Automatic Formal Verification of Digital Components of IoTs Using CBMC

Qurat-ul-Ain, Osman Hasan and Kashif Saghar
School of Electrical Engineering and Computer Science (SEECS)
National University of Sciences and Technology (NUST)
Islamabad, Pakistan
Email: {qain.msee15seecs,osman.hasan,kashif.saghar}@seecs.nust.edu.pk

***Abstract*—These days, internet of things (IoT) are being widely used in many safety-critical domains, like healthcare and transportation. Thus, their functional correctness is very important. However, simulation based analysis is based on sampling methods and thus their results are not complete and cannot be termed as accurate. Formal verification has been recently proposed to verify the digital components of IoT devices and thus overcome the incompleteness issues of simulation. However, formal verification process requires manual development of a formal model of the given circuit and its desired properties. Moreover, the verification of the relationship between the formally specified model and its properties sometimes also requires manual interventions. These manual efforts can be quite cumbersome while verifying large systems and thus make formal verification of IoT devices somewhat infeasible for industrial usage. To overcome these limitations, we present a tool chain to automatically formally verify digital components of IoT devices, which are usually expressed in the Verilog language. The proposed methodology primarily leverages upon the strong verification support for the C language. The idea is to convert the given Verilog code and its properties to C language and use bounded model checking to verify the obtained C code. The formally verified C code is then converted back to Verilog to facilitate circuit design steps i.e., synthesis, timing analysis etc., and thus continue with the regular digital system design flow. For illustration, we present the verification of several widely used components of IoT devices, including an ALU and a 64-bit processor, which are fairly complex and to the best of our knowledge have never been formally verified automatically before.**

Keywords*—*Formal Verification, Verilog, Digital Circuits, Internet of Things, Bounded Model-Checking

I. Introduction

The emergence of the internet of things (IoT) is enabling many physical systems of the world for data exchange and communication over the internet. This combination of cyber and physical systems are extensively being used in many safety-critical domains, like transportation and healthcare. Verification of digital designs, which is an essential component of every IoT device, is of utmost importance due to the heavy cost of undetected bugs in the hardware. Traditionally, digital designs are verified using simulation, which ascertains the correctness of the design by observing the behaviour of the circuit under a subset of all possible inputs due to inability to perform exhaustive simulation for large circuits. This non-exhaustive nature of simulation does not allow us to guarantee an accurate analysis and thus the circuit under analysis may contain bugs.

Formal verification [1] is an accurate alternative to simulation that overcomes its limitations by proving or disproving the correctness of the given design against its desired properties mathematically. The main principle behind formal analysis of a digital circuit is to construct a computer based mathematical model of the given circuit and formally verify within a computer, that this model meets rigorous specifications of intended behaviour. Thus, the engineer working with a formal methods based verification tool has to develop a formal model of the given circuit and the formal specifications of the desired properties. Moreover, she has to be involved in the verification task as well.

Due to importance of formal verification in hardware design, various formal verification tools have been proposed using different modeling and verification techniques. For example, recently higher-order-logic theorem proving [1] has been quite frequently used for this purpose. For example, PROVERIFIC [2] allows hardware design verification through Property Specific Language (PSL) assertions. A gate level description of the electronic control unit has been verified using the HOL theorem prover [3]. Braibant [4] proposed to formally verify hardware circuits using the Coq theorem prover. In all these works, the user has to develop a formal model of the given circuit manually and verify the model with the help of desired properties. Manual verification makes the above-mentioned techniques quite infeasible for hardware engineers in the industry.

