Formal Verification of a Protocol Converter Memory Manager using FormalCheck

Jounaidi Ben Hassen and Sofiène Tahar

Electrical and Computer Engineering Department

Concordia University, Montreal, Canada

Email: {jounaidi, tahar}@ece.concordia.ca

Technical Report

April 2003

Abstract

We present in this report the formal specification and verification results of the Memory Manager block of a System-on-a-Chip (SoC) platform Protocol Converter using the model checking tool FormalCheck. The Memory Manager represents the main block of the protocol converter system and is made of five modules, namely, a Memory Manager Controller, an Address Counter Register, a Data Counter Register, a Packet Counter Register and a Packet Assembler. First, we extracted some constraints to define the environment for the Memory Manager. Then we specified a number of relevant liveness and safety properties expressible in FormalCheck and accomplished their verification under the defined set of constraints. Through extensive verification, we have been able to find a number of bugs in the design that were omitted by simulation. This experience demonstrates the usefulness of formal verification techniques to complement traditional verification by simulation.

1 Introduction

The increasing complexities of hardware designs have made verification and error detection on the critical path of the design process. Moreover, some of these errors may, sometimes, cause catastrophic loss of money, time, or even human life. A major goal of designers is to construct systems that operate reliably despite their complexity. The same complexities, however, are responsible for the inability to achieve an integral verification of the design, and thus to gain a high degree of confidence in its correctness. This is in particular a serious problem for System-on-a-chip (SoC) designs which may contain processor cores, custom logic, memory and IP (Intellectual Property) blocks on a single chip.

Traditionally, simulation is used to verify the "correctness" of a design, however, it is no more able to keep pace with the increasing complexity of hardware designs. In fact, as an integrated circuit functionality increases linearly, the amount of vectors required to fully test this functionality increases exponentially. It therefore becomes impossible for a human being to fathom all vectors required to fully test a circuit. To overcome this difficulty, formal hardware verification methods [9] are now being deployed, and became useful tools for detection of functional design errors. By using formal verification in parallel with the design efforts, the overall design cycle can be reduced by insuring a maximum design coverage, while maintaining a high degree of confidence in the verification result. For instance, the objective of formal verification is to verify that the design model conforms to an abstract specification, consisting of a set of properties which together describe partially the intended functionality of the design. Those techniques have proven their efficiency to formally verifying industrial size systems.

This report aims to explore a case study on the model checking of the Memory Manager component of an SoC platform Protocol Converter System and to show that formal methods are strong enough for the verification of such complex design. The Memory Manager of the Protocol Converter System [6], designed by the "Groupe de Recherche en Micro-électronique" at the École Polytechnique de Montréal, was chosen to be verified as a research case to experiment the benefits of formally verifying a design using model checking over using simulation to find bugs inside a design. The model checking is based on the FormalCheck tool of Cadence [5]. The architecture of the Memory Manager is described at the Register-Transfer Level (RTL) coded in VHDL (VHSIC Hardware Description Language). The Memory Manager represents the main block of the protocol converter system and is made of five modules, namely, a Memory Manager Controller, an Address Counter Register, a Data Counter Register, a Packet Counter Register and a Packet Assembler. In a first step, we extracted some constraints to define the environment for the Memory Manager. Then we specified a number of relevant liveness and safety properties expressible using FormalCheck templates and accomplished their verification under the defined set of constraints. Through extensive verification, we have been able to find a number of bugs in the design that were not caught by simulation. This experience demonstrates the usefulness of formal verification techniques to complement traditional verification by simulation.

The rest of the report is structured as follows: In Section 2, we give a brief overview of the Formal verification techniques and we present the tool FormalCheck and its verification options. In Section 3, we describe the behavior of the Memory Manager and its interaction with the other components of the Protocol Converter System. In Section 4, we define the required set of constraints on the Memory Manager. In Section 5, we specify the set of properties to be verified. In Section 6, we present our results of the formal verification using model checking. We discuss some related work in Section 7 and conclude the report in Section 8.

2 Formal Verification Techniques

Formal verification techniques are based on the application of mathematical reasoning to the specification and validation of systems to insure the correctness of designs [9]. There are mainly two techniques of formal verification, namely *theorem proving* and *model checking*. Although both of these approaches are used to analyze a system for desired properties; there are many differences between them.

Theorem proving is an approach where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas.

Model checking is a technique that relies on building a model of a system and checking that a desired property holds in that model by exploring a state space search in that model. Model checking is mainly used in hardware and protocol verification. Two general approaches to model checking are used in practice today. The first, temporal model checking, is a technique developed in the 1980s by Clarke and Emerson [7] and by Queille and Sifakis [14]. In this approach specifications are expressed in a temporal logic [13] and systems are modeled as finite state systems. An efficient search procedure is used to check if a given finite state transition system is a model for the specification. In the second approach, the specification is given as an automaton; then the system, also modeled as an automaton, is compared to the specification to determine whether or not its behavior conforms to that of the specification.

In contrast to theorem proving, model checking is completely automatic and fast. It can be used to check partial specification. Thus, it can provide information about a system's correctness even if the system has not been completely specified. In addition, it produces counterexamples, which usually represent residual errors in design, and so can be used to aid in debugging.

In the work presented here, we have chosen model checking as formal verification method based on the Cadence tool FormalCheck [5]. The choice of FormalCheck is based firstly on the simple fact that this is a mature and commercial tool, and secondly that has a suitable user-friendly interface, where properties can be expressed intuitively.

FormalCheck

FormalCheck, originally developed at Bell-Labs and now part of Cadence products, supports the synthesizable subsets of Verilog and VHDL hardware description languages. As a model checker, it verifies that a design model exhibits specific behaviors (*properties*) that are required by the design specification. If an error is present, then FormalCheck displays a counterexample as a waveform diagram. Often, a design model is expected to exhibit the stated properties only when components of the model, such as primary inputs, satisfy stated *constraints*. Together, the properties and the constraints that compromise one application of the model checker are called a *query* [5].

