International Journal of Engineering Research and Development
e-ISSN: 2278-067X, p-ISSN: 2278-800X, www.ijerd.com
Volume 9, Issue 6 (December 2013), PP. 34-49

Correct by Construction design approach through mapping B models into VHDL

Nahla El-Araby[1], Ayman M. Wahba[2], Mohamed Taher[3]
[1]*Electrical and Electronics Engineering Dept., Canadian International College, Cairo, Egypt, www.cic-cairo.com*
[2]*Computer and Systems Engineering Department, Faculty of Engineering Ain Shams University, Cairo, Egypt http://eng.asu.edu.eg*
[3]*Computer and Systems Eng. Dept., Faculty of Engineering, Ain Shams University,* Cairo, Egypt

Abstract:- B method is one of the common paradigms used in formal verification. It offers a strong verification domain as it based on a mathematical and logical approach. The proof obligations (properties that must be satisfied) are automatically generated from the model, also the available tools provides both automatic and interactive proofs. VHDL is a mature implementation domain where many synthesis and simulation tools are available. The work in this paper presents a technique to convert B machines into the corresponding VHDL implementation in order to implement a correct by construction system, which benefits from the advantages of both strong domains, and maintain the properties of the verified model. We reached for a method to cross the gap and convert the B machines into VHDL implementations and a tool was designed to apply the proposed technique. Five popular models were used as workbenches where we applied the developed technique. Simulation at some critical points was used to ensure that the generated VHDL satisfy the verified properties in the original B machine.

Keywords:- Verification, B models, VHDL, correct by construction, real-time.

I. INTRODUCTION

Throughout the previous years, the complexity and size of digital systems has increased dramatically, as a result design flow phases changed a lot. Simulation used to be the most common procedure to assure the correctness of a system under design, but it cannot exhaustively examine all the execution scenarios of the system. A different approach to validate a system by formally reasoning the system behavior is Formal verification, where the system implementation is checked against the requirements or the properties to be satisfied. The most common paradigms are based on theorem proving, model checking and language containment.

People and products safety are directly affected by the reliability of automated systems. Safety aspects should be considered from early design stages up to operational stages and this needs a very accurate design approach [1]. This becomes more sophisticated in real time systems as real-time systems differ from untimed systems in that the correct behaviour relies on computation results plus the time on which they were produced. The resulting state-space explosion makes it infeasible to run a satisfactory number of simulation traces to achieve enough coverage of the state spaces and enough confidence in the design correctness within a project schedule. Even if it were feasible to have extensive coverage of the system, missing only single untested sequence of events may cause the system failure.

The common approach for system design is to start the design cycle by implementing the basic requirements then starting to test and correct errors in the developed design. This "construct-by-correction" approach leads to a long and more expensive design cycle.

Since the B method offers a strong framework for developing and verifying models at different abstraction levels, the verified B models can be used to develop "correct-by-construction" designs, but the problem is that some verified properties may be lost during converting the model into an implementation. VHDL is a mature implementation domain where many synthesis and simulation tools are available.

Automatic conversion of the verified B models into implementation avoids losing any of the verified properties. In addition to providing an implementation directly mapped from the verified model which achieve "correct-by-construction" design approach.

Also the developed implementation will take the advantages of both the verification B domain and the strong well matured VHDL implementation domain.

The work in this paper presents a technique and a tool to convert B machines into the corresponding VHDL implementation in order to build a "correct- by-construction" system, which maintain the properties of the verified B model, and benefit from the advantages of both strong domains.

The paper starts by explaining the definition and basics of formal verification and discusses the different methods used in section 2, and then a detailed explanation for the B method mathematics and modelling approaches are presented in section 3. The grammar for both B machines and VHDL codes are illustrated in section 4 then the proposed technique for conversion from B to VHDL is shown. Section 5 presents the application of the proposed technique on various models which are used as workbenches and shows the simulation results. Conclusion and future directions are presented in section 6.

II. FORMAL VERIFICATION

Formal verification means to thoroughly investigate the correctness of system designs expressed as mathematical models. Formal verification is a useful and powerful technique for guaranteeing the reliability of systems in design stages [2]. In recent years, several approaches to applying formal verification techniques on automation systems dependability have been proposed. These range from formal verification by theorem proving [3] to formal verification by model-checking [4], [5], [6], [7]. Many achievements in the formal verification of real-time systems are presented in [8], [9], [10] and [11]

The verification problems of timed systems are usually exponentially more complex than their untimed counterparts. Most major projects are spending over 50% of their development costs in verification and integration, so using formal verification can substantially reduce the explosive growth of verification and integration costs and improve the quality of system designs in industry. On the one hand, using formal verification for complex real-time systems will likely enhance the intelligence and performance of simulation and testing. For example, coverage metrics can be more precisely mapped to the functions to be verified. Also, formal verification can be used to carefully check the components and the interfaces and progressively could be accepted as standard methods in the automation of industrial quality control. It is claimed that this approach has already had a remarkable effect on the SLAM project of Microsoft, which plans to include model-checking capability in its Windows driver development kit (DDK) [12].

Formal specification is defined by the IEEE standard as a specification written in a formal method. Formal methods are particular type of mathematically-based procedures for the specification, development and verification of systems. Performing appropriate mathematical analysis that contributes to the reliability and robustness of a design is the motivation for using formal methods for design. Systems can be formally described at different levels of abstraction.

The formal description can be used to guide further development activities; moreover, it can be used to verify that the requirements for the system being developed have been entirely and precisely specified. A variety of formal methods and notations available are available, like Z notation, VDM and B-Method.

Verification plays a vital role in the design cycle of any safety critical system. The development of any system is not complete without careful testing and verification that the implementation satisfies the system requirements. In the past, verification was an informal process performed by the designer. But as the complexity of systems increased, it became necessary to consider the verification as a separate step in the overall development cycle. Verification techniques can be either based on simulation or based on formal methods. Simulation is based on a model that describes the possible behavior of the system design at hand. This model is executable in some sense, such that a simulator can determine the system's behavior on the basis of some scenarios. Formal Verification is defined as "establishing properties of hardware or software designs using logic, rather than (just) testing or informal arguments. This involves formal specification of the requirement, formal modeling of the implementation, and precise rules of inference to prove that the implementation satisfies the specification" [13]. Three categories can be used to classify the Formal Verification methods - equivalence checking, model checking and theorem proving.

Formal Verification depends on mathematical models and formal representations for system designs where the model is examined to ensure its correctness according to required behavior.

Unlike simulation-based verification method which is input oriented, formal method-based verification is output oriented as the designer provides the properties of the outputs from the system. Formal verification overcomes the drawback of simulation based methods, by representing the system and its properties mathematically and logically then investigating the models to ensure that the system is satisfying the required properties in all its states. Figure1 illustrates the flow of formal verification.

Design verification is classified in [14] into two types: equivalence checking which verifies that two versions of the design functionally equivalent, and Model checking where we verify that the implementation satisfies the specifications, in other words checking the model against the properties. Another important formal verification technique is theorem proving [15], [16] which is based on a pure mathematical or logical approach where the verification problem is described as a theorem in a formal theory. A formal Theory is a language in which the formulas are written, a set of axioms are developed, and a set of inference rules are used for proving. Theorems can be proved with rules and axioms. A desired property is satisfied if a proof can be constructed from the system axioms and inference rules.

In [17] an international survey of the use of industrial methods in industry is presented. The survey provides a view of the situation by comparing some significant projects used formal verification techniques effectively. Also [18] provides a study of selected projects and companies that have used formal methods in the design of safety-critical systems and [19] gives a general inspection of this industry in the UK.