Automatic verification of hardware circuits has been recently proposed in [5] using HOL theorem prover. However, the proposed idea is limited to verification of combinational circuits only. It has been observed that most of the formal verification tools [6]–[8] perform synthesis at the bit-level. Moreover, the existing formal verification techniques mostly convert the given Verilog design to a net-list using And-inverted graphs, which restricts the verification of Verilog designs specified at the register transfer level (RTL) [9]. Reveal [10] is a formal verification tool for verifying complex control logic of digital designs with wide datapaths. Reveal employs counterexample-guided abstraction refinement (CEGAR) but is not available as an open source development, which limits its extensive usage. It can be observed that many formal verification tools, like VIS[7], VCEGAR [11], and UCLID[12], cannot be used for verifying larger designs as the tools time out since their requirements often exceed the available memory for large designs. The authors in [13] perform formal hardware verification through bounded model checking to overcome this

problem. However, the final output obtained in this work is a formally verified C code, which executes sequentially unlike Verilog/VHDL, which performs parallel operations.
In order to overcome the above-mentioned limitations, we propose to first transform the Verilog code of the circuit to be verified to C using a Verilog to C translator. This step allows representing the given Verilog code at the word level, which can be considered as a software net-list. We propose to formally verify this C file with respect to the given properties using the bounded model-checking technique. This verified C file is proposed to be converted to Verilog at the RTL level after verification. Thus, the proposed methodology allows us to obtain formally verified Verilog files at the RTL level. Moreover, it allows us to leverage upon the efficient verification methods developed for C code verification for verifying hardware designs and thus makes the verification more scalable. The conversion from C to Verilog also integrates the high-level synthesis process within the verification framework and is thus another useful outcome of the proposed methodology. The two main contributions of this paper are:

1. We present a tool chain using open source tools to formally verify hardware design. For this, we mainly use three tools; V2C (Verilog to C converter) [9] , CBMC (bounded model-checker for C) [14] and the BAMBU (C to Verilog converter) [15] tool.
2. To illustrate the practical effectiveness of the proposed methodology, we utilize it to formally verify some digital circuits, including Multiplexer, Encoder, Adder, Arithmetic logic unit (ALU) and a 64-bit processor.

The rest of the paper is organized as follows. Section 2 describes the proposed methodology in detail. The illustrative case studies are explained in Section 3. Finally, Section 4 concludes the paper.

II. Methodology

The proposed methodology, depicted in Figure 1, requires two inputs, i.e., (i) the Verilog code of the digital circuit and (ii) the desired properties of the given circuit that need to be verified. The final output of our methodology is the formally verified and synthesized Verilog code. As depicted in Figure 1, the first step in the proposed methodology is to convert the Verilog code along with its desired properties, specified as assertions, to C language using the V2C tool. The next step is to parse and then verify the obtained C code using the CBMC model checker. If the C code verification is successful, then we convert the formally verified C code to its corresponding Verilog code using the high level synthesis tool BAMBU. Otherwise, the code is checked to find if the failures are related to modeling or translation errors or are actual functional bugs. The input Verilog code is revised accordingly and the process is repeated until we successfully verify all the given properties. The main steps involved in the proposed methodology are explained in detail below:

A. Verilog to C Conversion

Various tools are available for translating Verilog files to C files. Verilator [16] is an open source tool, which allows converting Verilog codes to C or C++. It can also be used for simulating the Verilog code or converting C files to Verilog code. However, we noticed that the output generated via Verilator is not the most optimal C implementation of the given functionality. It suffices for simulation purposes but does not suit our context where the verification complexity grows exponentially with the size of the model.
Verilog2C++ [17] is another tool that can be used to convert simple Verilog files to C++. VTOC [18] is another tool for converting Verilog to C file, which is quite efficient but we passed on it due to its non-availability as a free-ware tool.
V2C (Verilog to ANSI-C) [9] also allows translating Verilog to C and is available as open source. The generated C file is quite optimal and is nearly equivalent to the size of Verilog file and easily understandable. After exploring various Verilog to C converters, we decided to use the V2C tool for our work. The generated C program is cycle-accurate and can be used as the word-level representation for the considered hardware circuits. The translation of C to world-level gives an opportunity to use Satisfiability Modulo Theories (SMT) based tools for the verification purpose. However, one of the main limitations of the V2C tool is that the generated C file may contain some syntax errors. We have developed an automatic tool to rectify these errors before saving the C file.

