Formalization of digital circuits using the $B$ method

J-L. Boulanger$^1$, A. Aljer$^2$, G. Mariano$^3$

$^1$Consultant
$^2$LIFL University of Sciences and Technology, Lille, France
$^3$INRETS The French National Institute for Transport and Safety Research, Arcueil, France

Abstract

The goal of this paper is to show how it is possible to combine the advantages of the $B$ method in order to design secure (secure in this paper means more than correct, the design performs what the client wants, furthermore it guarantees to note the unwanted cases) digital circuits that may be easily developed and does not need a design test. The circuit design may be based on the libraries of well-known circuit design language like VHDL. Our goal is to make use of $B$ method to produce the electronic or numeric circuits. At the beginning, the circuit specifications are written in the abstract machine. The refinement direction is determined by the basic elements which are used to construct the desired circuit. So the designer can orient the development to the needed level. This level can be found as a basic library in $B$. We demonstrate how VHDL packages can be translated as $B$ circuit components for giving to the designer a high-level view. Using this approach, one can develop a circuit of which each part of which all the specification parts has already been proven to be correct.

1 Introduction

The $B$ method due to J.R Abrial [3] is a formal method for the incremental development of specifications and their refinements down to an implementation. It is a model-based approach similar to $Z$ [8] and VDM [6]. The software design in $B$ starts from mathematical specifications. Little by little, through many refinement
steps ([7]), the designer tries to obtain a complete and executable specification. This process must be monotonic, that is any refinement has to be proved coherent according to the previous steps of refinement. The abstract machine [2] is the basic element of a B development. It encapsulates some state data and offers some operations. The description of an abstract machine is composed of three parts: the declarative part which describes the states and their properties, the execution part which introduces operations and composition clauses. In the B development, the proofs accompany the construction of software. Each time an abstract machine is defined or modified, there are proof obligations related to its mathematical consistency; if the machine is a refinement or an implementation, there are also proof obligations of its correctness with respect to the previous steps of the development chain. On the other hand, VHDL ([1] and [4]) is an IEEE Standard since 1987. VHDL is a programming language used to express the hardware components with a high level of abstraction. It is a good utility to describe the integrated circuits, or complete system of hardware and software. Also it can be used to declare the circuit behaviour.

2 Modelling of digital circuits

This section describes and models some synchronized basic components which will be then reused. A synchronized circuit is view as a box, within which an (or more) input line is entering, and out of which an output line (or more) is emerging. A synchronized circuit is supposed to be synchronized by a clock.

2.1 Modelling of a basic logic gate, Not

The Not component is the simplest logic gate. It is described with a boolean expression that described the behaviour: \( (\text{in} = 0 \Rightarrow \text{out} = 1) \) \& \( (\text{in} = 1 \Rightarrow \text{out} = 0) \). We can use the last expression to write an abstract machine:

**MACHINE B_Not_0**

**DEFINITIONS**

\[
\text{Compute-(xx,yy)} == ((xx = \text{TRUE}) \Rightarrow (yy = \text{FALSE}) \& ((xx = \text{FALSE}) \Rightarrow (yy = \text{TRUE})))
\]

**VARIABLES** in, out

**INVARIANT** in: BOOL \& out: BOOL \& Compute-(in,out)

**INITIALISATION** in: BOOL \& out: BOOL \& Compute-(in,out)

**OPERATIONS**

\[
\text{In (val) = PRE val:BOOL Then in, out:BOOL & in val & out:BOOL & Compute-(in.out) END; val <= - Out = val := out END}
\]

In this abstract machine the input and the output are represented by the global variables in and out. These variables can be assigned by all of the defined operations. Each output is attached to a read operation and each input is attached to a store one. The environment of the circuit will use these operations to know the circuit output or to change the input. The behaviour of the port is described using the Compute definition, which gives us the possibility to express systematically the definition of the function. The INVARIANT clause states the static laws, in our case the properties, that the data must obey whatever the operation that is
applied to it. This abstract machine is automatically proved by the $B$ tool and is not-deterministic since we use the operator $\text{list}\_\text{var} : (\text{predicate})$. This operator indicates that the list of variable become such that the predicate is true. We build a library that contains the standard ports, structure which is based on the last machine form. Given a circuit, we used the following modifications:

- The additions of the operations associated to the supplementary ports; the circuits have more than one port in general, so we use $\text{in}_1 \ldots \text{in}_n$ to mention the $n$ in ports of the circuit and $\text{out}_1, \ldots, \text{out}_m$ to mention the $m$ out ports.
- The modification of the $\text{Compute}\_\text{definition}$ corresponding to the logical specification.