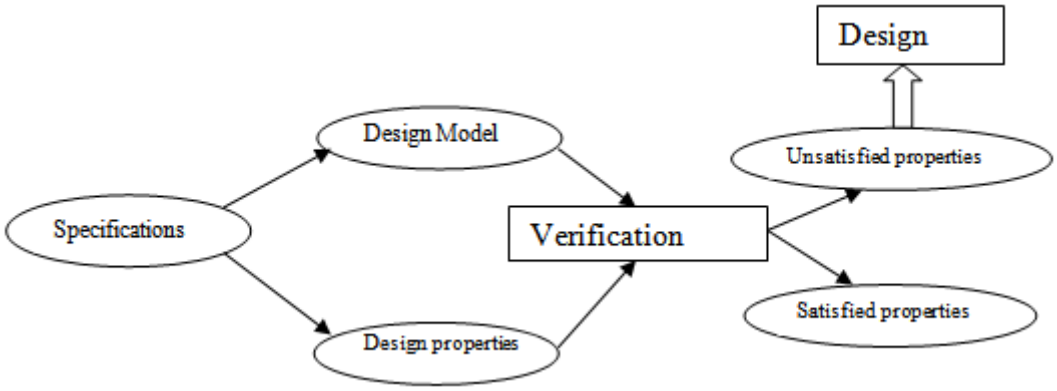

Fig.1: Formal verification process

III. B METHOD

The B method is a model-oriented formal method for engineering software systems developed by Abrial [16]. It is a comprehensive formal method that covers the entire development cycle. The method is based on the mathematical principles of set theory and predicate calculus while its semantics is given using a variant of Dijkstra's weakest precondition calculus [17]. A hierarchy of components that are described using the Abstract Machine Notation (AMN) constitutes a B specification. Each component in a specification represents a state machine where a set of variables defines its state and a set of operations query and modify that state. Generalized substitutions describe state transitions. Constraints on the operation and variable types are described as invariants of a machine. In B models Abstract Machines are the top-level components describing state machines in an abstract way. Refinements are another type of components that exist in a B specification; they represent enriched versions of either an Abstract Machine or another Refinement. The last type of components is Implementations where ultimate refinement of an Abstract Machine is described; both data and operations need to be implementable in a high-level programming language.

Syntax and type checking can be performed on a system modelled in B. Also a B model consistency can be verified to check the preservation of invariants and the correctness of all refinement steps.

Table I: Commonly used B operators

Notation	Semantics
P(X)	Set of all subsets of X
X × Y	Cartesian product of the sets X and Y
X ↔ Y	Set of relation of X to Y, or equivalently P(X x Y)
X —>> Y	Set of partial functions from X to Y
X → Y	Set of total functions from X to Y
X |—>> Y	Set of partial injective functions from X to Y
Id(X)	Identity relation on X
R – 1	Inverse relation on X
Dom(R)	Domain of the relation R
Ran(R)	Range of the relation R
R[X]	Relational image of X under the relation R
X R	Binary relation R restricted to pairs with first component in X
X R	Binary relation R restricted to pairs with first component not in X
R X	Binary relation R restricted to pairs with second component in X
R S	Relation R overridden by S, Equivalent to (dom(S) R) U S
R (X)	Direct product. Defined as {x, (y,z) | x,y Є R ^ x, z Є S}

IV. MAPPING B MODELS INTO VHDL CODES

The version of this template is V2. Most of the formatting instructions in this document have been compiled by Causal Productions from the IJERD LaTeX style files. Causal Productions offers both A4 templates and US Letter templates for LaTeX and Microsoft Word. The LaTeX templates depend on the

official IJERDtran.cls and IJERDtran.bst files, whereas the Microsoft Word templates are self-contained. Causal Productions has used its best efforts to ensure that the templates have the same appearance.

A. Introduction

In this work a tool was developed to automatically convert verified B model into VHDL implementations to provide a way that avoids loosing any of the verified properties during the implementation stage. This section is concerned with the mapping technique developed in this work.

B. Converting B into VHDL

Now we will explain how to convert each clause in the B machine to the required VHDL code. VARIABLES clause in the B machine can be transformed into signal declarations, SETS clause can be transformed into enumerated type in VHDL, OPERATIONS clause can be described using VHDL if or case statements. In this section the details of the mapping process will be discussed.

An initialization step must be performed in the beginning which can be called a flattening process. In this flattening process the root machine that is including B machines as system components is analyzed and whenever an operation from an included machine is called the calling condition is inserted in the included machine operation.

Then the signals assessed in the conditions are examined to find effective and remove ineffective signals. Signals taking two different values to call same operation are considered ineffective, because the operation will be called whatever the value of the signal is.

Also the changing of signal values accompanying calling any of the operations is inserted in the operation definition inside the included machine. Then those inserted signals are examined to delete those that appear with two different assignments.

This flattening step is very important to cross the gap between the way an included machine is expressed and handled in B models and the design of hardware components in and the connection of components VHDL

The B machine starts with the word "MACHINE" followed by the machine name; the machine name is used as the entity name in the corresponding VHDL code.

MACHINE mname(Name) → entityname = Name

The VARIABLES clause in B machine defines the names of the variables which are mapped into port names in the VHDL code. The way variables are used in the operations determines the port directions. A variable that is used as condition must be an input port, a variable that is assigned values during operations is an output port, and variables used in both conditions and sometimes assigned values in other operations are defined as input/output ports.

VARIABLES vname(Vname_1, …, Vname_n)
→ ports(Vname_1, Vdirection_1, Vtype_1,…, Vname_n, Vdirection_n, Vtype_n).

Vname == operation_condition → Vdirection = in;
Vname == operation_variable → Vdirection = out;
Vname == operation_x_condition & operation_y_variable → Vdirection = inout.

The SET clause is mapped into an enumerated type in VHDL with the same set name as the type name and the set values as the values the type can take; also a signal is defined with the enumerated type to represent the states of the machine.

Setdef(Setname, setvalue_1,….,setvalue_n) →
enumerated_type_name =Setname,
enumerated_type_value(setvalue_1,….,setvalue_n),
signal_name_1 = state,
signal_name_2 = next_state,
signal_type = Setname

Also a process is created in the VHDL code to map the state transitions corresponding to the set values, this process only operates the moving from state to next state or the ending state. The condition of moving to the ending state is left to the user to define it manually in the VHDL code.

```
Setdef(Setname, setvalue_1,….,setvalue_n) →
    state_trans_process(clk = clk +1, "user defined condition end state "->
                                                    next_state =  end state; state = next_state)
```

This process is considered a default process that increments the clock and moves the machine to the next state. As explained in the previous chapter the INVARIANT clause is used to define the types of the variables and also defines the properties that must be satisfied by the model. The verification process ensures that the properties defined by the INVARIANT clause are maintained by the machine operations in all system states, so during the implementation there is no need to map those properties somewhere in the VHDL code as they are mapped implicitly during inside the operations. So the INVARIANT clause is used only for mapping the types of the variables into the types of the ports in the VHDL code.

A bool type variable is mapped into std_logic type. If a variable type is an integer or the set name then it is mapped into std_logic_vector with undefined size, the designer has to define it manually.

```
Vtype == bool → VtypeI = std_logic;
Vtype == INTEGER → VtypeI = std_logic_vector(x to y);
Vtype == Setname → VtypeI = std_logic_vector(x to y).
```

The INITIALIZATION statement in a B machine is handled by a synchronous reset process in the VHDL code.

```
INITIALIZATION (Vname, :=, Vinit) →      reset_process(reset = '1')->  Vname, <=, Vinit
```

The main part of the code is in the OPERATIONS clause which can take different ways according to the described system. The clause starts with the keyword OPERATIONS, then each operation starts by an operation name followed by an equal sign then the statements of the operations. A process is created inside the architecture body of the VHDL code to include the mapping of the operations statements, the process will be labeled by the operations.

