

Formal Verification of a Framer OC-768 (40 Gbits/s)

DEVAUCELLE Christophe

Trainee at IBM Microelectronics Essones Labs

Corbeil-Essones, France

devaucel@enst.fr

Abstract: The main method for hardware verification in microelectronics consists of running an arbitrary number of test vectors, which is also known as simulation. However, as the complexity of the design increases, the coverage for this method becomes limited. For many years, formal verification was the answer, but its use was very limited. The size of design was too small in an industrial setting, there were limitations for writing the VHDL for some tools, and the language used to describe the model to check was too complex. In this paper, I evaluate RuleBase, a formal verification tool with a simple, but powerful language that confronts the above limitations, and its use in checking the Framer OC-768.

Keywords: RuleBase, Model Checker, Formal Verification, Hardware Verification, Framer OC-768

1. Introduction

Today, the main method for hardware verification in microelectronics consists of running an arbitrary number of test vectors. This method is known as simulation. In simulation, a test vector is applied to the logic model, and the results of the simulation are examined. However, as the complexity of the design increases, the coverage for this method becomes limited.

Formal verification is a novel technique for logic verification of hardware designs. It attempts to address the problem of coverage by mathematically proving that a design is correct with respect to its specification. Formal verification is a method that provides exhaustive coverage, and is therefore, of interest to many development and research labs.

In this document, I evaluate RuleBase, a formal verification tool developed at the IBM Haifa Research Labs on the Framer OC-768 from the IBM Research Lab at Essones. In Section 2, I provide an overview of RuleBase, the formal verification tool. In Section 3, I present the verification results, and in Section 4, I present my conclusions as to whether the RuleBase tool should be integrated into the verification methods at the IBM Essones Lab.

2. RuleBase

RuleBase is a formal verification tool created by the IBM Haifa Research Labs. More precisely, RuleBase is a symbolic model checker, which means that it compares a synthesizable design, written in VHDL or Verilog, to a model defined by a set of rules (composed by one or more temporal and logical formulas).

The core of RuleBase is an enhanced version of SMV, a model checker created at Carnegie-Mellon University (Pittsburgh, Pennsylvania). The main characteristic of SMV is to represent the design by a state machine and to compress the space state using Ordered Binary Decision Diagram (OBDD).

RuleBase proposes different algorithms for BDD order including optimization and reduction algorithms, which suppress logic that is unnecessary for the rule being checked to raise the size of checkable design, and some new functions such as On-The-Fly Model Checking, which allows RuleBase to find a counter-example without building the whole space state.

RuleBase also enhances the language used to write the rules. In SMV, the language is called CTL (Computational Tree Logic). CTL is very efficient for the model-checking algorithm, but formulas can quickly become complicated. SUGAR is an evolution of CTL and is used in RuleBase for creating formulas. Sugar is far more intuitive and tries to keep the efficiency of CTL.

To check a design, RuleBase needs not only a synthesizable design and a set of rules, but also a description of the behavior of the input, which is called the environment. To simplify the design, designers often assume a certain behavior for the input. If the designers do not provide this behavior to RuleBase, it will produce many false counter-examples, which result from an illegal input sequence.

3. Results

The following sections present my results after performing formal verification on the Framer OC-768 using RuleBase.

3.1 Block Interface with Network Processor on Transmit Side

- Presentation:** This block receives data from a Network Processor and sends it to a FIFO. The Network Processor sends words of 16 bits at 400 MHz double-edged with another bit of flag, which indicates if the word is a data or control word.

Control words include: Idle, Payload (just before data), Training (a word used in a training sequence for synchronization), and Reserved (a word used to extend the address of a payload, but is not supported). The four last bits of a control word correspond to the 4 bits of parity (code DIP-4 for Diagonal Interleaved Parity), which protect the previous transfer of data.

This block analyzes words that come from the Network Processor. It checks the DIP-4, the protocol, the address in the Payload control words, and then writes the data to the FIFO with the correct format (2 bits of flags by bytes of data. The `Start of Packet` is always on the first byte). Moreover, this block sends a FIFO status to the Network Processor.
- Environment:** The environment is very simple. The signals of configuration are fixed and the input words are left undetermined. To make the rule easier to write, there is a function that generates all the control words with the correct format. Choices for the DIP-4 include: fixed, always good, or always bad. Some other function checks the DIP-2 that is sent after a sequence of FIFO status. Some modes have also been created to simplify the environment.
- Rules:** All the outputs have been checked. Moreover, some internal signals have also been checked, including a flag that decodes incoming data, the state machine that analyzes the protocol, and the internal block that verifies the DIP-4. The verification of the DIP-4 checked all the possible sequences (a sequence can be 16 * 16 bytes, so it has 2^{2048} possibilities) that would have been impossible to check exhaustively with simulation.