### 2.2 Modelling of complex circuit: a multiplexor

A multiplexor circuit is used in this section as an example to show how to reuse components previously modelled in order to obtain complex integrated circuits. Basing ourselves on the logical specification of a component, we can supply an assembly of many simple components to achieve the desired function. This is called the synthesis of a numeric circuit. The multiplexor with $n$ inputs is a circuit with $n$ principal inputs and one output. The output value is equal to the value of the input of the number $i$; $i$ is determined by other inputs. In our example ($n = 2$), the input named $\text{Select}$ gives the possibility to choose one of the two inputs $a$ and $b$. We write this: 

\[
((\text{Select}=\text{FALSE}) \Rightarrow (\text{out} = \text{in}_1)) \& ((\text{Select}=\text{TRUE}) \Rightarrow (\text{out} = \text{in}_2)).
\]

To describe the abstract machine for the multiplexor we can use one that is similar to the $\text{Not}$ gate which is presented above. The principal difference between the two machines is the definition of the $\text{compute}\_\text{function}$:

**MACHINE B-Mux-0**

**DEFINITIONS**

Compute-$(xx,yy,zz,\text{res}) := \text{bool}(((xx=\text{FALSE}) \Rightarrow (\text{res} = yy)) \& ((xx=\text{TRUE}) \Rightarrow (\text{res} = zz)))$

**VARIABLES** $\text{Select, in}_1, \text{in}_2, \text{out}$

**INVARIANT**

$\text{Select, in}_1, \text{in}_2, \text{out} : (\text{Select} : \text{BOOL} \& \text{in}_1 : \text{BOOL} \& \text{in}_2 : \text{BOOL} \& \text{out} : \text{BOOL} \& \text{Compute-}(\text{Select, in}_1, \text{in}_2, \text{out}))$

**INITIALISATION**

Select, $\text{in}_1, \text{in}_2, \text{out} : (\text{Select} : \text{BOOL} \& \text{in}_1 : \text{BOOL} \& \text{in}_2 : \text{BOOL} \& \text{out} : \text{BOOL} \& \text{Compute-}(\text{Select, in}_1, \text{in}_2, \text{out}))$

**OPERATIONS**

$\text{in}_1(\text{val}) = \text{in}_1(\text{out}) ; (\text{in}_1 : \text{BOOL} \& \text{in}_1 : \text{val} \& \text{out} : \text{BOOL} \& \text{Compute-}(\text{Select, in}_1, \text{in}_2, \text{out}))$

$\text{in}_2(\text{val}) = \text{in}_2(\text{out}) ; (\text{in}_2 : \text{BOOL} \& \text{in}_2 : \text{val} \& \text{out} : \text{BOOL} \& \text{Compute-}(\text{Select, in}_1, \text{in}_2, \text{out}))$

$\text{Gate} (\text{val}) = \text{Select, out} ; (\text{Select} : \text{BOOL} \& \text{Select} : \text{val} \& \text{out} : \text{BOOL} \& \text{Compute-}(\text{Select, in}_1, \text{in}_2, \text{out}))$

$\text{val} \leftarrow \text{Out} = \text{val} \leftarrow \text{out}$

**END**

This abstract machine is fully proved by the $B$ tool. The previous $B$ specification can be refined by just modifying the previous boolean expression defined in the definition $\text{Compute}\_\text{.}$ We rewrite the previous boolean expression that describe the 2-multiplexor: 

\[
\text{out} = (\text{in}_1 \& \text{NOT}(\text{Select})) \lor (\text{in}_2 \& \text{Select})
\]

**REFINEMENT B-Mux-1**

**REFINES** $B_{\text{Mux-0}}$

**DEFINITIONS**
The simple circuits machines are used to refine the complex ones. In this example, four machines are used in the refinement of the multiplexor. This abstract machine is just a systematic translation which may be done automatically.