Operations in the B machines are used to change the state of the system. They assign values to the machine variables depending on certain conditions. The operations begin with an operation name followed by an equal sign then the statements. Some operations call some functions or operations defined in the included machines, these calls are enclosed by the keywords "BEGIN" and "END". In most cases the operations are made of conditional statements. Conditional statements in B are described using one of the keywords "IF", "SELECT", "PRE". The three statements have approximately the same structure, so we will call either of them conditional statement. All the conditional statements are mapped into "if" statements in VHDL, the designer can convert "if" statements into case statements manually when needed.

```
conditional_statement(condition1, statement)
        → if_statement (Vcondition1, Vstatement);

conditional_statement(condition1, statement, else_statement)
        → if_statement (Vcondition1, Vstatement, Velse_statement);

conditional_statement(condition1, condition2, statement)
        → if_statement (Vcondition1, Vcondition2,Vstatement);

conditional_statement(condition1, condition2, statement, else_statement)
        → if_statement (Vcondition1, Vcondition2,Vstatement, Velse_statement);

conditional_statement(condition1, condition2, condition3, statement)
        → if_statement (Vcondition1, Vcondition2, Vcondition3, Vstatement);

conditional_statement(condition1, condition2, condition3, statement, else_statement)
 → if_statement (Vcondition1, Vcondition2, Vcondition3,Vstatement, Velse_statement).
```

The above figure describes how the mapping technique handles different forms of conditional statements in B machines and their corresponding VHDL mapping. Nesting and multiple conditions (up to 3 conditions for simplicity) are allowed in the shown conditional statements. The condition part can be either conjunction or disjunction of conditions.

The condition part in any conditional statement consists of a condition_variable that is one of the machine defined variables, followed by a relational operator then a condition value. The condition is mapped to the corresponding code part in VHDL.

```
Condition(Cond_variable, rel_op, cond_value)
                    → V_Cond_variable, mapped_rel_op, V_condvalue.

Cond_variable → Vname_n.
```

Condition value can be either a Boolean value(true or false), a digit, or one of the machine defined set values.

```
cond_value(true) → V_condvalue('1');
cond_value(false) →V_condvalue('0');
cond_value(digit) → V_condvalue(digit);
cond_value(setvalue_n) → V_condvalue(setvalue_n);
```

The statement part of the conditional statement may be a simple statement assigning values to one of the defined machine variables, or a nested statement introducing new conditional statements. The statement may be a conjunction of two or more statements.

```
Statement (simple_statement) → Vsimple_statement;
Statement (simple_statement1, simple_statement2) → Vsimple_statement1, Vsimplestatement2;
Statement (nested_conditional_statement) → Vnested_conditional_statement.
```

A simple statement consists of a statement variable that is one of the machine defined variables, followed by the assignment operator, then the assigned value.

```
Simple_statement(stat_variable, assign_op, stat_value)
                    → Vstat_variable, Vassign_op, Vstat_value)
```

```
stat_variable (Vname_n)→ Vstat_variable(Vname_n).
```

The assigned value is one of three options: the first option is to be one of the set values defined in the B machine, the second option is to be a numerical value, and the third one is to be a mathematical operation on one of the machine defined variables.

```
stat_value (setvalue_n) → Vstat_value( setvalue_n);
stat_value (N) → Vstat_value(N);
stat_value ( Vname_n, m_op, N) → Vstat_value( Vname_n, m_op, N).
```

The nested conditional statement is similar to the original one, but it allows an "elseif" part.
Also multiple conditions (conjunction or disjunction of conditions) are allowed in the nested conditional statement.

```
n_conditional_statement(condition, statement)
        → if_statement (Vcondition, Vstatement).
n_conditional_statement(condition, statement, elseif_condition, else_statement)
        → if_statement (Vcondition, Vstatement, Velseif_condition, Velseif_statement).
n_conditional_statement(condition, statement, elseif_condition1, elseif_condition2, else_statement)
        → if_statement (Vcondition, Vstatement, Velseif_condition1, Velseif_condition2,
Velse_statement).
n_conditional_statement(condition1, condition2, statement)
        → if_statement (Vcondition1, Vcondition2,Vstatement).
n_conditional_statement(condition1, condition2, statement, elseif_condition, else_statement)
        → if_statement (Vcondition1, Vcondition2, Vstatement, Velseif_condition,  Velseif_statement).
n_conditional_statement(condition1, condition2, statement, elseif_condition1, elseif_condition2,
else_statement)
```

```
→ if_statement (Vcondition1, Vcondition2, Vstatement, Velseif_condition1, Velseif_condition2, Velse_statement).
```

Operators are mapped from B machines to the corresponding VHDL operators. There are four groups of operators; the first is the group of relational operators used in conditions (equal, not equal, greater than, less than, greater than or equal, less than or equal). Some of the relational operators of B and VHDL use Whenever the "INCLUDES", and "SEES" statements appear in a B machines, this means that the corresponding VHDL implementation should includes some components. Each included or seen machine is mapped into a component, and the including machine will contain the code connecting all included or seen components.

```
INCLUDES inc_mnames(inc_name_1, … , inc_name_n)
                → component_1 = inc_name_1, …, component_n =inc_name_n.
```

```
SEES seen_mnames(seen _name_1, … , seen _name_n)
                → component_1 = seen _name_1, …, component_n = seen _name_n.
```

C. Tool structure

In this work a tool was developed to automate the mapping of verified B models into VHDL implementations. The tool scans a text file containing the B machine and gives out the corresponding VHDL that in some cases will need some adjustments from the designer. The tool starts by converting the input file into a list of words. This list of words is then checked against the predefined grammar rules of B machines. The grammar rules of B are defined using DCG rules in SWI prolog. The input B file should follow the described rules exactly, any mistake will cause the tool to fail and stop. The next step is to extract and adjust the variables that will be used in writing the VHDL code. The last part of the tool is the writer function that follows the defined VHDL syntax and writes out the VHDL file using extracted and adjusted variables. In this section we are going to describe the tool structure in details. Figure 2 shows the structure of the developed tool.

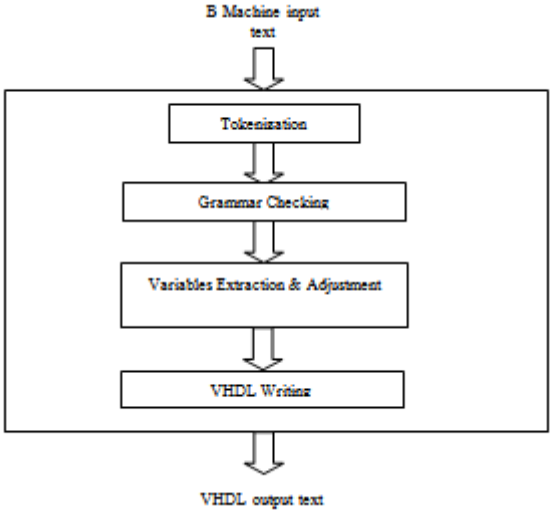

Fig.2: Structure of the developed tool

1) B machine tokenization:

This part of the tool simply scans the input file and gives out a list of words separated by commas. The part is built up using a prolog function that scans the file till its end ignoring new line separators. The space marks the beginning of a new token. Part of the prolog function is shown below.

```
parse:-  see('bin.txt'), read_line(Lines),  write_words(Lines), seen,  !.
read_word(C, [], C):- space(C), !.
read_word(C, [], C) :- newline(C), !.
read_word(Char, [Char|Chars], Last):- get0(Next),     read_word(Next, Chars, Last).
```

2) B grammar Checker:

Any language can be considered as a set of sentences or strings of finite length [20]. To define a new, or describe an existing one – either a natural or a programming language – the specification ought to include

only well-formed strings. Much formalism has been proposed to facilitate systematic descriptions of languages, remarkably the formalism of context free grammars (CFGs). A CFG is built from production rules describing the semantics of the described language. Many Prolog systems utilize a particular syntax for language specifications called Definite Clause Grammars (DCGs) [20]. When such a description is encountered, the system automatically compiles it into a Prolog program. DCGs are generalizations of CFGs.