The following is an example that shows how easy it is to check that when the DIP-4 is correct, no error is signaled:

```
formula "Check if the calculus of the dip4 is good" {
  AG ( spit_dip4error(0..1)=0 )
}
```
- Results:**

- **Implementation of a new function:**

When the block receives two `Start of Packets` without an `End of Packet` in between, then the first packet must be aborted and the protocol error counter must be incremented. RuleBase was very useful for verifying this function, since, at first, there were some special cases (e.g., packets of less than four bytes) where the function did not work. This method is also very fast because RuleBase needs less than two minutes to check this function.

- **First bug:**

Due to the format of the output flag, the block does not normally support packets of one byte. The problem is that if the block receives such a packet, it will write a packet of 4 bytes without an `End of Packet` on the FIFO, which can corrupt the following packet.

- **Second bug:**

When reserved words are used, there are some cases where an `End of Packet` is written to the FIFO even if a `Start of Packet` was not sent before. This bug would not have been easy to find with simulation because we can only observe the bug for certain sequences.

- **Third bug:**

On reset, the principal state machine, which checks the protocol, is in the state of `Synchro`. It waits until it receives enough consecutive good `DIP-4` to be synchronized. For synchronization, the Network Processor sends a Training sequence (10 Training control words followed by 10 Training data words) that is repeated several times. If the block is synchronized before the end of the training sequence, the block increments the protocol error counter. This bug would have been difficult to find with simulation because it deals with the configuration of the SPIT and the Network Processor.

- **Fourth bug:**

Payload control words contain an address that must be the same as the address of the SPIT. If not, the block must signal an error and the bad address must be written to a register. Two problems exist: first, the block did not signal an error when there was no `Start of Packet` in the Payload, and second, if an error was signaled, the block wrote the good address in the register instead of the bad.

Before any optimization, this block had 413 flip-flops, 2966 gates, and 79 inputs.

(Rules in bold allowed me to find bugs)

Rule Name	FF	Environment Variable	Gates	Iterations	BDD nodes	States	User Time	Memory (Mo)
<i>StateReachable</i>	96	12	1629	45	3718025	$2 \cdot 10^{16}$	1 h	98
<i>DecodeCtrlWrd</i>	32	6	489	0	14797	288	30 sec	26
<i>StateDataBurst</i>	90	11	1489	0	87509	512	40 sec	29
<i>Valid</i>	97	12	1635	0	200168	9216	20 sec	31
<i>Size</i>	93	11	1550	0	200044	$5 \cdot 10^6$	3 min	31
<i>FSel</i>	91	11	1493	0	53429	512	20 sec	28

<i>MSel</i>	92	12	1637	0	177936	1152	20 sec	30
<i>Sop</i>	92	11	1516	0	17663	2048	1 min	30
<i>Eop</i>	93	11	1524	0	87464	$2 \cdot 10^6$	1 min 40	29
<i>Abort</i>	92	11	1587	0	715840	$8 \cdot 10^6$	2 min 40	41
<i>ProtErr</i>	105	11	1847	0	452397	$2 \cdot 10^7$	4 min	37
<i>SpitData</i>	156	11	1761	0	201178	$5 \cdot 10^6$	1 min 30	31
<i>Spit_Pkt_Inc</i>	33	11	605	0	17425	8192	10 sec	29
<i>Add_Err</i>	40	4	489	0	38345	576	20 sec	27
<i>Spit_Flag</i>	119	11	1809	0	708431	$7 \cdot 10^7$	5 min	41
<i>Spit_Flag2</i>	138	11	2047	0	708416	512	30 sec	41
<i>Spit_Flag3</i>	130	12	2073	0	675329	1152	1 min 10	40
<i>Spit_Write</i>	108	13	1703	0	700080	$1 \cdot 10^5$	3 min	41
<i>TwoSop</i>	105	11	1749	0	376300	$1 \cdot 10^5$	1 min 40	35
<i>Synchro</i>	99	11	1736	0	734604	$1 \cdot 10^6$	30 sec	41
<i>Dip4</i>	70	54	2018	0	710817	$1 \cdot 10^{12}$	40 sec	41
<i>BadDip4</i>	70	54	2090	0	1035978	$9 \cdot 10^{12}$	3 min	47
<i>FifoStatus</i>	104	16	1333	45	752567	$1 \cdot 10^5$	30 sec	41

