TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 11/11 pp212-218
Volume 20, Number 2, April 2015

Deadlock Detection in FPGA Design: A Practical Approach

Dexi Wang, Fei He*, Yangdong Deng, Chao Su, Ming Gu, and Jiaguang Sun

Abstract: Formal verification of VHSIC Hardware Description Language (VHDL) in Field-Programmable Gate Array (FPGA) design has been discussed for many years. In this paper we provide a practical approach to do so. We present a semi-automatic way to verify FPGA VHDL software deadlocks, especially those that reside in automata. A domain is defined to represent the VHDL modules that will be verified; these modules will be transformed into Verilog models and be verified by SMV tools. By analyzing the verification results of SMV, deadlocks can be found; after looking back to the VHDL code, the deadlocking code is located and the problem is solved. VHDL verification is particularly important in safety-critical software. As an example, our solution is applied to a Multifunction Vehicle Bus Controller (MVBC) system for a train. The safety properties were tested well in the development stage, but experienced a breakdown during the long-term software testing stage, which was mainly caused by deadlocks in the VHDL software. In this special case, we managed to locate the VHDL deadlocks and solve the problem by the FPGA deadlock detection approach provided in this paper, which demonstrates that our solution works well.

Key words: Field-Programmable Gate Array (FPGA); VHSIC Hardware Description Language (VHDL); verification; deadlocks; Multifunction Vehicle Bus Controller (MVBC)

1 Introduction

VHSIC Hardware Description Language (VHDL) is a hardware description language used in many embedded systems. While there are many formal methods for verifying VHDL, there is no widely used VHDL verification solution.

During the last two decades, many researchers have presented approaches to VHDL verification. Bawa and Encrenaz[1] developed a tool named VPN, which translates a subset of VHDL'87 into a formal model based on Interpreted and Timed Petri Nets (ITPN). Borrione et al.[2, 3] proposed to introduce mechanically supported formal reasoning in the design flow, by producing a model of VHDL behavioral specifications in the logic of the ACL2 theorem prover. Formal specification and verification of transfer-protocols for system design has been specified by Bickford and Jamsek[4] and Drgehorn et al.[5] extended them, making it easier to verify systemwide properties. Braibant and Chlipala[6] presented formal verification of hardware synthesis, which was a big step forward for VHDL verification.

The practice presented in this paper is about transforming VHDL into Verilog models, and then verifying the Verilog by using model-checking tools, such as CadenceSMV and NuSMV[7]. Bounded model-checking tools, such as SAT, are first used to perform time-saving verification and then the verification result is analyzed to find deadlocks. After finding these deadlocks, corresponding code lines are located in the VHDL sources. By debugging these source lines, the deadlock problem is solved.

Finding deadlocks in verification results is currently the only manual procedure in our approach. We see

• All authors are with the Institute of Software Theory and Systems, School of Software, Tsinghua University, Beijing 100084, China. E-mail: dx-wang12@mails.tsinghua.edu.cn; hefei@tsinghua.edu.cn; dengyd@tsinghua.edu.cn; hacker-sc@163.com; guming@tsinghua.edu.cn; sunjg@tsinghua.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2015-03-03; accepted: 2015-03-13

no value to making this procedure fully automatic; since the verification results are logs of the verification procedure, attempting an automatic analysis of textual logs may be less efficient than doing it manually. The main contributions of this paper are:

(1) An analysis of the issue whose main work is to identify possibly buggy source-code files, and seek out all automata. After finding the automata, we draw transition diagrams to help identify deadlocks in them.

(2) A general solution to locating deadlock codes by the use of server software verification tools such as Veritak, Cadence SMV, and the like.

(3) A discussion of lessons we learned when executing VHDL verification, including resolving compatibility problems, understanding verification outputs, and finding buggy code lines.

The rest of the paper is organized as follows. Section 2 describes the Multifunction Vehicle Bus Controller (MVBC) case that is verified and resolved. Section 3 shows the general solution and algorithm to verify VHDL code and find deadlocks. Section 4 shows the details and results of our experiments. Finally, Section 5 presents related work and concludes.