In our tool we the B grammar was described using DCG rules in SWI prolog. The left hand side of the rule denotes the name of the statement that is being defined in the right hand side. Figure shows the DCG rule for the condition part of the conditional statement. The right hand side of the rule states that the condition consists of a condition variable followed by a relational operator then a value. The next part of the figure shows the definition of the condition value using three rules with the same left hand side. This indicates that the condition value may be one of three options, a boolean value (true or false), a digit or a set value.

```
cond(CondVname,Condvalue,R) -->condition(CondVname),eq(R), condvalue(Condvalue).
condvalue(Bvalue)--> bvalue(Bvalue).
condvalue(Digit)-->digit(Digit).
condvalue(CSetvalue)-->csetvalue(CSetvalue).
```

Another DCG rule is used to describe the statement part of the conditional statement. It starts by one of the variable names then the assignment operator in B ":= " followed by the statement value. The rule describes the conjunction of two statements.

```
simplestat(Svname,Var,Digit1,Digit2,Svnamestat2,Varstat2,Digit1stat2,Digit2stat2)-->
svname(Svname),col,eqs,statvalue(Var,Digit1,Digit2),(and,simplestat2(Svnamestat2,Varstat2,Digit1stat2,Digit2
stat2);!).
```

The statement value is then described to be one of three options: a set value, numerical value, or a number added to one of the machine variables.

```
statvalue(Var,Digit1,Digit2) -->csetvalue(Var);(digit(Digit1);(svname(Var),plus,digit(Digit2) )).
```

3) Extracting and adjusting Variables:

This part of the tool is concerned with the variables that are required in writing the VHDL code. DCG rules in prolog allows variable passing. The variables needed in writing the VHDL code are passed to the writing function through DCG rules.

The machine name is passed to the writing function through the variable "Name".

```
mname(Name) -->[Name], {mname(Name) }.
```

Similarly for the variable types.

```
vtype1(Vtype1) -->[Vtype1], {vtype1(Vtype1) }.
```

But the variable types need some adjustments before being used by the writing function. A boolean type is converted to std_logic, integer and set variables are converted into std_logic_vectors. The size of the vector is left to the designer to determine it manually.

```
(Vtype1 == bool -> Vvtype1 = std_logic ;(Vtype1 == INTEGER -> Vvtype1 = std_logic_vector(x to y); Vtype1
== Setname -> Vvtype1 = std_logic_vector(x to y)))
```

Another example for variables that needs adjustments is the boolean condition values, true is converted into '1' and false into '0'.

```
Condvalue1=="TRUE" -> Condvalue1 = '1'; Condvalue1=="FALSE" ->  Condvalue1 ='0'
```

4) VHDL writer:

The VHDL writer is the last stage, it uses the variables passed to it and follows the VHDL defined rules to write the final code. The write predicate which is a predefined predicate in prolog is used. A text file is created and the lines of code start to be written.

```
tell('ToFile2.txt'),
        open(File, append, Stream,[type(text)]),
```

First the machine name is inserted as an entity name

```
write(Stream, ('entity ')),
write(Stream, Name),     write(Stream,' is'), nl(Stream),
```

Then the variable names are written as ports with their type and direction according to the VHDL syntax.

```
write(Stream, ('port(')),
        write(Stream, Vname1),
        write(Stream, (': ')),write(Stream, Vdirection1),          write(Stream, (' ')),write(Stream,
Vvtype1),write(Stream, ('; ')),
```

Next the architecture part is written.

```
write(Stream,('architecture behavioral of ')),write(Stream, Name),write(Stream,(' is )),nl(Stream),
```

Type and signal definitions corresponding to “SETS” clause in the B machine are written before the key word “begin” of the architecture.

```
write(Stream,('type ')),write(Stream,Setname),write(Stream,(' is (')),write(Stream,Setvalue1),write(Stream,(' ,
')),write(Stream,Setvalue2),          write(Stream,(' , ')),write(Stream,Setvalue3), write(Stream,(') ; ')),nl(Stream)
        ,write(Stream,(' Signal state, next_state : ' )),write(Stream,Setname),write(Stream,(';')),nl(Stream),
```

It should be noted that variables are used by the writing functions to indicate the existence of their statements. That is if a variable is empty it means that its statement is not found. This is used to check for multiple conditions, nested conditional statements and similar variations in the B code.
A part of the operations process is shown.

```
write(Stream,('operations:process(insert the sensitvity list  here)')),nl(Stream),     write(Stream,(' begin ' )),

        nl(Stream),write(Stream,('if')),write(Stream,('
(')),write(Stream,IVnameop1),write(Stream,R1),write(Stream,Condvalue1),write(Stream,(' )')),(IVnameop12=[]-
>!;write(Stream,('and')),write(Stream,IVnameop12),write(Stream,R12),write(Stream,Condvalue12),write(Strea
m,('                                                                                                    )')))),(IVnameop13=[]-
>!;write(Stream,('and')),write(Stream,IVnameop13),write(Stream,(R13)),write(Stream,Condvalue13),write(Stre
am,(' )')))),write(Stream,(' then')),
```

The writing continues in the same way till the end and produces a text file. The designer has to do some adjustments manually like the size of vectors and the processes sensitivity lists. Then this code can be used by synthesis and simulation tools.

V. EXPERIMNETAL RESULTS

Five models were selected as workbenches to be used to apply the conversion from B to VHDL developed in this thesis:
Simplified flight control system Model
Train Gate problem
Soldier Torch problem
Production Cell
Platform screen door controller

A. Case Study 1: Simplified fight control system

The considered flight control system consists of three redundant functions. The system produces periodic signals to the flaps. The system is an asynchronous distributed system, where each function is running on a separate computing device running with its own clock. There is no synchronization between the three clocks. The communication is handled through command messages sent and received over communication channels between the devices. This model was discussed in detail in [21] which was the initial step in this research

B. Case Study 2: The train Gate problem

The train-gate case study is a railway control system which controls access to a bridge for several trains. The bridge is a critical shared resource that may be accessed only by one train at a time. The system is defined as a number of trains (2 in the given simple case) and a controller. A train can not be stopped instantly and restarting also takes time. Therefore, there are timing constraints on the trains before entering the bridge. When approaching, a train sends a appr! signal. Thereafter, it has 10 time units to receive a stop signal. This allows it to stop safely before the bridge. After these 10 time units, it takes further 10 time units to reach the bridge if the train is not stopped. If a train is stopped, it resumes its course when the controller sends a go! signal to it after a previous train has left the bridge and sent a leave! signal. The details of this case study are given in the UPPAAL tutorial [22].

The problem is modeled in three main components train, gate controller, and queue. B machines are used to model each component and verify its operation. Then a root machine is used to model and verify the overall operation using 2 train instants for simplicity.

The verified properties for this system are:

The gate can receive and store messages from approaching trains in the queue.

Train 1 can cross the bridge.

Train 1 can be crossing the bridge while Train 2 is waiting to cross. We check for similar properties for the other trains.

Whenever a train approaches the bridge, it will eventually cross.

The two trains can't cross the gate at the same time

The system is deadlock-free.