2.3 Modelling methodology

In this section, our aim is to give a general method to model a circuit without knowing the details of the desired circuit. At first the analogy of development between the $B$ method and the numeric circuits design is presented. In $B$ the initial specifications of the desired circuit are written in the Abstract Machine using mathematical expressions. The abstract machine is refined to obtain the first refinement machine. An abstract machine refinement is performed by determining the data types or by adding algorithms that satisfies a part of the specifications. The $B$ tools generate the necessary proofs to demonstrate that the refinement is correct. Many of these proofs are automatically proved. The others may be proved in cooperation with the designer. The refinement step may be repeated many times. More and more the components of the circuit become precise as well as its behaviour. The result of the last step of the refinement is called the implementation machine, in which the behaviour of the circuit is deterministic. The implementation may be TRUTH TABLES; these tables are concise but are not suitable to describe the large scale circuits; at the same time they need a lot of proofs. In our methodology we represented each port of the circuit by a local variable. So that the connection between the ports is represented as a fixed relation between the global variables. These relations are represented in the INVARIANT clause which contains all the relations that must be satisfied in a machine and in its refinements. In the $B$ method the OPERATIONS clause represents the dynamic part of a machine in which the values of the global variables may be changed. So a signal propagation is done by an operation call. The $B$ method gives us the possibility to reuse other machines. So many already defined machines may be reused as a components of
more complex ones. An already defined machine may also be renamed and reused by adding other operations to produce a more developed circuit. Using this method, 80% of the needed proofs are proved automatically, and the others need a cooperation with the designer. Sometimes little changes of the source machines are necessary to create the desired proofs. This enables the designer to correct the possible errors of his design. The general method described in this section provide the capabilities to define some abstract machines that modelling the behaviour of some circuit. But we want modelling some realistic VHDL component and we need some abstract machines that correspond to standard library.

<table>
<thead>
<tr>
<th>Circuits synthesis</th>
<th>B Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Specifications</td>
<td>Abstract machine</td>
</tr>
<tr>
<td>Architecture Specification</td>
<td></td>
</tr>
<tr>
<td>&amp; Behaviour Details</td>
<td>Refinements</td>
</tr>
<tr>
<td>Validation “functional to material”</td>
<td>Proof</td>
</tr>
<tr>
<td>Physical Description</td>
<td>Implementation Machine</td>
</tr>
<tr>
<td>Port</td>
<td>Global Variable</td>
</tr>
<tr>
<td>Connection</td>
<td>Invariant of relation between variables</td>
</tr>
<tr>
<td>Signal Propagation</td>
<td>Operation call (transmission of values)</td>
</tr>
<tr>
<td>Reusing</td>
<td>Importation + Renaming</td>
</tr>
</tbody>
</table>

3 VHDL libraries

VHDL depends on the conception of modules. The hierarchy enables the programmer to write the program as units. Some of these units are elementary expressions which can be directly compiled. The others may be decomposed into many units. This mechanism makes the work of team easier, especially when the complexity of the module increase. A special library is built for each programmer using his correct units which have been compiled. The programmer may use his libraries and the libraries of his colleagues and the general libraries. The following section introduces one of the most used VHDL libraries, the STD_LOGIC_1164 library and its equivalent in $B$.

3.1 The STD_LOGIC_1164 Library

There are nine values in this logic; each of these values may be assigned to a variable or to a signal and form the extended bit. An extension of the classic logic is built to treat these nine values in the VHDL package called STD_LOGIC_1164 instead of two values in the classic one. So the elements of this package are:

- a principal type that contains nine values, many subtypes which contain some of these values. And complex types which contain vectors of these types,
- the basic logic operations over the previous types,
- and many functions to cast a type to another,
- many other functions to solve the problem of the signals which have many different resources,
two functions to determine the direction of the change of a signal if it is with *rising_edge* or *falling_edge*,

three functions to decide if a signal is not determined (if its value is U, X, Z, W or _),

the `STD_LOGIC_1164` package contains also many attributes, which are adjectives that may given the language components.