2 MVBC

High-speed trains are important in daily life, and represent a vital part of the economy for many countries. The Multifunction Vehicle Bus (MVB) is an important part of train communication networks. MVB controllers play a vital role in the digital operation of trains, affecting operations such as speeding up, braking, closing train doors, and so on[8]. We created our own MVB controller[9], using VHDL and C. The hardware part, that is, the layer used to design the chip logic, is in pure VHDL. VHDL is widely used in hardware manufacturing, but there is no VHDL verification tool; so little effort is made to verify VHDL code during development.

In this example, after the MVBC code was finished and released to the train manufacturer for testing, the manufacturer reported a big issue after dozens of days of testing. The issue is that some Central Control Unit (CCU) devices' lifecycles in the train head and tail timed out. The problem is obviously in the hardware programming of the MVBC, but the failure rarely occurs in daily running, so it is hard to reproduce this issue by testing. We decided to use software verification technologies to locate the problem and propose solutions to this issue. The main focus of locating buggy code is deadlocks in the VHDL automata.

The problem is that some CCU devices' lifecycles in the train head and tail timed out and this is all we know. The details of this issue in this article are omitted since they are not the main point; the issue report we received simply described lifecycle timeouts. It is a hardware device problem, and any unit of the hardware may have caused the problem. Before starting to find a solution, we established a strategy of approaching the matter by beginning at bottom-level hardware and proceeding to top-level hardware automata. Firstly, electrical and communication testing were performed on these devices; all the devices worked well. Different types of data transmission were also tested; the results showed that these devices were in good condition.

Secondly, the device monitor log was analyzed and the analysis suggested that the main cause may be deadlocks in some modules' automata.

In summary, the approach described above is used to locate deadlocks in VHDL automata, which is similar to analyzing timing properties[10]. The next section discusses detailed deadlock detection in Field-Programmable Gate Array (FPGA) design.

3 Deadlock Detection in FPGA Design

3.1 Verification domain construction

The issue circumstances are analyzed to locate buggy modules. The procedures and techniques used to locate buggy modules vary from one situation to another, because different software products may have different architectures, so one single bug may be related to one or more modules.

Locating buggy modules strongly depends on one's understanding of the target system. If a person has no strong understanding of the target system, or the bug is so deep inside the system that it is difficult to determine which module is suspicious, we can take the whole system as the verification domain.

Though the procedure of locating suspect modules cannot be automated or unified, the verification domain can still be constructed formally. The VHDL verification domain is defined as $D = \{M, V, A, P\}$, where D is the verification domain, M represents the set of the suspect modules to be verified, V is the set of Verilog models transformed from M, A stands for

automata found in M, and P is the set of properties to be verified.

From a VHDL module M, a Verilog model V can be easily obtained. There is no direct verification techniques for VHDL, but there are techniques for Verilog, and there are also tools performing the transformation.

The VHDL definition of automata is usually of the form shown in Fig. 1.

Automaton A in verification domain D is used to locate suspicious code lines from verification results.

Verification properties are Computation Tree Logic (CTL) or Linear Temporal Logic (LTL) assertions[11, 12]. In practical verification one or more properties can be verified at the same time, and one can add more properties to P according to the verification result.

3.2 VHDL code transformation

The Veritak tool is used to transform a VHDL module into a Verilog model. The input is VHDL source files, i.e., files with extension .vhd, and the output is Verilog files, i.e., files with extension .v.

The transformation procedure is fully automatic and supported by a sophisticated tool, but there may be transformation failures that should be paid attention to, especially when these Verilog files are sent to SMV solvers. These failures can be of the following types:

(1) Missing component. A sign of missing modules, leading to dependencies not being included in the verification domain factor M. It is trivial to resolve this; just add the necessary modules to M.

(2) Main model undefined. If we define a main model to instance the core module so that SMV can verify it, the other modules will be initialized automatically by Verilog itself.

(3) Redefining. An SMV solver, such as CadenceSMV, may report this error with line number and source-file name. It is not a big problem and we simply note the reported redefinition.