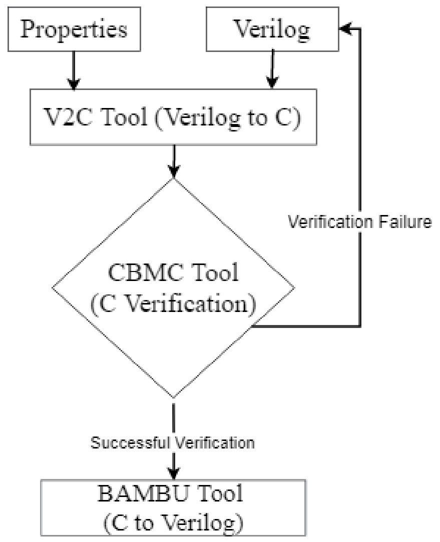

Fig. 1. Proposed Methodology

B. C Verification

Software verification is a mature field with many tools for the formal verification of the C code e.g., Blast [19], CPA-Checker [20], Sea-horn [21] and CBMC [14]. We selected CBMC to be used in our context because it is known to exhibit a better performance in terms of verification due to utilization of SAT algorithms for verification and bounded model-checking technique [11], Moreover, it includes almost all ANSI-C language features. CBMC is a user-friendly tool and supports fully automated verification. The user only has to provide the C file along with its properties for the verification of the C code.

C. C to Verilog Conversion

Like the above-mentioned cases, many tools are also available for C to Verilog conversion and we had to investigate them to find the most suitable option for the proposed methodology. These tools are usually characterized as high level synthesis tools. Some are available as open source, e.g., LegUp [22] and BAMBU [15], while most are commercial, e.g., Catapult-C [23], C2H [24] and Synphony C. LegUp is a compiler that synthesizes C code to hardware and is built within the LLVM compiler framework. BAMBU is also a high level synthesis tool that generates various implementations by trading-off latency and other resources. BAMBU utilizes a novel approach for handling the memory architecture to support complex constructs of C language. BAMBU is built within the GCC compiler and can be easily configured to various target devices and technologies. It also generates a test-bench to validate the result and Verilog code against its corresponding C code. Based on these unique features, we selected BAMBU for converting C to Verilog in the proposed methodology. The details of rest of them can be found in [25].

III. Case Studies

In order to illustrate the effectiveness and utilization of the proposed methodology for verifying Verilog models of digital circuits, we used a wide range of commonly used digital circuits, including gates, multiplexer, encoder, adder, ALU and a processor design. The proposed methodology was able to effectively formally verify all these circuits. Verification details of multiplexer, encoder, Adder, 64-bit ALU and the processor design are shared in this section.

A. Multiplexer

The input Veriog code for the 4x1 multiplexer that we verified is available at [25]. It was verified using the following properties, where the variables *a,b,c,d* represent the four inputs, *out* is the output and *s* is the select line which selects one of the inputs.

```
assert( !(s == 1) || (*mux_out == a));
assert( !(s == 2) || (*mux_out == b));
assert( !(s == 3) || (*mux_out == c));
assert( !(s == 4) || (*mux_out == d));
```

B. Encoder

The encoder circuit that we verified is a low enable module [25] i.e., whenever the *enable* signal is high, the output is always one. When the *enable* signal is low, the output follows the input as per the following properties where *in* and *enable* are input signals and *out* is output.

```
assert( (in == 32) || !(*encode_out == 60));
assert(!(enable==1 && in==32)||*encode_out==0);
```

C. Full Adder

We verified a 64-bit full adder [25] with inputs *a,b,c* and the outputs are *sum* and *carry*. The property for the *sum* output is verified as follows:

```
assert((*out_sum) || !(a + b+ c));
```