Properties that form the basis of a model checker's query fall into two categories: *safety* or *liveness*. Safety properties describe behaviors that can be shown to be false by a finite simulation trace. In FormalCheck, they can be expressed using one of two formats: The *always* format and the *never* format. Liveness properties describe behaviors that are *eventually* exhibited. They cannot be checked with a simulation tool unless one knows the maximum number of steps before the eventuality is fulfilled. Constraints can prevent the model checker from attempting to verify properties with illegal combinations of inputs. Each constraint has a corresponding property and falls into one of two categories: *safety* or *fairness*. A fairness constraint corresponds to a liveness property. Properties and constraints are sometimes given using Boolean expressions and *state variables*. Boolean expressions alone can only define an event that is true at a single point in time. However, the condition to be checked is often the culmination of a sequence of events. Such sequential conditions are defined in terms of state variables.

FormalCheck includes a number of verification modes. Typically, BDDs (*Binary Decision Diagrams*) [4] are used and verification is performed iteratively. Initially, FormalCheck reduces the design model to only needed components for verifying the property. This version is called *one-step reduction*. Sometimes, complex designs require more reduction. In this case, FormalCheck offers an *iterated reduction* algorithm. The iterated reduction algorithm seeks to find a portion of the design model on which it is sufficient to run verification. The portion by construction will be such that if the query is verified on this portion, then it is guaranteed to be true on the original model. Conversely, if the query fails on this portion, then the algorithm expands the error track to an error track of the original one-step reduction model. This operation is called the *reduction seed* and

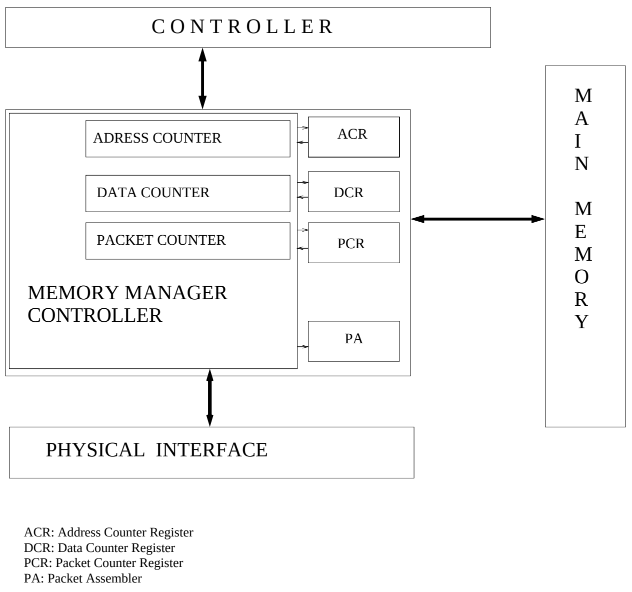

Figure 1: The Memory Manager Architecture

permits the definition of a suitable reduced model for the verification process [5].

3 The Memory Manager Block

The Protocol Converter System [6] is based on a System-on-Chip (SoC) platform. The main advantage of such a plate-form is its flexibility. In fact, its modularity allows the addition, the change or the drop of some modules without affecting the global architecture of the system. This system accepts incoming packets from a physical bus, converts their protocols and then sends them back through the physical interface. The Protocol Converter System is subdivided in three blocks as shown in Figure 1. The first one, formed by the Memory Manager and the Controller, is specialized in the reception of packets and preliminary treatments for the conversion. The second one is specialized in the transmission of converted packets, while the third one is the block which performs the conversion of packets coming from the first block and sends them back to the second block.

Packets come through a physical interface which communicates with the Memory Manager by transmitting data and some control signals. Upon reception, the Memory Manager stores coming data in the Main Memory and transmits to the Controller characteristics of the packet in transmission, including the address of the packet in the Main Memory, the protocol of the packet and its size. The communication between the Memory Manager and the Controller is insured via the Memory Manager Controller. Once the protocol conversion is done, the Controller asks the Memory Manager, via the Memory Manager Controller, to remove the converted packet from the Main Memory. So, the address of this packet and some other control signals are transmitted to the Memory Manager.

The functionality of the Memory Manager is insured by its different components. Besides the Memory Manager Controller, we can find three registers and the Packet Assembler. Intuitively, the Memory Manager Controller is responsible for the communication between the Memory Manager and the Controller; the registers are used as counters of addresses and packets; and the packet assembler is a module that concatenates packet's address and data.

Figure 2 presents the communication between the Memory Manager and the other system

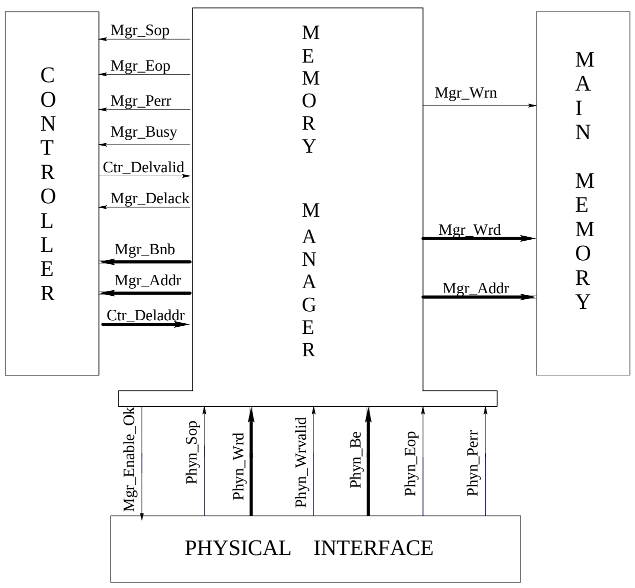

Figure 2: The Memory Manager Block Communication

components. When the Memory Manager is available to receive a packet from the Physical Interface, the signal `Mgr_Enable_Ok` is sampled to high. In this case, data are received by a 32-bit bus called `Phyn_Wrd`. The number of valid bytes is given by the 4-bit bus `Phyn_Be` and the validity of the transmitted word is given by the signal `Phyn_Wrvalid`. The signal `Phyn_Sop`, activated during one cycle, starts the reception of the new packet and the signal `Phyn_Eop`, activated during one cycle, indicates the end of transmission. If an error occurs during the transmission, then the signal `Phyn_Perr` is activated during one cycle.