(4) Unkown error. An SMV solver, such as CadenceSMV, reports an error without a detailed description. This is usually caused by some initialization codes, e.g., reg [3:0] s0 = 4'b0000 in the Verilog files; we note those initialization codes to continue.

After transformation, the Verilog set P is fulfilled in the verification domain.

```
NEXT_STATE_DECODE: process
(s0, data, data_enable, sd,ssc_ack,
port_buffer, MF_out1)
  begin
    next_s0 <= s0;
    case (s0) is
      when "0000" =>
        if data_enable = '1' then
          next_s0 <="1000";
        end if;
      when "0001" =>
        if (ssc_ack='1') then
          next_s0 <= "0010";
        end if;
      when "1000" =>
        next_s0 <= "0001";
      when others =>
        next_s0 <= "0000";
    end case;
  end process;
/* other process code; */
```

Fig. 1 Example of code snippets that implement automata.

3.3 Verilog verification

CadenceSMV is used to verify the transformed Verilog model. At this point in the process, the tool is set to use bounded-model checking for the time-saving verification procedure.

The property set P contains all the properties that are going to be verified; they are all CTL or LTL assertions. Usually one automaton and its corresponding properties are verified at a time. Since multiple properties can be verified at one time, P is updated according to the verification result. We designed a strategy based on the specification of the depth of the bounded model checking. First, it is set to 30, then the verification is performed. If there is no counterexample report, for non-critical modules, we assume that the property currently being verified is satisfied. For critical properties, the depth is set to 50 and the verification is rerun. If counterexamples are reported, the depth is firstly reset to 50 to find the longest counterexample for later analysis. If the reported counterexample is proved spurious, new assertions are added into P, to remove the spurious counterexample if possible.

3.4 Results analysis

Counterexample results are analyzed to find loops and loops found in the verification results are usually ones

that cause deadlocks. Analyzing the results is simple, the loops can be found in the output files of SMV.

3.5 Locating deadlocks

After finding loops, we go back to the VHDL code to find deadlocks. The deadlocks reside in the automata A corresponding to the verification model V.

4 Learned Lessons

4.1 Verification domain construction

The verification domain is constructed by analyzing the control flow and data flow of the project, and locating one or more suspicious modules. After analysis of the issue description, all the sources of suspicious modules are checked, and the domain factor M is finalized to a list of 12. They are:

(1) decoder, the core module of the signal receiver;
(2) monitorarminterface, the monitor of the arm chip interface;
(3) monitorreceiver, the monitor of the signal receiver;
(4) monitorsenderstate, the monitor of the signal sender state;
(5) Receive_FIFO_Controller, which manages the FIFO rule of the signal receiver;
(6) ShiftRegister, the core module of the signal sender;
(7) mf_send_control_unit, the main frame sending control unit;
(8) TM_ACCESS_CTRL, the telecom memory access control module;
(9) bigmux, the receiver signal distribution module;
(10) cmf_ssc_combine, which stores the main frame and determines whether it is the signal source or destination;
(11) trafficstorecontroller, the controller of storing signals;
(12) receiver_controller_2out, the controller of signal receiver.

All these modules are subject to having deadlocks that could cause the issue described in the last section, so each of them will be verified by our solution.

4.2 Noteworthy procedure

For a well-constructed verification domain $D = \{M, V, A, P\}$, verification source V should be accurate and acceptable to SMV solvers. As discussed above in Section 3.2, the transformed Verilog files may cause failures when verified by SMV tools, so some fixes are made according to the three types described above. One example of fixing initialization code is illustrated in Fig. 2.

LTL and bounded model checking are used to verify the fixed Verilog files. For LTL assertions, "infinitely often", i.e., GF, are often used. Besides normal assertion code, it is recommended that the verifier records circumstances that are useful for later recording and referencing. Here is a global assertion that is always assumed to be true and used for verifying other properties:

mutex0: assert G F (((main_controller_with_sram_inf1.tm_ack ='1') &&(main_controller_with_sram_inf1.tm_free ='0')) U ((main_controller_with_sram_inf1.tm_ack ='0') && (main_controller_with_sram_inf1.tm_free ='1'))).