### 3.2 The B components for `STD_LOGIC_1164`

The `B` counterpart of the VHDL `STD_LOGIC_1164` library mostly consists in two machines. The first machine, named `B_STD_LOGIC_1164_0`, contains all the definitions of types and the operations which concern the extended bit. The second machine, `B_STD_LOGIC_1164 VECTOR_0`, depends on the first to define the vectors and the corresponding operations. The machine `B_Signal_0` was created to process the signals.

- `B_STD_LOGIC_1164_0` We define the principal type `STD_ULOGIC` as a set of values, and its subtypes as subsets of this set. We define the subset as `CONSTANTS`. Under the clause `PROPERTIES` we declare the elements of these types. This method of declaration decreases the number of the necessary proofs which are needed to verify the consistency of the machine because the expressions under `PROPERTIES` clause are added as axiomes. The VHDL functions are represented under the `OPERATIONS` clause in `B`. Each operation is a call of a mathematic function. For each element of the their domains we define the correspondent element in the codomain using tables. We define these tables as constants which have their values under the `PROPERTIES` clause. In order to determine the type of the parameters we use the substitution `PRE pred THEN body END;` which enables to verify the type of the input variables. The type of the output is decided indirectly in the operation in the first affectation (`:=`). Each operation consists of a mathematic function call, so the result is included in the codomain of that function. As opposed to VHDL, the `B` method does not accept overloading: VHDL accepts several functions with the same name but with different signatures, and the wished function is decided only during the execution depending on the number and the type of its parameters. So in the `B` machines of the `STD_LOGIC_1164` we have chosen many `B` operations with different names, we introduced the type of parameters in these operations as a part of the their names. For example, six functions in VHDL have the name `TO_X01`. One has an Input variable of type `BIT`, another has an extended bit variable as input and the others have extended bit vector variables as Input. So we have in this machine two operations: From_std_ulogic_to_X01 and From_BIT_to_X01. The implementation of this operation depends on the designers needs so we used the substitution `skip`.

```vhdl
in VHDL Type std_ulogic IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '_')
in B SETS STD_ULOGIC = {UU,XX,00,11,ZZ,WW,LL,HH,DD}
in VHDL SUBTYPE std_logic IS resolved std_ulogic
```
in B CONSTANTS STD_LOGIC
PROPERTIES STD_LOGIC <: STD_ULOGIC & STD_LOGIC=STD_ULOGIC – {DD}
in VHDL SUBTYPE XOI is resolved std_ulogic RANGE 'X' TO '1'
in B CONSTANTS XOI
PROPERTIES XOI <: STD_ULOGIC & XO1 = {XX,OO,II}
in VHDL SUBTYPE UXOl is resolved std_ulogic RANGE 'U' TO '1'
in B CONSTANTS UXOl
PROPERTIES UXOl <: STD_ULOGIC & UXOl = {UU,XX,OO,II}
FUNCTION "NOT" (I:std_ulogic) RETURN UXOl IS
BEGIN RETURN (not_table(I)) END "NOT";
in B CONSTANTS NOT-STD
PROPERTIES NOT-STD:STD_LOGIC –> STD_ULOGIC
&NOT-STD = { UU -> UU, XX-> XX, OO->II , II->OO,  
ZZ->XX, WW->XX, LL->II, HH->OO, DD->XX }
OPERATION out <-- Not(in) = PRE in :STD-ULOGIC THEN out := NOT-STD(in) END