During the reception of packets, the Memory Manager activates writing in the Main Memory by setting the signal `Mgr_Wrn` to 0. Data are transmitted by the 32-bit bus `Mgr_Wrd`, and the address in which received data are stocked is defined by the 11-bit bus `Mgr_Addr`.

When the Memory Manager begins the reception of a packet, it informs the Controller by setting its signal `Mgr_Sop` to 1 during one cycle and sends it the address of the first word of the received packet via the bus `Mgr_Addr`. At the end of the transmission, the Controller is informed via the signal `Mgr_Eop` which is activated in this case during one cycle. The Memory Manager transmits to the Controller the total size of the received packet via the 11-bit bus `Mgr_Bnb`. If an error occurs during transmission, then the Memory Manager informs the Controller by setting the signal `Mgr_Perr` to 1 during one cycle.

Once the conversion done, the Controller asks the Memory Manager for the suppression of the packet from the Main Memory. If the Memory Manager is available (`Mgr_Busy=0`), then the Controller sends a suppression order via the port `Ctr_Delvalid`. The address of the packet to be deleted is transmitted to the Memory Manager by the 11-bit bus `Mgr_Deladdr`. At the end of the suppression, the Memory Manager sends an acknowledgment to the Controller by activating the signal `Mgr_Delack` during one cycle. Table 1 presents a list of major signals used in the architecture of the Memory Manager to provide control or to transmit data between the different components of the Memory Manager [6].

	Signal	I/O	Size	Description
	`Sys_Clk`	IN	1	System Clock
	`Sys_Rst_N`	IN	1	System Reset
PHY	`Phyn_Wrd`	IN	32	Data bus
	`Phyn_Be`	IN	4	Number of valid bits in the data bus
	`Phyn_Wrvalid`	IN	1	Validity of transmitted word
	`Phyn_Sop`	IN	1	Start reception of a new packet
	`Phyn_Eop`	IN	1	End of transmission
	`Phyn_Perr`	IN	1	Error during transmission
	`Mgr_Enable_Ok`	OUT	1	Availability to receive a packet
MM	`Mgr_Wrd`	OUT	32	Stored data in the Main Memory.
	`Mgr_Addr`	OUT	11	Address bus
	`Mgr_Wrn`	OUT	1	Activates writing in the Main Memory
CTRLLER	`Ctr_Delvalid`	IN	1	Suppression order from the Controller
	`Ctr_Deladdr`	IN	11	Address of packet to be deleted
	`Mgr_Addr`	OUT	11	Address of received packet
	`Mgr_Sop`	OUT	1	Start of packet reception
	`Mgr_Eop`	OUT	1	End of transmission
	`Mgr_Perr`	OUT	1	Error during transmission
	`Mgr_Bnb`	OUT	11	Size of the received packet
	`Mgr_Busy`	OUT	1	Availability for a suppression request
	`Mgr_Delack`	OUT	1	Acknowledgment after suppression

Table 1: Description of Memory Manager signals

4 Environment Constraints

In FormalCheck, primary signals are assumed to be non-deterministic, meaning they could acquire any value within their range on any edge of the clock. However, in most cases correct design operation is allowed on a single edge of the clock. For this reason, properties should be observed using the appropriate clock edge.

In our experiment, we defined the following Constraints as default for all properties used to verify the Memory Manager:

- Constraint 1: We defined a clock constraint on the `Sys_Clk` signal, starting with low for one crank and then high for one crank[1]. In formalCheck, this constraint can be defined as follows:

```
Clock Constraint: System_Clock Signal: Sys_Clk
    Extract:No
    Default: Yes
    Start: Low
    1st Duration: 1
    2nd Duration: 1
```

  Sometimes FormalCheck can report an irrelevant failure caused by input changes that are not on the relevant clock edge. These irrelevant failure can be eliminated by using the clock

[1]A crank in FormalCheck is considered as the propagation delay of a flip-flop. Since there is no concept of absolute time, FormalCheck uses a crank as the unit of time and observes the events relative to this unit.

extraction feature. However, in our case we did not need this feature since the model contains flip-flops with asynchronous preset. Thus, we should desactivate this feature and define manually the system clock. When the *Default* option is activated, the constraint is defined for every property. Since constraints will be used in the verification of all the queries, we activate the *Default* option. *Durations* are given in number of cranks and mean that the signal `Sys_Clk` is sampled to low for one crank (1st duration), then to high for one crank (2nd Duration).

- Constraint 2: This is a reset constraint on the `Sys_Rst_N` signal, starting with low for two cranks and then goes to high forever[2].

```
Reset Constraint: System_Reset Signal: Sys_Rst_N
    Default: Yes
    Start: Low
    Transition  Duration  Value
    Start       2         0
                forever   1
```

Since FormalCheck does not support signal initialization, this constraint is useful to assign initial values to signals. To allow all input signals to take their initial value, the signal `Sys_Rst_N` should be sampled to 0 during one clock cycle (2 cranks), then it is sample to 1 forever.

- Constraint 3: The signals `Phyn_Sop`, `Phyn_Eop` and `Phyn_Perr` are activated during one cycle of the system's clock rising edge (`sys_Clk=rising`). This can be expressed by these safety Constraints:

```
Constraint OneClockSop
    Type: Never
    Assume Never:(Phyn_Sop = 1) and (WasSop = 1)
    Option:Default
```

This constraint means that the signal `Phyn_Sop` can never be activated during two clock cycles, where `WasSop` is a state variable that indicates if the signal `Phyn_Sop` is already activated:

```
WasSOP: Range 0 to 1
    Initial: 0
    if (Phyn_Sop = 1) and (Sys_clk = 0) then
       WasSOP := 1;
    else
       WasSOP := 0;
    end if;
```

The signal `Phyn_Sop` changes value only on the rising edge of system's clock can be expressed by the following safety constraint:

[2]Activities of the Memory Manager block are triggered by an active low asynchronous reset.

```
Constraint SyncSOP
    Type: Always
    Assume Always: (Sys_Clk = rising) or (Phyn_Sop = stable)
    Options: Default
```

Saying that the signal Phyn_Sop changes value only on the rising edge of the system's clock means that if Phyn_Sop changes its value, forcibly Sys_Clk is rising. If it is not the case, Phyn_Sop can only be stable. This is a safety constraint because it should always be verified. Similarly, we define the four other constraints for Phyn_Eop and phyn_Perr as well as the two state variables WasEOP and WasPERR.