Using RuleBase, most of the rules are very quick to check (under two minutes). This is due to the algorithm *Light Proof* that can check a formula without building the space state. *StateReachable* is the only rule that takes longer to check when compared to the others, which is because it is a Liveness formula (a formula with the operator EF).

3.2 Control Block of the Load Balancer

- **Presentation:** The Load Balancer distributes individual traffic flows at 40 Gb/s onto the 4 SPI-4 operating at 10 Gb/s. There are three main blocks in the Load Balancer: a parser, which extracts pertinent information from the header of a packet, such as the IP address; a Hash Function, which compresses the key coming from the parser; and a Look-Up Table, which is used by the algorithm that balances the data flow between the four SPI.

The only block of the Load Balancer that RuleBase has verified is the control block of the Look-Up Table. More precisely, RuleBase has only checked one internal function of this block — the pipeline forwarding. Writing in the Look-Up Table takes four cycles, and so, if we have to read at the same address during these four cycles, we must read data that is still in the pipeline. This is something

difficult to test with simulation because we have to imagine all the possible input sequences. With a model checker it is very simple; we just have to say that if write and read are at a distance of one cycle from each other than the data must be read at the first pipeline,

- **Environment:** The environment is very simple since most inputs are fixed. There is only one signal valid and one signal for writing, both of which are undetermined. Moreover, the address of the data (the key given by the Hash Function) can take four distinct values. Four because, in the worst case, there is a write/read at a different address at each cycle, and after four cycles the data is always read from the Look-Up Table.
- **Rule:** There are four distinct cases given by the distance (in cycle) between a write and a read at the same address. For each of these cases, I checked that the mux selects the good pipeline.
The following is an example of the formula I used:

```
formula "Two read on the same address with at least four cycle between" {
AG ( {valid=0,valid=1,address(0..13)=0 & valid=1, {address(0..13)!=0}[3..],
valid=1 & address(0..13)!=0, address(0..13)=0} (AX(muxin_ctrl(0.. 1)=0)))
} -- note: valid is always set one cycle before address
```
- **Results:** No bug has been found with the very last version of the block. If I keep the same environment and rule for checking the previous version of the block, RuleBase immediately finds a bug. This bug has been very difficult to find with simulation since it was only found after several months of simulation at the top level (simulation of the whole framer). For comparison, it takes me a little over one day to write the environment and the rule and find the bug with RuleBase.

Before any optimization, the control block had 299 flip-flops, 2369 gates, and 205 inputs.

Rule Name	FF	Environment Variable	Gates	Iterations	BDD nodes	States	User Time	Memory (Mo)
<i>PacketForwarding</i>	14	3	124	0	8911	8192	40 sec	28

We can see the efficiency of the algorithms of reduction. On the 299 flip-flops and 2369 gates from the original design, RuleBase only keeps 14 flip-flops and 124 gates. Moreover, RuleBase needs less than one minute to exhaustively check a function that would have been very difficult and long to test with simulations.

3.3 Block Between the Interface Block with the Network Processor Receive Side and the FIFO

- **Presentation:** This block changes the format of the FIFO (256 bits of data and 64 bits of flags) to the format of the SPIT (32 bits of data and 8 bits of flags). Moreover, the block must also rearrange data so that a `Start of Packet` must always occur on the first bit of the data.
- **Environment:** The environment is quite complex because it must manage 64 bits of flags with some rules to which it must conform, such as no more than two `End of Packets` in a row or the `End of Packet` must be separated by a `Start of Packet`. Moreover, I need to know the position of the `End of Packet` or the `Start of Packet` for the rules. All of this makes the run long (more than five or six hours for the first run) and so the debug phase for the environment and for the rules could have been very long.
I created another environment where the 56 last flags are fixed and only the 8 first flags are undetermined. With this environment, a run only takes a few minutes and I can debug the environment and rules quickly.
- **Rules:** The following main functions were checked:

- **FIFO Empty:**

When the FIFO is empty, the signal `spbr_valid`, which indicates that there is valid data to read, must be set at 0 until there is new data in the FIFO. The following is an example of such a formula:

```
formula "End of Packet in the first 4 bytes of data" {
AG ({typeFlag=sopEnd & clk200=1 & fifo_empty=0, true,
    clk200=1 & typeFlag=eopEnd & pozEop<4 & pozSop!=0 & fifo_empty=0,
    (fifo_read=0)[*],fifo_read=1 & spir_read=1,{fifo_empty=1 & spir_read=1}[2]}
    (AX[2](spbr_valid=0 until fifo_empty=0) ) )
} -- read during two cycles so that the 4 bytes of data have been send to the SPIR
```

- **Position of Start of Packet:**

The Start of Packet flag must always occur at the first bit of the output flag.

- **Position of End of Packet:**

The distance between the End of Packet and the Start of Packet must be the same in the input and in the output, so no data are lost.

These two last functions have been partitioned in many formulas and rules so that the formula is simple (Safety) and the space state is not too big.

- **Results:** As for the previous block, no bugs were found in the last version. RuleBase did, however, find a bug in the version of the block that occurred before the previous version. This bug was very difficult to find with simulation and it was hard to figure out in which block the bug existed since the bug was found with simulation at the top level. With RuleBase, we know exactly where and what the problem is since the rule `Fifo_Empty` failed. In some cases, the signal `valid` could be set to 1 where the FIFO was empty.

Before any optimization, this block had 402 flip-flops, 1914 gates, and 326 inputs.

(Rules in bold allowed me to find bugs)

Rule Name	FF	Environment Variable	Gates	Iterations	BDD nodes	States	User Time	Memory (Mo)
<i>Sop1</i>	83	4	32916	45	3212393	$2 \cdot 10^{12}$	2 h	127
<i>Sop2</i>	83	4	32916	45	3221768	$2 \cdot 10^{12}$	2 h	127
<i>Fifo_Empty</i>	80	5	32750	50	3424176	$3 \cdot 10^{13}$	3 h	131
<i>Eop_1</i>	89	4	32976	46	3500919	$6 \cdot 10^{12}$	2 h 45	133
<i>Eop_2</i>	89	4	32976	46	4264008	$1 \cdot 10^{13}$	3 h 45	147
<i>Eop_3</i>	89	4	32976	46	4009787	$2 \cdot 10^{13}$	2 h	142
<i>Eop_4</i>	89	4	32976	46	3254148	$3 \cdot 10^{11}$	2 h	142
<i>Eop2_1</i>	89	4	32976	40	3718217	$3 \cdot 10^{11}$	2 h 15	142
<i>Eop2_2</i>	89	4	32976	40	3172644	$2 \cdot 10^{11}$	2 h 20	142
<i>Eop3</i>	82	4	32897	0 (tautology)	710007	2	40 min	74

All these numbers correspond to the complete environment (where all the flags are undetermined) with the research of a witness. We can see that there are many more gates after reduction, which is due to the very complex environment. With the other environment (only 8 flags are undetermined), and there are only 1706 gates after reduction.

3.4 Block Interface with Network Processor on Receive Side

- **Presentation:** This block changes 32 bits of data and 8 bits of flag at 400 MHz in 16 bits of data and 1 bit of flag (indicates if the word is data or control) at 400 MHz double-edge. This block must generate a

control word (Payload before a data transfer, Training for the synchronization, Idle when there is no data to send).

- **Environment:** The environment manages four main points:
 - **Clocks:**
There are three clocks: 800 MHz for the output (400 MHz double edge), 400 MHz for the input data, and 200 MHz for the FIFO status.
 - **FIFO status:**
The FIFO status must follow a precise protocol otherwise this block will not send any data. This protocol is respected in the main environment, but there is also a mode where the protocol can be wrong to check the block in case of error in the protocol.
 - **Input flags:**
Input flags must respect certain rules, such as a `Start of Packet` can only occur on the first bit, and there are never two `Start of Packets` without an `End of Packet` in between. Some modes, which limit the possibility on the input flags, have been created to deal with the size problem.
 - **Signal *valid*:**
This signal is used in hand shaking between this block and the previous one. When this block wants to read data from the FIFO, it sets the signal *read* to 1. Then, if there is data in the FIFO, one cycle later, *valid* is set to 1. Moreover, this signal can only fall back to 0 if there is an `End of Packet` or after eight cycles have run. Finally, when a packet of less than 4 bytes is sent, the *valid* signal must return to 0 during two cycles. If it is a packet of less than 8 bytes, it must return to 0 during one cycle.