• B_STD_LOGIC_1164_VECTOR_0 In order to deal with the vectors, we  
 wrote a machine which can use the previous one. Using the SEES clause, 
 this machine can read all the constants of the previous seen machine (ie: the 
 tables of the functions). It contains a general function named Apply which 
 takes four parameters op, inl, in2, out. It applies the operation op, which is 
 a table function in the previous machine, over all the elements of the input 
 vectors inl, in2 and gives as output the vector out. We define the vector 
 as a structure of two elements, the first being a function. Its domain is the 
 maximum size of the vector and its codomain is the value of the vector. 
 The second element is the size of the vector. We collect all the vectors used 
 in a set with a maximum size. Under the OPERATIONS clause we find 
 the correspondent operations of vector functions in the STD_LOGIC_1164 
 package.
in VHDL TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE => ) OF std_ulogic;
in B CONSTANTS STD_LOGIC VECTOR, STD_ULOGIC VECTOR,Max_ElemeNT 
PROPERTIES Max_ElemeNT:NAT1 &
STD_LOGIC VECTOR = struct ( vector: 1 ..Max_ElemeNT -->STD_LOGIC,vector_size:NAT1 ) &
&card(STD_LOGIC VECTOR)<MAX_VECTOR &
&!(xx:STD_LOGIC_VECTOR) --> xx\vector_size<Max_ElemeNT) ...
in VHDL FUNCTION "and" ( 1, r : std_logic_vector ) RETURN std_logic_vector IS
ALIAS lv : std_logic_vector ( 1 TO i'LENGTH ) IS i;
ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
VARIABLE result : std_logic_vector ( 1 TO i'LENGTH );
BEGIN IF ( i'LENGTH /= r'LENGTH ) THEN ASSERT FALSE
REPORT "arguments of overloaded 'and' operator are not of the same length" 
SEVERITY FAILURE;
ELSEOR i IN result'RANGE LOOP result(i) := and_table (lv(i), rv(i));
END LOOP;
END IF;
RETURN result;
END "and";
in B DEFINITIONS
**APPLY**(op, in1, in2, out) ==

ANY vv WHERE vv : STD_LOGIC_VECTOR & !xx.((xx: 1.. max(in1 vector size, in2 vector size)) => ((vv vector)(xx) = op((in1 vector)(xx), (in2 vector)(xx))))

THEN out := vv END

**OPERATIONS**

out <= And(in1, in2) = PRE in1: STD_LOGIC_VECTOR & in2: STD_LOGIC_VECTOR & in1 vector size = in2 vector size THEN APPLY(AND_STD, in1, in2, out) END;

- **B_Signal 0**

Most of the operation in the previous two abstracts machines are included in order to find correspondents to the logic function or to the type converting functions. So we need to simulate a signal in order to give a concrete implementation to these functions. These abstract machine may be used to solve this problem and could be used as a base to translate other VHDL packages. Here the signal is expressed in this machine as a structure of two elements, the first is a value of std_logic type (or std_ulogic) and the second is a pointer to another structure of the same type.

**ABSTRACT_CONSTANTS SIGNAL_ULOGIC**, **SIGNAL_nil**, F_SIGNAL

**PROPERTIES** SIGNAL_ULOGIC = struct (value : STD_ULOGIC, next : SIGNAL_ULOGIC)

& card(SIGNAL) < MAX_SIGNAL & SIGNAL_nil : SIGNAL_ULOGIC

& F SIGNAL : SIGNAL_ULOGIC -> SIGNAL_ULOGIC

& F SIGNAL (SIGNAL nil) = rec(value := DD, next := SIGNAL nil)

With these definitions, it is easy to define an operation which correspond to the RISING_EDGE functions which return a TRUE value if the signal value changes from a low level to a high one.

in VHDL FUNCTION rising_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN IS

BEGIN

RETURN (s'EVENT AND (To_XO1(s)='1') AND (To_XO1(s'LAST_VALUE)='0'));

END;

in B out <= rising_edge(in) = PRE in: SIGNAL_ULOGIC THEN

out: (out: BOOL & out = bool((in'value='1') & ((F SIGNAL (in'next)')value = '0'))) END;

The last three B machines give the principal characteristics of the standard package. To be closed to the standard package we tried to write the B correspondents with the same elements names and definitions. But because of the differences between the two languages, many points must be taken of care:

- The correspondence between the STD_LOGIC_1164 package parts and the B machines parts is not one to one so we can not transmit some comments which give some details about the international pragmas of all parts of this package.

- In many functions in STD_LOGIC_1164 we find the instruction : ALIAS lv: std_logic_vector (1 TO l'LENGTH ) IS lv we use this instruction to arrange its input vector so that their elements occupy the first positions in the local function array. This arrangement facilitates vector treatments in the function. In B components presented above, we proposed that the vectors are normally represented.

- The last versions of the STD_LOGIC_1164 contain many attribute instructions. Each attribute instruction associates a characteristic with a type or with an object. For example attribute REFLEXIVE of resolved function is TRUE These attributes may be used in the package or in the libraries that depend on this package. In our machines each time we need these attributes, we use expressions that gives the necessary characteristics.
For the future work when we will translate the machines that depend on this package we thought to find the necessary expressions each time we use these attributes.