This assertion is a description of the signal ACK, which is the acknowledgement signal of the receiver of the CCU. We assume that the signal ACK can infinitely often be 1, and it is freed until it has a response.

All the other verification cases are based on this global assertion. One example of LTL assertions with comments is shown in Fig. 3.

Bounded Model Checking Depth is the SAT depth we specified for this verification. As discussed above, it

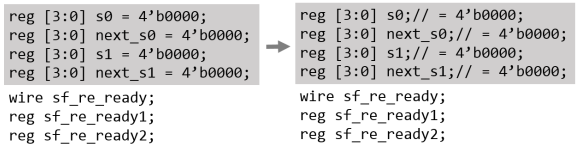


Fig. 2 Example: initialization code fix.

Verification NO.02
Bounded Model Checking Depth: 30
Verification Module: ShiftRegisters.v
Verification Property: the initial state can infinitely often go back to initial state
Additional Settings: endstore signal can infinitely often be 1
Verification Result: Passed
Result Analysis: given the setting that endstore signal can not infinitely often be
0 (mutex2), the verification is passed;
refer to verification NO.01, it is clear that as long as signal endstore always be 0 sometime,
variable state will go into deadlock.
mutex1: assert G F (main_controller.utranceiver.state ='000');
mutex2: assert G F (main_controller.utranceiver.endstore ='1');
using mutex0, mutex2 **prove** mutex1;
assume mutex0, mutex2;

Fig. 3 Example of LTL assertion.

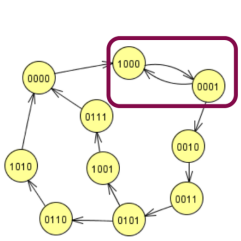
will be changed according to the verification results.

The **Verification Module** shows the Verilog model to be verified. It has the same name as its VHDL module. **Verification Property** is the verbal description of the property to be verified; for detecting deadlocks, we usually use the phase that the initial state can infinitely often return to to demonstrate that there are no deadlocks as long as the automata can always return to their initial state in the future. **Additional Settings** adds new assertions to get rid of spurious counterexamples confirmed by the hardware team. **Verification Result** shows the result as either "Passed" or "Failed". **Result Analysis** analyzes the verification result and discusses the following verification settings, either making a short summary of passed cases or providing new deeper verification settings for the current model.

It is interesting to know that the SMV tool we used, CadenceSMV, has poor support for verification result diagrams, e.g., exporting trace files or result view files does not work. We have to check output files, files with extension .out, to find deadlocks. The analysis of these failed verification results is completely manual. In the output files, there are lines saying "loop begins here" by finding loops and the state variable in the automata, and we can draw transition diagrams of the automata.

Here is an example of the analysis procedure. By performing the procedure introduced above, we find a deadlock in the module cmf ssc combine; the state in the automata runs into a loop on 0001, i.e., 1000, 0001, 1000, 0001, 1000, 0001, $\cdots$, the transition diagram and corresponding code are shown in Fig. 4.

As shown in Fig. 4, the transition from 1000 to 0001 always happens, while the transition from 0001 to 1000 is subject to conditional commands. From the source code above, we can say that due to the incompleteness of these lines of code, when ssc_ack and data_enable both equal "0", there will always be no state transition, and the automaton runs into a deadlock.




```
when "0001" =>
-- ssc_ack
  if (ssc_ack = '1') then
    next_s0 <= "0010";
  -- anti-dead lock
  elsif (data_enable = '1') then
    next_s0 <= "1000";
  end if;

when "1000" =>
-- ssc_ask
  next_s0 <= "0001";
```

Fig. 4 Example: Automata deadlock and source code in cmf ssc combine.

4.3 Verification results

All the verification cases for all 12 suspicious modules are executed one by one and the verification results are shown in Table 1.

As shown in Table 1, we verified 12 modules—8 passed, 3 skipped, and 1 failed. For the modules that passed, verification cases are rerun at depth 50 to make sure of their correctness; those marked as "Skipped" mean no automata were found in them, so no verification will be carried out. cmf_ssc_combine is the only module that failed verification.

