entity boundaries. Signals may only enter and exit entities through ports.



An entity named TRANSMITTER with a port called A of mode in and data type Bit, a port called B of mode inout (or buffer), data type Bit and kind bus, and with a generic called CRC of type Boolean and default value False.

```
TRANSMITTER : process (B)
begin
  T <= '0';
  TEMP := C;
  C <= 1000;
end process TRANSMITTER;
```

```
signal T : Bit register;
signal S : Bit;
BUS_ACCESS : block (B)
begin
  T <= guarded S;
end block BUS_ACCESS;
```

```
entity TRANSMITTER is
end TRANSMITTER;
```

```
port ( A : out Bit );
```

```
port ( B : inout Bit);
port ( B : buffer Bit);
```

```
generic(MEM_SIZE:Integer:=1024);
```

```
entity TRANSMITTER is
generic (CRC:Boolean:= False);
port ( A : in Bit;
      B : inout Bit bus );
end TRANSMITTER;
```

## **Referances**

- [1] R.J.A. Buhr, *System Design with Ada*, Englewood Cliffs, NJ, Prentice-Hall Inc., 1984
- [2] R.J.A. Buhr, *Practical Visual Techniques in System Design : With Applications to Ada*, Englewood Cliffs, NJ, Prentice-Hall Inc., 1990