Using Atelier B, 36 proof obligations were generated and verified. The FSM and B machines modeling the train, gate controller, queue and the root machine are shown in the following figures.

The proposed technique was used to generate the VHDL implementation for the system components, Simulations were used to check the operation of the model and showed that the verified properties were maintained.

An example of the performed simulations is shown in figure 3 that shows the simulation at the instant when train1 sends a signal to the gate controller indicating it is approaching the gate, as a result of receiving this signal the approaching train id is set to 1 and a counter is started to count the required delay.

Also figure 4 shows that an approaching train is added to the queue and stopped while another one was crossing.

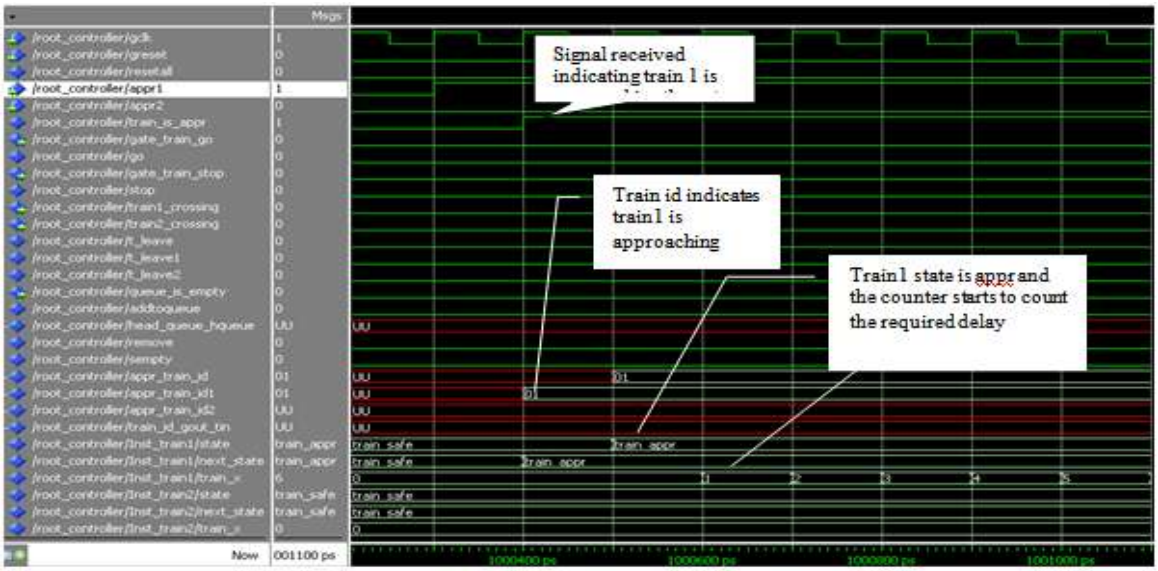

Fig.3: Train 1 approaching

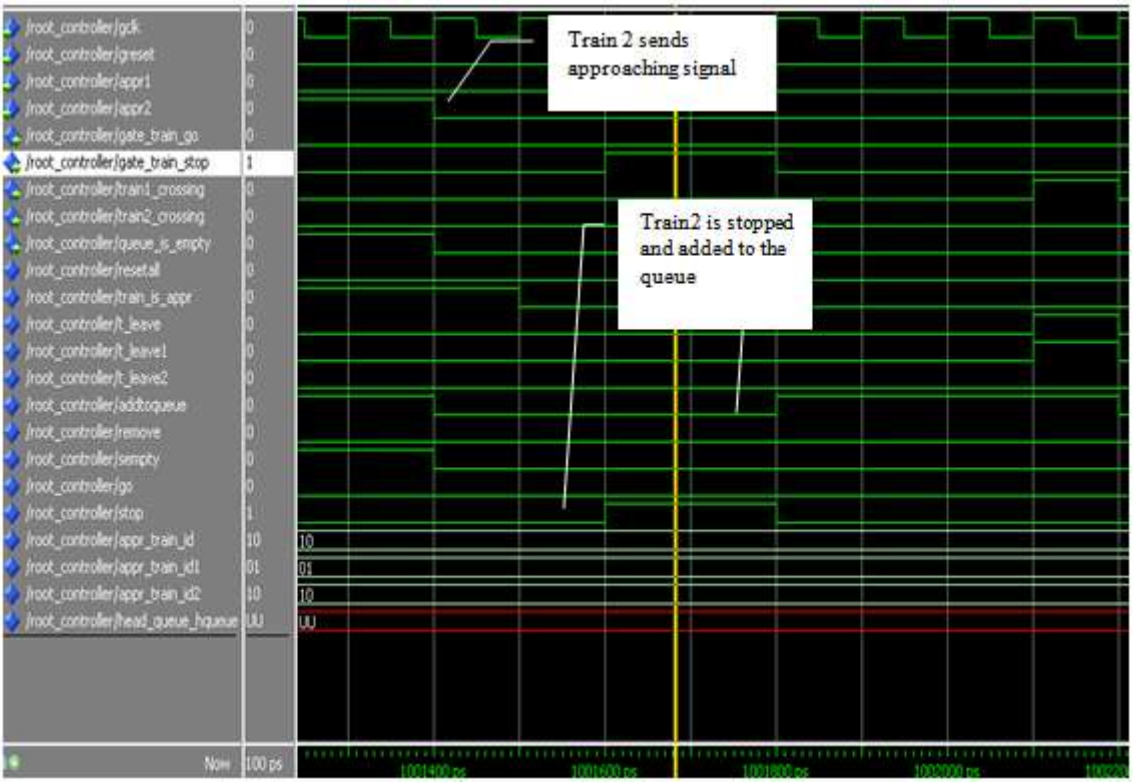

Fig.4: An approaching train is added to the queue and stopped while another one was crossing

C. Case Study 3: The Bridge

In this case study the bridge example is used, which is the famous situation in which we have 2 soldiers and single torch, while each soldier has a different speed. This test case is one of the UPPAAL demo designs.
B machines are used to describe the operation of the soldier and the torch. The root B-machine includes 3 machine instances; one for Torch machine and two for Soldier machine named Viking1 and Viking2. Each Viking has a different time delay; the property that was verified is that the safe state will be entered after the specified delay for the soldier. Atelier B tool was used, 20 proof obligations was generated and verified.
Our proposed technique was applied to generate the corresponding VHDL implementation for the model. Simulation results are used to ensure that the generated VHDL code maintains the verified properties.

The following simulation in figure 5 shows Viking 2 taking the torch and a counter counting the required dealy then Viking 2 enters the safe state.