- Constraint 4: A constraint that synchronizes the activation of the signal Ctr_Delvalid with the falling edge of the system's clock.

- Constraint 5: A constraint that synchronizes the falling edge of the signal Ctr_Delvalid with the falling edge of the signal Mgr_Delack. This constraint guarantees that the signal Ctr_Delvalid will stand active until the reception of the delete acknowledgment from the Memory Manager.

- Constraint 6: A constraint that synchronizes the activation of the signal Phyn_Be with the rising edge of the system's clock. There is no constraint on the duration of activation.

- Constraint 7: When Phyn_Wrvalid = 1, Phyn_Be should indicate that at least one word is valid in the transmitted packet. This is expressed by the following Constraint:

```
Constraint WrvalidNotPhynBe
    Type: Never
    Assume Never: (Phyn_Wrvalid = 1) and (Phyn_Be = 0)
    Options: Default
```

- Constraint 8: The Physical Interface cannot send a packet while the Memory Manager is receiving another packet. Therefore, two successive Phyn_Sop should be separated by either Phyn_Eop or Phyn_Perr. Reciprocally, the activation of Phyn_Eop or Phyn_Perr should occur to mark the end of a current transmission:

```
Constraint S_EorPerr_S
    Type: Never
    Assume Never: InTransmission = 1 and Phyn_Sop = rising
    Options: Default
```

InTransmission is a state variable that indicates if the Memory Manager is currently receiving a packet from the Physical Interface:

```
InTransmission: Range 0 to 1
    Initial 0
    if Phyn_Sop = rising then
      Intransmission := 1;
    elsif (Phyn_Eop = rising or Phyn_Perr = rising) then
      InTransmission := 0;
    end if;
```

The second constraint can be written as follows:

```
Constraint EorPerr_S_EorPerr
    Type: Never
    Assume Never: (Phyn_Eop = rising or Phyn_Perr = rising) and
    EndTransmission = 1
    Options: Default
```

EndTransmission is a state variable that indicates if the Memory Manager had terminated the reception of a packet from the Physical Interface:

```
EndTransmission: Range 0 to 1
    Initial 1
    if(Phyn_Eop = rising or Phyn_Perr = rising) then
        Endtransmission := 1;
    elsif Phyn_Sop = rising then
        EndTransmission := 0;
    end if;
```

- Constraint 9: We assume that a packet transmission should end. Thus, if the signal `phyn_Sop` is activated, then eventually the signal `phyn_Eop` or `phyn_Perr` is activated. In addition, the length of a packet should be respected. A packet need at least four clock cycles and at most 380 clock cycles to be totally transmitted [6].

```
Constraint AfterSopEopOrPerr
    Type: Eventually
    After: Phyn_Sop = 1
    Assume Eventually: (Phyn_Eop = 1) or (Phyn_Perr = 1)
    Fulfill Delay: 4  Duration: 380
    Of Edge: Sys_Clk = rising
    Options: Default
```

Informally, this constraint means that after the start of a packet transmission, eventually the transmission will end correctly or after an error detection, after at least 4 occurrences and at most 380 occurrences of the rising edge of the system clock.

- Constraint 10: For simplification purposes, we assume that the two signals `Phyn_Eop` and `Phyn_Perr` cannot be activated simultaneously. Thus, the safety constraint can be formalized:

```
Constraint EopAndPerr
    Type: Never
    Assume Never: (Phyn_Eop = 1) and (Phyn_Perr = 1)
    Options: Default
```

- Constraint 11: To simplify further more our verification, we assume that neither `Phyn_Eop` nor `Phyn_Perr` can be activated simultaneously with the signal `Phyn_Sop`. We expressed this constraint as follows:

  ```
  Constraint SopAndEopOrPerr
      Type: Never
      Assume Never: (Phyn_Sop = 1) and
      (Phyn_Eop = 1 or Phyn_Perr = 1)
      Options: Default
  ```

- Constraint 12: The two signals `Phyn_Sop` and `Phyn_Wrvalid` rise always simultaneously and the two signals `Phyn_Eop` (respectively `Phyn_Perr`) and `Phyn_Wrvalid` fall always simultaneously:

  ```
  Constraint SyncSopAndWrvalid
      Type: Always
      Assume Always: (not((Phyn_Sop = rising) or
      (Phyn_Wrvalid = rising))) or
      ((Phyn_Sop = rising) and
      (Phyn_Wrvalid = rising))
      Options: Default

  Constraint SynEop_PerrAndWrvalid
      Type: Always
      Assume Always: (Phyn_Eop /= falling and Phyn_Perr /= falling and
      Phyn_Wrvalid /= falling) or
      (Phyn_Eop = falling and  Phyn_Wrvalid = falling) or
      (Phyn_Perr = falling and Phyn_Wrvalid=falling)
      Options: Default
  ```

  Here, the symbol "`/=`" means in FormalCheck "different from".

  Sometimes the formulation of the constraint is trivial, but in other cases, some theoretical background is required. The last two constraints are examples of such a case.

  Proof:

  We will give here a marginal proof of the last given constraint `SynEop_PerrAndWrvalid`. For this, we we define first the following proposition:

  $$Fall(X)\text{: The signal } X \text{ is falling.}$$

  In our case, and for simplification reason, we will denote `Phyn_Eop` by e, `Phyn_Perr` by p and `Phyn_Wrvalid` by w.