- In VHDL we can use general expressions as: TYPE STD_LOGIC_VECTOR is ARRAY (NATURAL RANGE <=) of std_ulogic B doesn’t give the same capacity. So, for our example, we thought to determine the limits of the vector range. We do so in the B_STD_LOGIC_1164_VECTOR_0 machine.

- In the To_bit function, which converts a given value from extended bit type to a normal bit type, we give the value xmap to the undetermined values (U, Z, W, X, -). In the function To_bit the xmap is initialised to 0. It may be redefined in the future only by changing the function To_bit. To have the same modification in B, we must change the PROPERTIES clause which is, in the std_logic_1164_0.mch, independent of the To_bit functions.

- The VHDL language gives the programmer the possibility to prepare an error message that may be displayed if there is an error during the execution. In STD_LOGIC_1164, we have many of these massage. For example:

```vhdl
IF (l'LENGTH/=r'LENGTH) THEN ASSERT FALSE REPORT "arguments of overloaded 'nand' operator are not of the same length" SEVERITY FAILURE;
```

3.3 Example

We present in this section, how the designer may use the methodology proposed in the section 2.3 to design a circuit based on the VHDL library. This example is an extension of an example of the multiplexor proposed before in the first section. Instead of two boolean values in the first example, TRUE and FALSE, we use an extended bit with the nine values. The output depends on the value of the input Select_a; it takes the value of In_a if Select_a equals 00 or LL, it takes the value of In_B is Select_a equals II or HH, otherwise it takes the value XX. The following mathematic expression may be summarize by the definition:

\[
\begin{align*}
((\text{Select}_a = \text{OO OR } \text{Select}_a = \text{LL}) & \Rightarrow (\text{out} = \text{in}_a)) & \\
((\text{Select}_a = \text{II OR } \text{Select}_a = \text{HH}) & \Rightarrow (\text{out} = \text{in}_b)) & \\
((\text{Select}_a = \text{UU OR } \text{Select}_a = \text{XX OR } \text{Select}_a = \text{ZZ OR } \text{Select}_a = \text{WW OR } \text{Select}_a = \text{DD}) & \Rightarrow (\text{out} = \text{XX}))
\end{align*}
\]

4 Conclusions

To develop an electronic circuit in B requires to understand the refinement calculus, that is, a certain adaptation time for a circuit designer. Most recent imperative languages like VHDL include facilities as manipulation of a vector without explicit length or functions with various signatures. Because the B method does not include such features, these differences induce difficulties in transforming VHDL to B. One solution to the problem of vector length is used in the B_STD_LOGIC_1164_VECTOR_0 by passing the length as a parameter and by using the universal quantifier V. To design circuits, a large quantity of proofs may be generated and must be proved automatically or in cooperation with the designer.
The size of these proofs depends on the quality of the program, the capacity of the prover, the designer, and the tool we use. For very large circuits, the number of the necessary proofs could be extremely large. When the circuit is designed in $B$, we can prove that it satisfies the required specifications but we have not yet any tools to create the real circuits directly from the $B$ specification. But with $B$ method it is possible to represent the circuits as near as it is needed to the physical level. In [5] authors refined the $B$-NOT-0 abstract machine based themselves on abstracts machines of CMOS transistors. To simulate a signal propagation, the conception of the time must be represented somewhere. In this report a list of extended bit values is used to represent a signal. A global clock may be used to control the harmony between all the signals of the circuit. The most important characteristic of the $B$ method is that it produces a secure circuit. The circuit which are obtained in the implementation satisfies 100% the specifications in the abstract machine. It is possible to add the required safety conditions under the $INVARIANT$ clause in the abstract machine. So the circuit design is completely correct. The error may occur only when we describe the specifications or if the physical circuit does not correspond to the proposed model. The cost and the time of the test is won. To develop a circuit that was proved before, it is enough to prove only the new characteristics. It is not necessary to reprove all the old ones. So the designer may use all the circuits which are designed before, he can make some changes and then prove a small part related to the new characteristics. The $B$ tools can automatically decide which parts of the model are changed or added in order to be reproved. This characteristic is quite important for the necessary modifications of the integrated circuits development.

References