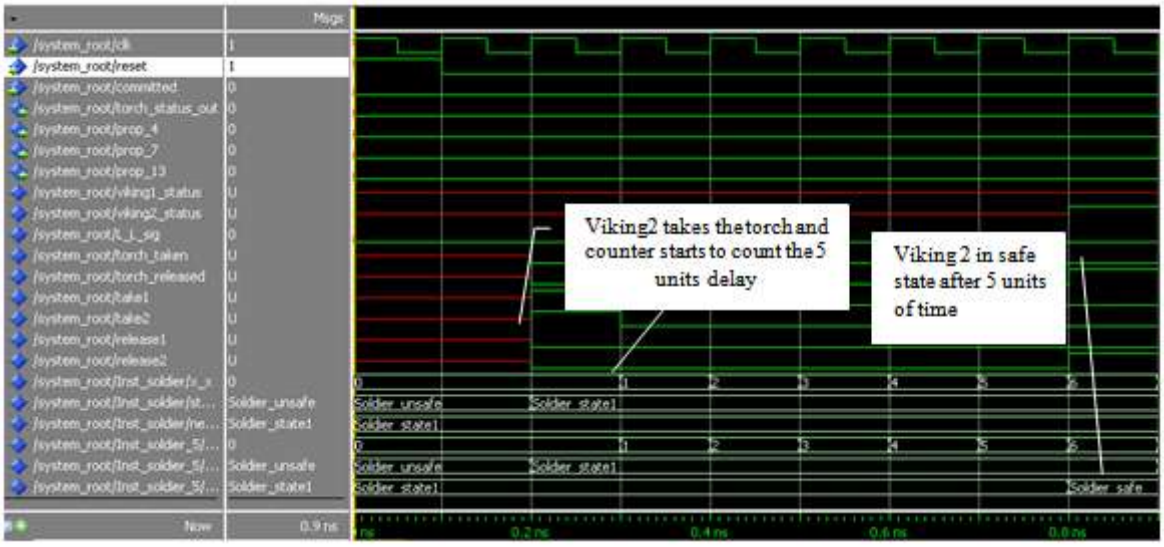

Fig.5: Viking 2 is in safe state after 5 clocks

D. Case Study 4: Production Cell

The production cell case study is an attempt to define a realistic industrial application. It was developed by FZI in Karlsruke as part of the Korso Project [23]. The work in [24] simplified the model and added timing to all operations and requirements. The overall purpose of the cell is to take metal plates from a feed belt, press them and then move them to a deposit belt. Plates are moved by a robot. Arm A of the robot takes a plate from the table which itself must twist and rise when it gets a single plate and places the plate on the press. When the plate has been pressed, Arm B of the robot carries the plate to the deposit belt. The arms of the robot are fixed with respect to each other so the robot controller must coordinate its operations on the 2 arms. A sensor is used to communicate to the robot that a plate has arrived at the cell.

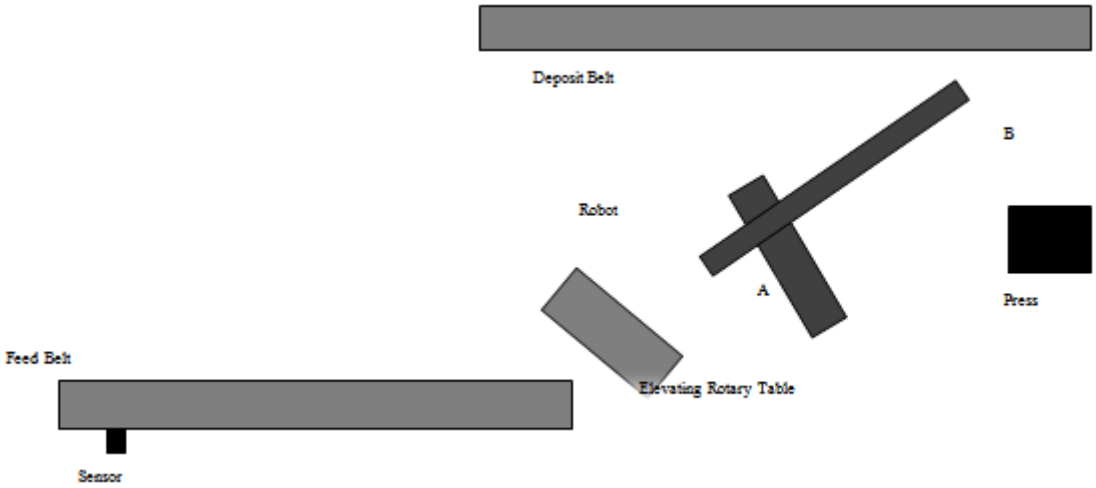

Fig.6: Figure Production Cell

The main property to be verified is the worst case plate traversal time, an assumption is made that whenever the cell is able to accept a further plate, one is available. The six concurrent elements of the production cell and the plate passing through the system are represented by FSMs.

A B machine is used to model each of the six components of the productions cell. Atelier B was used to verify the operation of each component and the time constraints. B model is used to represent the overall operation of the production cell and verify the synchronization of the movement of the plate starting from the feed belt till it reaches the press then is moved to the deposit belt. The temporal values of production cell activities are shown in table 1. 105 proof obligations were generated and proved.

Our proposed technique is then used to generate the VHDL implementation for each of the cell elements. Simulations were used to ensure the correct operation at certain time instants. The following figures show the simulation results for the robot.

Table II: Description of timing for production cell elements

Device	Description	Time units
FEED BELT	Move to sensor	3
FEED BELT	Move to table	1
TABLE	Raise and twist	2
TABLE	Return and twist	2
PRESS	Press plate	22-25
PRESS	Return ready for new plate	18-20
DEPOSITBELT	Move plate out of cell	4
ROBOT	To press	5
ROBOT	Turn 90	15
ROBOT	To deposit belt	5
ROBOT	From conv to table	25
ROBOT	From conv to wait position	22
ROBOT	From press to wait position	17
ROBOT	From wait position to table	3
ROBOT	From wait position to press	2
ROBOT	At wait position	2

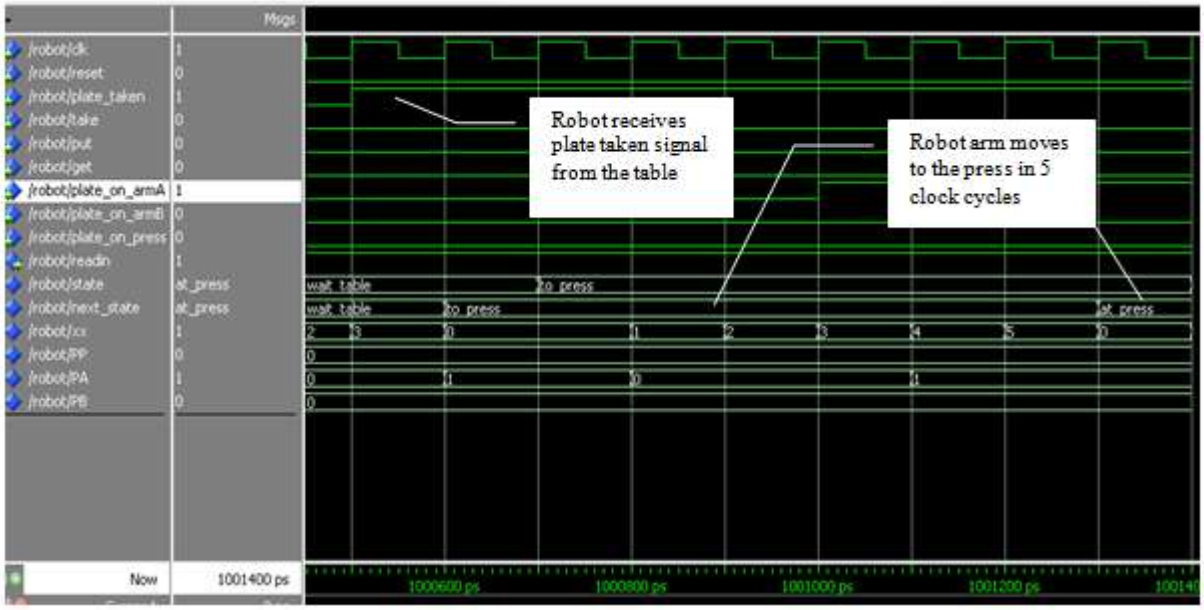

Fig.7: Robot model

The simulation in figure 7 shows the robot model when plate taken signal is received from the table it moves the arm to the press in 5 clock cycles then resets the counter.
After reaching the press the robot turns 90 degrees, this takes 15 time units as shown in figure 8.