  Signals `Phyn_Eop` (or `Phyn_Perr`) and `Phyn_Wrvalid` fall simultaneously means that if `Phyn_Eop` or `Phyn_Perr` is falling then `Phyn_Wrvalid` is falling and reciprocally. This can be expressed as follows:

$$C = ((Fall(e) \vee Fall(p)) \Rightarrow Fall(w)) \wedge (Fall(w) \Rightarrow (Fall(e) \vee Fall(p))).$$

Since $A \Rightarrow B \equiv \neg A \vee B$ for any 2 propositions A and B , the constraint can be expressed as follows:

$$C = \neg(Fall(e) \vee Fall(p)) \vee Fall(w) \wedge \neg Fall(w) \vee (Fall(e) \vee Fall(p)).$$

By using the distribution rule : $(A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$ for any propositions A, B and C, we obtain:

$$C = (\neg(Fall(e) \vee Fall(p)) \wedge \neg Fall(w)) \vee (Fall(w) \wedge \neg Fall(w)) \vee (\neg(Fall(e) \vee Fall(p)) \wedge (Fall(e) \vee Fall(p))) \vee ((Fall(e) \vee Fall(p)) \wedge Fall(w)).$$

Since for a given proposition A, $\neg A \wedge A = False$ and $A \vee False = A$, last expression can be simplified as follows:

$$C = \neg(Fall(e) \vee Fall(p)) \wedge \neg Fall(w) \vee ((Fall(e) \vee Fall(p)) \wedge Fall(w)).$$

By applying one more time the distribution rule on the last expression, we obtain:

$$C = (\neg(Fall(e) \vee Fall(p)) \wedge \neg Fall(w)) \vee (Fall(e) \wedge Fall(w)) \vee (Fall(p) \wedge Fall(w)).$$

We know that $\neg(A \vee B) \equiv \neg A \wedge \neg B$. Thus, we deduce:

$$C = (\neg Fall(e) \wedge \neg Fall(p) \wedge \neg Fall(w)) \vee (Fall(e) \wedge Fall(w)) \vee (Fall(p) \wedge Fall(w)).$$

In terms of FormalCheck, this expression can be written as:

```
Assume Always: (not(Phyn_Eop =falling) and
not(Phyn_Perr = falling) and
not(Phyn_Wrvalid =falling)) or
(Phyn_Eop = falling and
Phyn_Wrvalid = falling) or
(Phyn_Perr = falling and
Phyn_Wrvalid = falling)
```

which is equivalent to the expression of `SynEop_PerrAndWrvalid` □.

5 Properties Specification

After establishing a proper environment, and define all needed constraints, we consider 10 queries of the Memory Manager, including liveness and safety properties. The following properties have been defined based on specification and test benches given in [6]. For all of them, we used *Symbolic (BDD)* as algorithm and *1-Step* as reduction technique [5].

- QUERY 1: First of all, we verified the global reset of the Memory Manager. During the reset process, launched by the signal `Sys_Rst_N`, the output signals of the Memory Manager should take their appropriate values. Thus, signals `Mgr_Sop`, `Mgr_Eop`, `Mgr_Perr`, `Mgr_busy`, and `Mgr_Delack` should be sampled to low, whereas signals `Mgr_enable_Ok` and `Mgr_wr_n` should be sampled to high. This query is composed of 7 properties. Each signal is verified by an associated property. Exceptionally for this query, only the two constraints `System_Clock` and `System_Reset` are included. Properties of this query are expressed as follows:

```
Query:  Reset
    PROPERTIES:
property: ResetMgr_Sop
   Type:  Always
   After: Sys_Rst_N = rising
   Always: Mgr_Sop = 0
   Options: Fulfill Delay: 0
            Duration: 1 counts of
            Sys_Clk = rising
.... property: ResetMgr_Enable_Ok
   Type:  Always
   After:  Sys_Rst_N = rising
   Always: Mgr_Enable_Ok = 1
   Options: Fulfill Delay: 0
            Duration: 1 counts of
            Sys_Clk = rising
....
```

  The options in these properties are used to define the time and the duration of verification. The *fulfill delay* expresses the delay added between the enabling condition (`Sys_Rst_N = rising`) and the checking of the fulfilling condition. In our case, the delay is zero which means that the fulfilling condition is checked immediately after the enabling condition is true. The duration of the verification is expressed by the *duration* option. In our case, the verification windows terminates after one occurrence of the event `Sys_Clk = rising`.

- QUERY 2: According to the Memory Manager documentation, when the Memory Manager begins the reception of a packet, it informs the Controller by sampling the signal `Mgr_Sop` to high during one cycle starting at the falling edge of the system's clock defined by the signal `Sys_Clk`. In this query, we verify the duration of the signal `Mgr_Sop` (first property) and its synchronization with the clock (second property). In FormalCheck, this query is expressed as follows:

```
Query: Verif_Mgr_Sop
    PROPERTIES:
property: OneClkMgr_Sop
    Type: Never
    Never: (Mgr_Sop = 1) and (WasMgr_Sop = 1)
    Options: (None)
property: SyncMgr_Sop
    Type: Always
    Always: (Sys_Clk = falling) or (Mgr_Sop = stable)
    Options: (None)
WasMgr_Sop: Range 0 to 1
    Initial: 0
    if (Mgr_Sop = 1) and (Sys_Clk = 1) then
      WasMgr_Sop := 1;
    else
      WasMgr_Sop := 0;
    end if;
```

where WasMgr_Sop is a state variable that indicates if the signal Mgr_Sop is already activated or not.