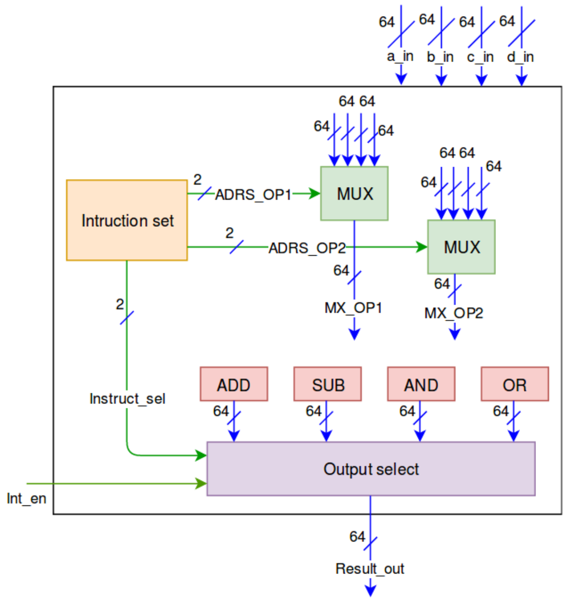

Fig. 2. Block Diagram of Processor

D. 64-Bit Arithmetic Logic Unit (ALU)

The 64-bit ALU that we formally verified using the proposed methodology supports OR, AND, Add, and Subtract operations and its code is available at [25]. The inputs are *a_in, b_in* and the output is *result_out*. The formally verified properties are:

```
assert(!(inst_sel==0) || *result_out==(a_in+b_in));
assert(!(inst_sel==1) || *result_out==(a_in-b_in));
assert(!(inst_sel==2) || *result_out==(a_in&&b_in));
assert(!(inst_sel==3) || *result_out==(a_in||b_in));
```

E. 64-Bit Processor

We also formally verified a 64-bit processor using the proposed methodology. Its code is available at [25] and the block diagram is shown in Figure 2. The processor accepts 64-bits inputs, i.e., *a_in, b_in, c_in, d_in*. These inputs are given to two multiplexers. The processor supports an instruction set, which has 2 bit *ADRS_OP1* (2 bits Operand 1), *2 bit ADRS_OP2* (2 bits Operand 2) and 2 bit of *instruct_sel* (instruction select). *ADRS_OP1* gives the address of one of the four inputs to be selected as the first operand. This *ADRS_OP1* is applied to the select line of the first multiplexer. Similarly, *ADRS_OP2* is used as the select line of the second multiplexer and thus selects the second operand of the ALU. The input *instruct_sel* is used to select one of the operations (ADD,SUB,AND,OR) of the ALU to be performed to these operands. The output of this operation is then given to the final output port, i.e.,*result_out*. The processor also has a high active reset pin, i.e., *Int_en*. Whenever *int_* en is one, *Result_out* becomes one irrespective of the given input. We verified the following properties for this processor using the proposed methodology.

```
assert(!(int_en == 0 && instruct_sel == 0
&& ADRS_OP1 == 0 && ADRS_OP2 == 2) ||
(*result_out == (a_in + c_in)));
```

```
assert(!(int_en == 0 && instruct_sel == 1
&& ADRS_OP1 == 0 && ADRS_OP2 == 3) ||
(*result_out == (a_in − d_in)));

assert(!(int_en == 0 && instruct_sel == 2
&& ADRS_OP1 == 0 && ADRS_OP2 == 3) ||
(*result_out == (a_in || d_in)));

assert(!(int_en == 0 && instruct_sel == 3
&& ADRS_OP1 == 0 && ADRS_OP2 == 3) ||
(*result_out == (a_in && d_in)));

assert(!(int_en == 1) || *result_out == 1);
```

The synthesis results for all the above-mentioned case studies are summarized in Table 1. We can observe that BAMBU synthesizes the circuits efficiently in less than one second with the maximum time consumed for the 64-bit processor is only 0.29 seconds. The successful formal verification of the 64-bit processor using the proposed methodology clearly indicates its effectiveness as this kind of a large circuits cannot be exhaustively verified using simulation. Moreover, all the verification results, reported in this section, were automatically obtained in a push button manner. This features makes the proposed methodology quite suitable for the industrial setting.