Even if the behavior of certain signals seems a bit complex, it is very easy to describe in EDL.

- **Rules:** I checked that the output protocol is followed, the DIP-4 is correctly generated, and that all the data is sent. The main problem is the size of the design, so I tried to raise the efficiency of the reductions by simplifying the environment in many rules.
- **Results:**
 - **First bug:**
When the block receives a packet of less than 8 bytes followed by a packet of 2 bytes while the block is sending a training sequence, then between these two packets, the block sends a Payload control word followed by an IDLE, which is forbidden by the protocol. Clearly, this bug is quite impossible to find with simulation, since we must not only generate the particular sequence packet of less than 8 bytes followed by a packet of 2 bytes, but we also must synchronize this sequence with the sending of the Training sequence.
 - **Second bug:**
RuleBase found a case where the block sends data without sending a Payload control word before the data. Contrary to the previous bug, I have not been able to reproduce this bug in simulation, even if such a bug was observed during simulation at the top level. I, therefore, cannot affirm that this bug comes from the VHDL and not from my environment. When the first bug is corrected, this bug disappears.

Before any optimization, this block had 911 flip-flops, 6255 gates, and 116 inputs.

(Rules in bold allowed me to find bugs)

Rule Name	FF	Environment Variable	Gates	Iterations	BDD nodes	States	User Time	Memory (Mo)
<i>FlagSop1</i>	233	19	2812	0	1502687	7680	3 min 30	58
<i>FlagSop2</i>	207	19	2676	617	46918979	$3 \cdot 10^{20}$	9 jours	928
<i>FlagSop3</i>	250	19	2625	373	-	$1 \cdot 10^{11}$	-	-
<i>FlagEopAbort</i>	234	14	2271	651	2871592	$2 \cdot 10^{16}$	1 h	88
<i>Padding</i>	343	19	3183	-	-	-	-	-
<i>Credit</i>	-	-	-	-	-	-	-	-
<i>Training</i>	-	-	-	-	-	-	-	-
<i>FifoStatusError</i>	20	7	236	646	709123	$2 \cdot 10^9$	7 min	45
<i>Pause</i>	230	19	2496	374	91739338	$5 \cdot 10^{11}$	12 jours	1783
<i>PayloadData</i>	-	-	-	-	-	-	-	-
<i>NoReserved</i>	199	19	2636	0	747896	$2 \cdot 10^9$	1 min	43
<i>Sop</i>	-	-	-	-	-	-	-	-
<i>CtrlWord</i>	206	19	2419	348	42170507	$3 \cdot 10^{11}$	4 jours	836

As we can see, calculation times vary from one minute to several days. We also notice that RuleBase can use a lot of memory (up to 1.6 Gb). This block shows the size limit of a design that RuleBase can handle. However, it is important to know that all of these numbers correspond to the case where there is no error. RuleBase must create the whole state space; for example, the calculation time for PayloadData takes only one or two hours to find the two bugs.

4. Conclusion

For many years, the use of formal verification was very limited. The size of design was too small in an industrial setting, there were limitations for writing the VHDL for some tools, and the language used to describe the model to check was too complex.

RuleBase makes formal verification easy with the very simple, but powerful language SUGAR. Even with a limitation on the size of the design, RuleBase can check the industrial design with its multiple optimization and reduction algorithms.

Even though RuleBase was introduced very late in the validation stage of the framer, it found some bugs that would have been impossible to find using simulation. We will continue to use RuleBase to check other blocks of the framer DEGAS and for future projects.

5. Acknowledgements

I would like to thank Cindy Eisner, Tamir Heyman, and Emmanuel Zarpas from the IBM Haifa, whose cooperation contributed to this work.

I would also like to thank Yaron Wolfstahl from IBM Haifa, Veronique Guerre from IBM Essones, and Renaud Pacalet from Telecom Paris for giving me the opportunity to work in the exciting area of formal verification.