- Query 3: Inspired by the specification given in the last query, we verified that the Controller is informed when the Memory Manager is receiving a packet. In addition, from [6], Mgr_Sop is sampled to high one clock cycle after the activation of the signal Phyn_Sop. We defined this property as follows:

```
Query: Phyn_SopMgr_Sop
    PROPERTIES:
property: PhynSop_MgrSop
    Type: Eventually
    After: (Phyn_Sop = 1 and Mgr_Enable_Ok = 1)
    Eventually: Mgr_Sop = 1
    Options: Fulfill Delay: 0
    Duration: 1 counts of
    Signal: Sys_Clk = rising
```

- Query 4: At the end of transmission (Phyn_Eop is sampled to high during one cycle), the Memory Manager informs the Controller by sampling the signal Mgr_Eop to high during one cycle starting at the falling edge of the system's clock. In FormalCheck, this query is expressed as follows:

```
Query: Verif_Mgr_Eop
    PROPERTIES:
property: OneClkMgr_Eop
    Type: Never
    Never: (Mgr_Eop = 1 and WasMgr_Eop = 1)
    Options: (None)
property: SyncMgr_Eop
    Type: Always
    Always: (Sys_Clk = falling or Mgr_Eop = stable)
    Options: (None)
WasMgr_Eop: Range 0 to 1
    Initial: 0
    if(Mgr_Eop = 1 and Sys_Clk = 1) then
      WasMgr_Eop := 1;
    else
      WasMgr_Eop := 0;
    end if;
```

where `WasMgr_Eop` is a state variable that indicates if the signal `Mgr_Eop` is already activated or not.

- QUERY 5: From the specification given in the last query, we conclude that the Controller should be informed by the Memory Manager at the end of packet transmission. From the Memory Manager documentation [6], The controller is informed one clock cycle after the occurrence of `Phyn_Eop` event. We defined this property as follows:

```
Query: Phyn_EopMgr_Eop
    PROPERTIES:
property: PhynEop_MgrEop
    Type: Eventually
    After: Phyn_Eop = 1
    Eventually: Mgr_Eop = 1
    Options: Fulfill Delay: 0 Duration: 1 counts of
    Signal: Sys_Clk = rising
```

- QUERY 6: When an error is occurred during transmission (`Phyn_Perr` is sampled to high), the Memory Manager informs the Controller by sampling the signal `Mgr_Perr` to high during one cycle starting in the falling edge of the system's clock. In FormalCheck, this query is expressed as follows:

```
Query: Verif_Mgr_Perr
    PROPERTIES:
property: OneClkMgr_Perr
    Type: Never
    Never: (Mgr_Perr = 1 and WasMgr_Perr = 1)
    Options: (None)
property: SyncMgr_Perr
    Type: Always
    Always: (Sys_Clk = falling or Mgr_Perr = stable)
    Options: (None)
WasMgr_Perr: Range 0 to 1
    Initial: 0
    if(Mgr_Perr = 1 and Sys_Clk = 1) then
      WasMgr_Perr := 1;
    else
      WasMgr_Perr := 0;
    end if;
```

where `WasMgr_Perr` is a state variable that indicates if the signal `Mgr_Perr` is already activated or not.

- QUERY 7: Inspired by the specification given in the last query, we conclude that the Controller should be informed by the Memory Manager in case of error transmission. We know also from [6], that the controller is informed after a delay of one clock cycle from the occurrence of `Phyn_Perr` event. We defined this property as follows:

```
Query: Phyn_PerrMgr_Perr
    PROPERTIES:
property: PhynPerr_MgrPerr
    Type: Eventually
    After: Phyn_Perr = 1
    Eventually: Mgr_Perr = 1
    Options: (None)
```

- QUERY 8: Similarly to signals `Mgr_Sop`, `Mgr_Eop` and `Mgr_Perr`, the signal `Mgr_Delack` is activated during one cycle count from the falling edge of the system's clock. We expressed this query by the following two properties:

```
Query: Verif_Mgr_Delack
    PROPERTIES:
property: OneClkMgr_Delack
    Type: Never
    Never: (Mgr_Delack = 1) and (WasMgr_Dealck = 1)
    Options: (None)
property: SyncMgr_Delack
    Type: Always
    Always: (Sys_Clk = falling) or
    (Mgr_Delack = stable)
    Options: (None)
WasMgr_Perr: Range 0 to 1
    Initial: 0
    if(Mgr_Delack = 1) and
      (Sys_Clk = 1) then
      WasMgr_Delack := 1;
    else
      WasMgr_Delack := 0;
    end if;
```

  where `WasMgr_Delack` is a state variable that indicates if the signal `Mgr_Delack` is already activated or not.

- QUERY 9: Every suppression request should be answered positively. This property can be expressed by the following query:

```
Query: CtrDelvalidMgrDelack
    PROPERTIES:
property: CtrDelvalMgrDelack
    Type: Eventually
    After: Ctr_Delvalid = 1
    Eventually: Mgr_Delack = 1
    Options: (None)
```

- QUERY 10: The Memory Manager can be available for packet reception or packet suppression exclusively:

```
Query AfterDelandNoSopNoEnbl
    PROPERTIES:
property: AfterDelandNoSopNoEnbl
    Type: Eventually
    After: (Ctr_Delvalid = 1) and (Phyn_Sop = 0)
    Eventually Mgr_Enable_Ok = 0
```

This query means that if the Controller sends a suppression request and the Memory Manager is not receiving a packet from the Physical Interface, then the signal `Mgr_Enable_Ok` should be sampled to low to inform the Physical Interface that the Memory Manager is busy and cannot receive packets for the moment

- Query 11: Similar to the last query, this query insures that when the Memory Manager is busy with the suppression request, the signal `Mgr_Enable_Ok` remain sampled to low until the suppression is finished.

  ```
  Query NoEnableUntilDelack
       PROPERTIES:
  property: NoEnableUntilDelack
       Type: Always
       After: Ctr_Delvalid = 1 and Mgr_Busy = 0
       Always: Mgr_Enable_Ok = 0
       Unless: Mgr_Delack = 1
  ```

- Query 12: The following query is to insure that the signal `Mgr_Delack` cannot be sampled to high without a suppression request from the Controller:

  ```
  Query NoDelvalidDelack
       PROPERTIES:
  property: NoDelvalidDelack
       Type: Never
       Never: Ctr_Delvalid = 0 and Mgr_Delack = 1
  ```