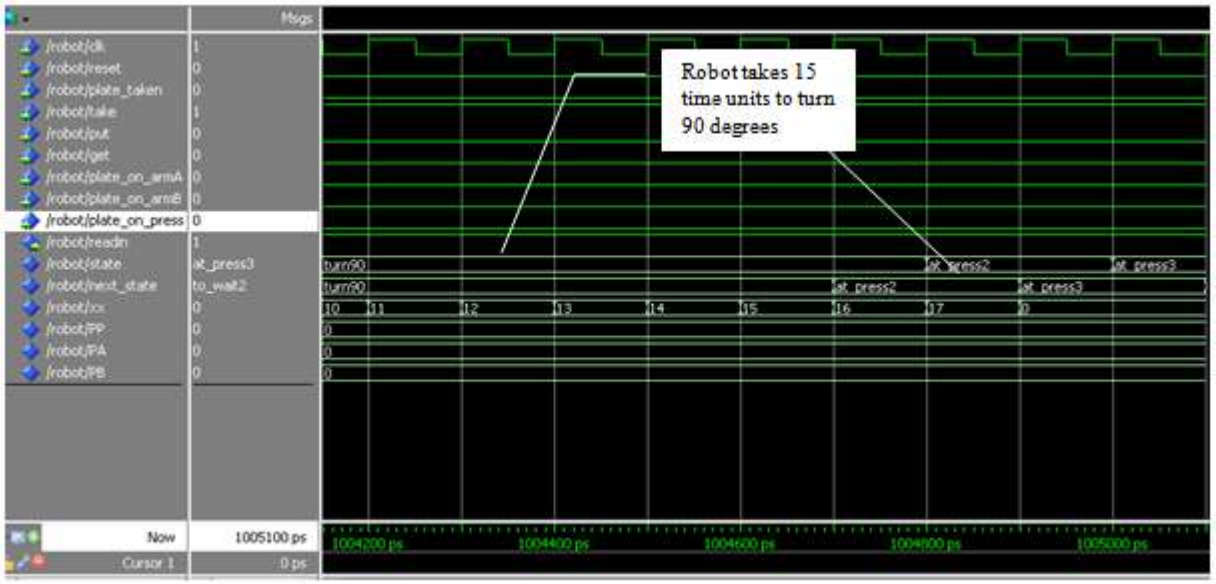

Fig.8: The robot turns 90 degrees in 15 time units

E. Case Study 5: Platform screen door controller

This application was presented in [25] as one of the application of the B formal method to the development of safety critical systems. In France, Platform screen doors (PSD) were used for years to prevent passengers to enter or to fall on tracks. Such a system was adopted by the METEOR driverless metro, as it dramatically improves trains availability. In order to offer higher quality services and more safety to its passengers, the public transportation operator was trying to introduce this kind of protection system in several lines, automated or not.

ClearSy was in charge of developing the control command controller that can detect the arrival, presence at a standstill and departure of trains without direct connection with them. Once the train is at standstill, the controller should be able to detect train doors opening and closing, and then issue PSD opening and closing orders. These orders have to be securely issued (failure by elaborating a wrong opening order may lead to passengers' injury or death).

In order to reach the required safety level during project timescale, it was decided to set up a development method reaching targeted reliability, and also ensuring traceability between the different stages of the projects in order to reduce the validation effort. This method was heavily based on the B formal method, and applied during most phases of the project.
The B method was used to:
- Verify on the overall system (PSD + controller) that functional constraints and safety properties were verified (no possibility to establish forbidden connections between train and platform or between train and tracks).
- Lead to the observation of dangerous system behaviour.

A new architecture was proposed, making use of usual sensors and processing based on temporal sequence recognition of sensor events. Hyper frequency, infrared and laser sensors help to improve system resistance to various perturbations. Redundancy among sensors using different technology raises measures confidence. These sensors were positioned on the platform and pointed to the tracks in order to measure train position, train speed and train door movements.

System and software specification were then formalized in B by the development team, taking into account only nominal behaviour for the sensors (in absence of perturbation). Models obtained from previous functional analysis (independent from any PSD controller architecture) were directly reused. The proposed architecture was modeled and inserted in these previous models. New architecture was successfully checked by proof to comply with functional specification of the system, including parts of the French underground regulations. Controller functions were then precisely modeled (train arrival, train detection, train departure, train door opening, train door closing, etc).
More details about the project are available in [25].
Our proposed technique was applied to the B machine of the PSD controller to develop the VHDL implementation.

Simulation was performed to show that the generated VHDL code maintains the verified properties. The following figures explain the behavior of the developed PSD controller.

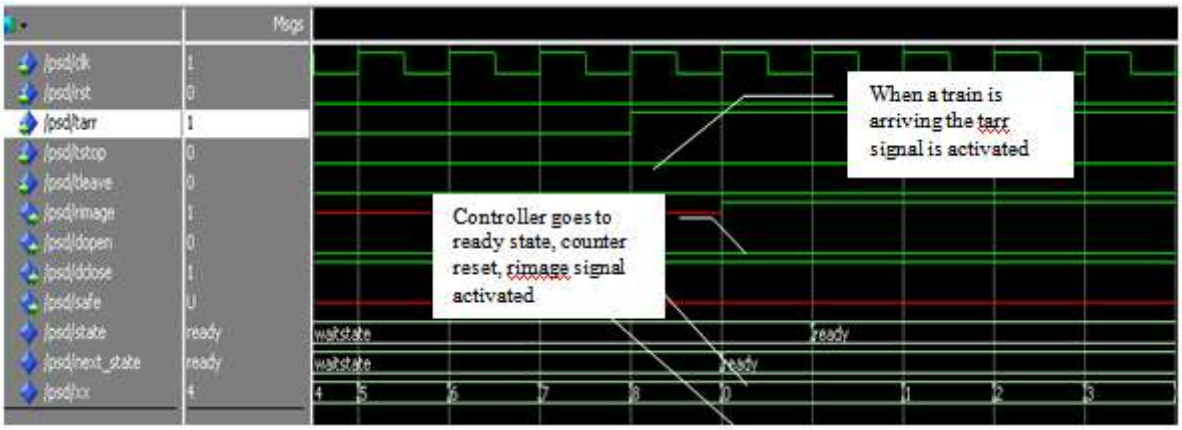

Fig.9: Going to the ready state

When a metro is arriving a signal is received by the controller so it moves to the ready state as in figure 9, and then starts gathering images. rimage signal is activated to start capturing images and the counter is rset to count the 50 time units figure 10.

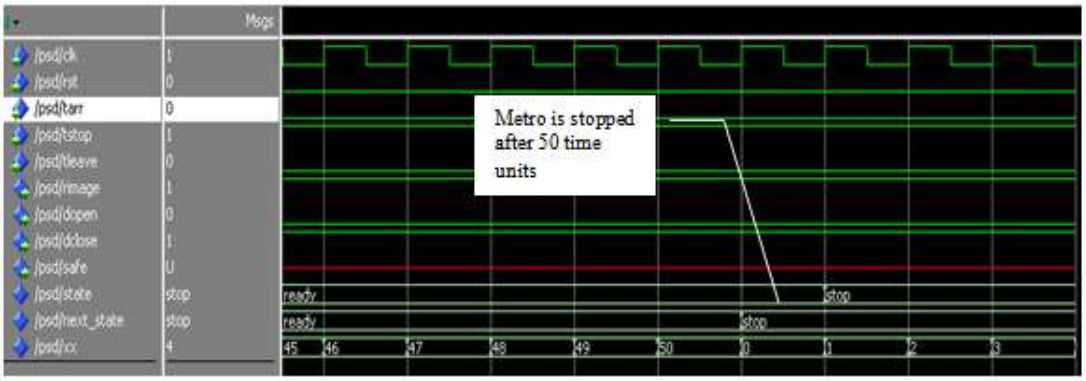

Fig.10: Metro stop

After the 50 units passed and the stop signal is received indicating that the metro arrived and is standstill the images are used to ensure the position of the doors, the controller state is moved to the stop state the counter is reset to count 5 units then issues the door opening signal

VI. CONCLUSION

The most common procedure to ensure the reliability of a design is simulation. Unfortunately simulation cannot fully inspect all the execution states of the system. The significant increase in the complexity and size of digital systems together with the nature of real time systems boosted up the need for a different approach for the validation of the behavior of a system in the early design stages. Formal verification is an approach to validate a system by formally reasoning the system behavior. In formal verification the system implementation is checked against the requirements or the properties to be satisfied.