TABLE I. C CODE TO VERILOG SYNTHESIS RESULT

Circuit	Number of cycles	Area (LE)	Clock Frequency (MHz)	clock Slack (ns)	Synthesis time (s)
16-bit Logic Gates	5	767	338.4074	7.0449	0.03
16-bit Multiplexer	3	486	352.8552	7.1659	0.06
16-bit Encoder	6	1320	147.1594	3.2046	0.22
64-bit Full Adder	4	2000	323.1005	6.9049	0.07
64-bit ALU	5	1099	249.6299	5.9940	0.06
64-bit Processor	10	2016	249.6300	5.9940	0.29

IV. CONCLUSION

The paper presents an automatic formal verification technique for Verilog models while minimizing the user involvement. This technique is very useful in ensuring the correctness of IoT devices, which are widely used in many safety and mission-critical domains. We propose to use CBMC, which is a verification tool for C language, for this purpose we propose a tool chain to formally verify digital circuits using the bounded model checking technique and obtain the final result at the RTL level using high level synthesis. For illustration, we verified various commonly used digital circuits, including a 64 bit processor. These promising results illustrate the practical utilization and effectiveness of the proposed methodology.

REFERENCES

[1] O. Hasan and S. Tahar, "Formal verification methods," in *Encyclopedia of Information Science and Technology, Third Edition*. IGI Global, 2015, pp. 7162–7170.

[2] P. Sule, Y. Kim, and N. Mansouri, "Proverific: Experiments in employing (psl) standard assertions in theorem-proving-based verification," in *Circuits and Systems, 2005. 48th Midwest Symposium on*. IEEE, 2005, pp. 112–115.

[3] S. Tverdyshev, "A verified platform for a gate-level electronic control unit," in *Formal Methods in Computer-Aided Design, 2009*. IEEE, 2009, pp. 164–171.

[4] T. Braibant, "Coquet: A coq library for verifying hardware," *Certified Programs and Proofs*, vol. 7086, pp. 330–345, 2011.

[5] S. Shiraz and O. Hasan, "A hol library for hardware verification using theorem proving," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 512–516, 2018.

[6] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design,*. IEEE, 2007, pp. 173–180.

[7] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo *et al.*, "Vis: A system for verification and synthesis," in *International conference on computer aided verification*. Springer, 1996, pp. 428–432.

[8] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification*. Springer, 2010, pp. 24–40.

[9] R. Mukherjee, M. Tautschnig, and D. Kroening, "v2c–a verilog to c translator," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 580–586.

[10] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Reveal: A formal verification tool for verilog designs," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2008, pp. 343–352.

[11] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word level predicate abstraction and refinement for verifying rtl verilog," in *Design Automation Conference*. IEEE, 2005, pp. 445–450.

[12] R. Bryant, S. Lahiri, and S. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Computer Aided Verification*. Springer, 2002, pp. 106–122.

[13] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI*, 2015, pp. 7–12.

[14] E. Clarke and D. Kroening, "Hardware verification using ansi-c programs as a reference," in *Asia and South Pacific Design Automation Conference*. ACM, 2003, pp. 308–311.

[15] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *Field Programmable Logic and Applications*. IEEE, 2013, pp. 1–4.

[16] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[17] "Verilog2c++," Available online: http://verilog2cpp.sourceforge.net, 2003.

[18] D. J. Greaves, "A verilog to c compiler," in *Rapid System Prototyping, 2000*. IEEE, 2000, pp. 122–127.

[19] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[20] D. Beyer and M. E. Keremoglu, "A tool for configurable software verification," in *Computer Aided Verification. LNCS*, vol. 6806. Springer, 2011, pp. 184–190.

[21] A. Gurfinkel, T. Kahsai, and J. A. Navas, "Seahorn: A framework for verifying c programs (competition contribution)." in *Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 447–450.

[22] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[23] T. Bollaert, "Catapult synthesis: a practical introduction to interactive c synthesis," *High-Level Synthesis*, pp. 29–52, 2008.

[24] I. Nios, "C2h compiler user guide, version 9.1," 2009.

[25] Q. Ain, "Automatic formal verification of verilog models of digital circuits," Available online: http://save.seecs.nust.edu.pk/projects/afvvmdc/, 2018.