- Query 13: The following query is introduced to insure that a suppression request should not be answered during a packet reception:

  ```
  Query NoDelWhenSop
       PROPERTIES:
  property: NoDelWhenSop
       Type: Always
       After: (Phyn_Sop = 1) and (Mgr_Enable_Ok = 1)
       Always: Mgr_Delack = 0
       Unless: (Phyn_Eop = 1) or (Phyn_Perr = 1)
  ```

- Query 14: When a packet is transmitted from the Physical Interface, the Memory Manager activates writing in the Main Memory by setting the signal `Mgr_Wrn` to low. This property can be expressed by the following query:

```
Query SopMgrWrn
     PROPERTIES:
property: SopMgrWrn
     Type: Eventually
     After: (Phyn_Sop = 1) and (Mgr_Enable_Ok = 1)
     Eventually: Mgr_Wrn = 0
```

- Query 15: When a packet transmission begins, the Memory Manager informs the Controller about the transmitted packet and stores it into the Main Memory. This query is introduced to know if each packet which the reception has been informed to the Controller is written in the Main Memory

```
Query MgrSopMgrWrn
     PROPERTIES:
property: MgrSopMgrWrn
     Type: Eventually
     After: Mgr_Sop = 1
     Eventually: Mgr_Wrn = 0
```

- Query 16: In this query we want to know if the Memory Manager informs the Controller about each written packet in the Main Memory:

```
Query MgrWrnMgrSop
     PROPERTIES:
property: MgrWrnMgrSop
     Type: Eventually
     After: Mgr_Wrn = 0
     Eventually: Mgr_Sop = 1
```

6 Experimental Results

All verifications in this project were executed on an *Ultra 5* Sun workstation with 256 MB RAM and UNIX operating system. For all properties, we used *Symbolic (BDD)* as algorithm and *1-Step* as reduction technique [5]. We found these modes effective enough to conduct the verification process. The experimental results are shown in Table 2, including the status of the property verification, the number of reached states (RS), the number of state variables in the model (SV), the average state coverage (SC), the search depth (SD), the CPU time (real time) in seconds and the memory usage in MB.

From Table 2, we can see that the system reset property and properties related to the duration of signals' activation are verified, but many others failed. Thus, when the Physical Interface starts a packet transmission, there is no guarantee that this packet will be detected by the Memory Manager and thereby will be stored in the Main Memory. In this case, the Controller will not be informed about this transmission and the packet will be lost. Intuitively, it seems logic that if the Memory Manager will not detect some packet transmissions, then it will not detect the end of these transmissions or the occurrence of some errors. In such cases, the Controller will not be informed by these activities performed by the Physical Interface since there is no direct dialogue between the Controller and the Physical Interface, and all information concerning the packet transmission are transmitted to the Controller by the Memory Manager.

In [6], there is an illustrated case in which packets are lost. Namely, when the Main Memory is full and no more space is available to store incoming packets; the Memory Manager will simply reject incoming packets without informing the Physical Interface. In this case, the Physical Interface will continue sending packets to the Memory Manager. However, in [6], this situation was defined as the only case in which packets are lost. In our work, FormalCheck gives a counterexample and shows that in spite of memory space availability, incoming packets can still get lost.

Another example is *Query 16*, which failure means that some packets are stored into the Main Memory by the Memory Manager, but the Controller does not know anything about them because the Memory Manager does not inform the Controller once they are transmitted by the Physical Interface. This means that some packets will remain in the Main Memory without conversion. Furthermore, the Controller will never ask the Memory Manager for the suppression of these packets.

Table 2: **Summary of Experimental Results**

QUERY	STATUS	RS	SV	SC (%)	SD	CPU (s)	Mem (MB)
Query 1	Verified	2.68^{+7}	64	87.50	773	70	14.37
Query 2	Verified	3.21^{+3}	15	96.77	38	10	2.32
Query 3	**Failed**	5.09^{+3}	59	88.98	12	12	10.62
Query 4	Verified	3.14^{+3}	31	96.77	38	9	2.32
Query 5	**Failed**	5.25^{+3}	30	100	42	9	9.54
Query 6	Verified	7.85^{+6}	51	97.06	803	9	2.32
Query 7	**Failed**	8.89^{+6}	52	98.08	802	40	10.85
Query 8	Verified	3.08^{+3}	31	96.77	39	8	2.32
Query 9	Verified	4.37^{+5}	31	100	796	18	9.56
Query 10	Verified	5.98^{+5}	33	96.97	798	15	9.66
Query 11	Verified	4.38^{+5}	34	98.53	798	15	9.68
Query 12	Verified	2.15^{+5}	32	100	844	18	9.66
Query 13	Verified	2.15^{+5}	33	100	844	17	9.67
Query 14	Verified	2.82^{+9}	46	98.41	844	12	2.34
Query 15	Verified	2.82^{+9}	46	98.39	844	13	2.34
Query 16	**Failed**	2.82^{+9}	64	98.44	844	192	16.63