B method is one of the common paradigms used in formal verification. B method offers a strong verification domain as it is based on a mathematical and logical approach. The proof obligations (properties that must be satisfied) are automatically generated from the model; also the available tools provide both automatic and interactive proofs. B-Method gives a formal description of systems; Verification is performed through theorem proving avoiding state explosion problem in Model checking. Proof obligations are deduced automatically from the model.

VHDL is a mature implementation domain where many synthesis and simulation tools are available.

It is required to reach an approach for developing “correct-by-construction” designs to enhance the long and expensive design cycle used in the “construct-by-correction” design approach.

The developed technique employed the strong verification framework offered by the B method together with the premature VHDL implementations to achieve “correct-by-construction” designs. We reached a direct conversion of the verified B models into implementation which avoids losing any of the verified properties.

Although the semantics of B is completely different than that of VHDL, and not all constructs of B are available in VHDL, the work in this thesis provided a technique to cross the challenging gap between the B and VHDL. Developing VHDL implementations directly from a verified B machines maintaining the verified specifications of the B model.

The proposed technique was applied on five popular models used as benchmarks: Simplified flight control system Model, Train Gate problem, Soldier Torch problem, Production Cell, Platform screen door controller. B machines for the models were verified using Atelier B tool. The Atelier B automatically generates and proves the proof obligations to ensure that the system requirements are satisfied. The proposed technique was used to generate the VHDL codes for the designs. The generated VHDL codes were synthesized on the Xilinx ISE tool and simulations were performed using ModelSim simulator. Simulation outputs showed that all the verified properties of the B models were maintained in the implementation. This proves that the proposed technique added the B and VHDL domains in a complete design cycle for a “correct-by-construction” approach.

There are two future directions for this work: The developed tool can include a larger subset of the B constructs, and allow more automation for the nesting of conditions in the input B machine, detecting the final state of the system, and determining the size of vectors for input and output ports and signals in the generated VHDL code.

The generated VHDL code was checked against the verified properties by simulation at some critical points. The developed tool can be integrated with a model checking tool to provide more confidence that the implemented system maintains all the verified model properties.

REFERENCES

[1]. Campos, J. C., Machado, J., and Seabra, E., "Property patterns for the formal verification of automated production systems", in Proceedings of the 17th World Congress, The International Federation of Automatic Control, pp. 5107-5112 , Seoul, Korea, 2008.

[2]. M. C. McFarland, "Formal Verification of Sequential Hardware: A Tutorial", IEEE Transactions on Computer-Aided Design of Integrated circuits and systems, pp. 663-654, 1993.

[3]. Roussel, J.-M and Denis, B., "Safety properties verification of ladder diagram programs", Journal Europeen des Systemes Automatisees, no. 36, pp. 905-917, 2002.

[4]. Smet, O. D. and Rossi, "Verification of a controller for a flexible manufacturing line written in ladder diagram via model-checking," in 21th American Control Conference, ACC'02, vol. 5, pp. 4147 - 4152, 2002.

[5]. O. Rossi, Validation formelle de programmes ladder pour automates programmables industriels, France: Ecole Normale Superieure de Cachan, 2003.

[6]. Gaid, M., B_erard, B. and Smet, "Verification of an evaporator system with uppaal", European journal of Automated Ayatems, vol. 9, no. 39, pp. 1079-1098, 2005.

[7]. Machado, J., Denis, B., and Lesage, Machado, "A generic approach to build plant models for DES verification purposes", in 8th International Workshop On Discrete Event Systems (WODES'06), pp. 407 – 412, 2006.

[8]. J. Bengtsson, W. O. D. Griffoen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and Y. Wang, Verification of an audio protocol with bus collision using UPPAAL, vol. 1102, Heidelberg, Germany: Springer-Verlag, 1996, pp. 244-256.

[9]. J. Romijn, "A timed verification of the IEEE 1394 leader election protocol", in 4th Int. ERCIM Workshop Formal Methods for Industrial Critical Systems (FMICS'99), pp. 3-29, 1999.

[10]. Stoelinga, D. P. L. Simons and M. I. A., "Mechanical verification of the tree 1394a root contention protocol using uppaal2k", Nijmegen, the Netherlands, pp. 509 -531, 2000.

[11]. T. Stauner, O. Müller, and M. Fuchs, Using HyTech to verify an automotive control system, vol. 1201, Heidelberg, Germany: Springer- Verlag, 1997, pp. 139-153.

[12]. T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: technology transfer of formal methods inside Microsoft", in Integrated Formal Methods Conference, Canterbury, U.K., pp. 1-20, 2004.

[13]. Anaheed Ayoub, Ayman M. Wahba, M. Sheirah, Analyzing safety-critical real-time systems, PhD thesis, Faculty of Engineering, Ain Shams University, Egypt 2010.

[14]. Lam, William K., "Hardware Design Verification: Simulation and Formal Method-Based Approaches", Prentice Hall, 2005.

[15]. C.-J. H. Seger, "An Introduction to Formal Verification", Technical Report 92-13, UBC, Department of Computer Science, Vancouver, B.C., Canada, June 1992.

[16]. J. Harrison, Theorem Proving for Verification, tutorial, Portland: Intel Corporation, 2008.

[17]. Craigen, D., Gerhart, S., Ralston, T.J., "An international survey of industrial applications of fromal methods", Atomic Energy Control Board of Canada, U.S. National Institute of Standards and Technology, and U.S. Naval Research Laboratories, 1993.

[18]. Stavridou., J. P. Bowen and V., "Safety-critical systems, formal methods and standards", IEE/BCS Software Engineering Journal, vol. 8, no. 4, pp. 189-209, 1993.

[19]. Lybrand, Coopers, "Safety related computer controlled systems market study", HMSO, Review for the Department of Trade and Industry, London, UK, ISBN-10: 0115153152, 1992.

[20]. [20]. MaÃluszynski, Ulf Nilsson and Jan, LOGIC, PROGRAMMING AND PROLOG (2ED), John Wiley & Sons Ltd, 2000.

[21]. El-Araby, N.A., Wahba, A.M., Taher, M.M., "Implementation of Formally verified Real Time Distributed Systems: Simplified Flight Control System", in Proceedings of the International Conference on Computer Engineering and Systems (ICCES'2011), pp. 25 - 32 , 2011

[22]. Gerd Behrmann, Alexandre David, Kim G. Larsen, A Tutorial on Uppaal, Denmark: Department of Computer Science, Aalborg University, 2004.

[23]. S. Tyszberowicz, "How to implement a safe real-time system: To observe implementaion of the production cell case study," Real-Time Systems, vol. 15, no. 1, pp. 61-90, 1998.

[24]. A. Burns, "How to Verify a Safe Real-Time System. The Application of Model Checking and a Timed Automata to the Production Cell Case Study", University of York - Department of Computer Science, Heslington, York, England, 1998.

[25]. Thierry Lecomte, Thiery Servat, Guilhem Pouzancre, "Formal Methods in Safety-Critical Railway Syatems", 10th Brasilian Symposium on Formal Methods, Clearsy, Aix en Provence, France, 2007.