7 Related Work

There exists a number of related work in the open literature on the use of model checking as a technique to verify models of commercial products. These case studies are too numerous to be listed here. In the following, we elaborate on a few of them. For instance, in [1], FormalCheck was used to verify the implementation of a SCI-PHY (Saturn Compatible Interface for ATM-PHY devices) Level 2 protocol engine, commercialized by PMC-Sierra, Inc. Some properties covering the essential behavior of the SCI-PHY protocol are defined and then checked on an abstracted model (8-PHY devices) as well as the original hardware model (32-PHY devices). The set of established properties contains 11 properties. However, only one error has been detected. In comparison, the design considered in our project is more complex and presents a generic behavior. In [10], VIS (Verification Interacting with Synthesis) [3], a model checking tool, was adopted for the verification of an Asynchronous Transfer Mode (ATM) switch used for real applications in the Cambridge Fairisle network. Abstracted and reduced models of the switch at different levels of the design hierarchy are established, then a set of typical properties of the switch and its components are verified. The benefit of this work is that it shows how the VIS tool can partially verify large size circuit design by using reduction, abstraction and property division. In [2], the same ATM switch fabric design is verified in FormalCheck and a comparison between the two hardware verification tools is given. From the experimental results in [2], it was shown that FormalCheck is faster than VIS and that the memory usage of FormalCheck is less than that in VIS for all verified properties. Moreover, no manual reduction or property composition were required in FormalCheck. This justify again our choice of FormalCheck to verify our system. In [12], VIS was used for the verification of an ATM ring (ATMR) media access control (MAC) protocol. Since VIS is a model checking tool targeting synchronous hardware system, this report shows how to simulate the asynchronous ATMR MAC in the synchronous VIS environment. However, this in turn has created a state space explosion. In FormalCheck such environment can be modeled using a set of constraints as illustrated in our project. VIS was also used in [11] to formally verify a commercial product of PMC-Sierra, Inc. that processes Routing, Cell counting, Monitoring, Policing (RCMP) for the network port interface of an ATM switch fabric. In this work, a design error which could lead the system to a deadlock state was detected. However, properties that involve the introduction of time delays were not verified because unlike FormalCheck, as used in our project, VIS does not support timed Verilog models. In [16], MDG (Multiway Decision Graph) [8] was used in the formal verification of a Telecom System Block commercialized by PMC-Sierra, Inc. which processes a portion of the SONET (Synchronous Optical Network) line overhead of a received date stream. While the MDG tool possesses efficient features for data abstraction, it does not support neither Verilog nor VHDL to be considered in a complex project like ours.

8 Conclusion

In this study, we explored the formal verification by model checking of the Memory Manager Component of the Protocol Converter system of a System-on-a-Chip platform. The Memory Manager is the main block of the system and consists of five modules, namely, a Memory Manager Controller, an Address Counter Register, a Data Counter Register, a Packet Counter Register and a Packet Assembler. After establishing a proper environment for the Memory Manager, we specified some relevant properties expressible in FormalCheck and accomplished their verification under the defined set of constraints. Our experimental results demonstrated the presence of many residual bugs in the design. The impact of these errors is very important since they cause, in some situations, the non-conversion of received packets and the non-liberation of stored packets from the Main Memory. Though the important effects of such situations on the functionalities of the Protocol Converter system, these errors were not detected by extensive post-design simulation efforts. Since all detected bugs represent serious problems, we informed the designers about them. By using test benches, designers confirmed the presence of these bugs and proceeded to modify and debug their design.

The main contribution of this work is the emphasis of the importance of formal methods for design verification and validation. In our case, we were able to detect many errors that were not detected by simulation. However, we should mention that formal techniques are not by themselves an alternative to verify if a design is correct with respect to a specification, but should be used with simulation as a complementary process to insure a maximum error detection.

When we deal with formal verification by model checking, we should pay a great attention to the set of defined constraints. Constraints are generally used to simulate the environment on which the system operates. They are the most delicate task since the set of constraints should be as complete as possible to describe the exact behavior of the environment.

Our extensive experience shows that the developed set of constraints actually defines a set of verifiable properties for the other components of the Protocol Converter system: the Controller and the Main Memory, whereas our specified properties for the Memory Manager, can be used as a the respective set of constraints for these two components. The principle benefits of permuting the roles of properties and constraints are that we do not have to redo the formalization of the system specification for each verified component and also we can be insured that all components of the global system are verified under the same set of assumptions and the same protocol of communication between each couple of interacting components.

In summary, our work joins other successful industrial-sized case studies in using model checking for hardware verification. We believe to have contributed fostering the evidence that model checking is now powerful enough to be widely used in industry to help in the verification of developed complex hardware designs.

References

[1] L. Barakatain and S. Tahar. Functional Verification of a SCI-PHY Level 2 Protocol Engine. In *IEEE International Conference on Information, Communications and Signal Processing (ICICS'01)*, October 2001.

[2] L. Barakatain and S. Tahar. Model Checking of the Fairisle ATM Switch Fabric using FormalCheck. In *Proc. IEEE Canadian Conference on Electrical and Computer Engineering*, Toronto, Canada, May 2001.

[3] R. K. Brayton and al. VIS: A System for Verification and Synthesis. In T. Henzinger and R. Alur, editors, *Eigth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] Cadence. *Formal Verification Using Affirma FormalCheck, version 2.4*, Auguest 1999.

[6] S. Carniguian, J. Chevalier, M.M Mbaye, S. Regimbal, and J-L. Trépanier. Intégration et vérification d'un convertisseur de protocoles. Rapport final, École Polytechnique de Montréal, Département de génie életrique, 2002.

[7] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proc. Logics of Programs Workshop*, volume 131 of Lecture Notes in Computer Science, pages 52–71, New York, 1981. Springer-Verlag.

[8] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in Hardware Design*, 10:7–46, February 1997.

[9] C. Kern and M. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 4:123–193, April 1999.

[10] J. Lu, S. Tahar, D. Voicu, and X. Song. Model Checking of a Real ATM Switch. In *IEEE International Conference on Computer Design (ICCD'98)*, pages 195–198, Austin, Texas, USA, October 1998.

[11] P. Murugesh and S. Tahar. Formal verification of the RCMP Egress routing logic. In *Proc. IEEE 11th International Conference on Microelectronics*, pages 89–92, Kuwait City, Kuwait, November 1999.

[12] H. Peng and S. Tahar. Hardware modeling and verifiation of an ATM Ring MAC Protocol. In *Proc. IEEE 12th International Conference on Microelectronics*, pages 21–24, Teheran, Iran, November 2000.

[13] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, January 1981.

[14] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CAESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium in Programming*, volume 137 of Lecture Notes in Computer Science, pages 337–351, New York, 1982. Springer-Verlag.

[15] A. Rushton. *VHDL for Logic Synthesis.* John Wiley & Sons Ltd, second edition, 1998.

[16] M. H. Zobair and S. Tahar. Formal Verification of a SONET Telecom System Block. In *International Conference on formal Engineering Methods (ICFEM'02)*, Shangai, China, October 2002. Lecture Notes in Computer Science, Springer Verlag.