5 Conclusions and Future Work

The verification solution works. As shown in the result, we found several deadlocks in the MVBC project. Some of them are confirmed to be impossible in the actual runtime environment, but the cmf_ssc_ combine module is found to be the root cause of the issue.

Though confirmed as "impossible", the deadlocks

Table 1 Verification results.

No.	Module	Result	Depth	Deadlock length	Comment
1	decoder	Passed	30/50	0	N/A
2	monitorarminterface	Skipped	0	0	No automata
3	monitorreceiver	Skipped	0	0	No automata
4	monitorsenderstate	Skipped	0	0	No automata
5	Receive FIFO Controller	Passed	30/50	0	N/A
6	ShiftRegister	Passed	30/50	0	N/A
7	mf send control unit	Passed	30/50	0	N/A
8	TM ACCESS CTRL	Passed	30/50	0	N/A
9	bigmux	Passed	30/50	0	N/A
10	cmf ssc combine	Failed	30/20/50/05	2/2/9/4	Deadlock found
11	trafficstorecontroller	Passed	30/50	0	N/A
12	receiver controller 2out	Passed	30/50	0	N/A

are not what the hardware team thinks they are. The deadlocks may not happen if all the hardware is in good condition, but they may occur in the future if some part of the circuit is not working, e.g., some pin is not soldered well or worn out. So it is still required that all the deadlocks be removed. It is the hardware team's responsibility to make sure the circuit works perfectly, but the software team is also obliged to prevent potential problems. Only in this way will the verification solution be meaningful.

At the same time, verification is currently limited to locating deadlocks in VHDL automata. However, many other parts of the code can be troublesome, so we plan to extend our verification to those parts in the future.

We currently use SAT and bounded-model checking to verify these automata, so we have to specify the depth of the SAT solver manually, according to our individual experience. This is not reliable and may vary from person to person. What's more, we find that it is time-consuming to design the overall verification strategy and depth dependency. To solve this problem, we plan to use Binary Decision Diagram (BDD), in addition, to verify the existing automata and we believe that this will make it more convincing and convenient, and we may find more potential risks in the VHDL software.

References

[1] R. Bawa and E. Encrenaz, A tool for translation of vhdl descriptions into a formal model and its application to formal verification and synthesis, *Lecture Notes in Computer Science*, vol. 1135, pp. 471–474, 1996. http://dx.doi.org/10.1007/3-540-61648-959.

[2] D. Borrione, P. Georgelin, and V. Rodrigues, Symbolic simulation and verification of vhdl with acl2, in *System-on-Chip Methodologies and Design Languages*, P. Ashenden, J. Mermet, and R. Seepold, Eds. Springer, 2001, pp. 59–69. http://dx.doi.org/10.1007/978-1-4757-3281-86.

[3] D. Borrione and P. Georgelin, Formal verification of vhdl using vhdl-like acl2 models, in *Electronic Chips and Systems Design Languages*, J. Mermet, Ed. Springer, 2001, pp. 273–284. http://dx.doi.org/10.1007/978-1-4757-3326-623.

[4] M. Bickford and D. Jamsek, Formal specification and verification of vhdl, *Lecture Notes in Computer Science*, vol. 1166, pp. 310–326, 1996. http://dx.doi.org/10.1007/BFb0031818.

[5] O. Drgehorn, H.-D. Hmmer, and W. Geisselhardt, Formal specification and verification of transfer-protocols for system-design in vhdl, in *Electronic Chips and Systems Design Languages*, J. Mermet, Ed. Springer, 2001, pp. 295–306. http://dx.doi.org/10.1007/978-1-4757-3326-625.

[6] T. Braibant and A. Chlipala, Formal verification of hardware synthesis, in *Proceedings of the 25th International Conference on Computer Aided Verification*, Berlin, Heidelberg: Springer-Verlag, 2013, pp. 213–228. http://dx.doi.org/10.1007/978-3-642-39799-814.

[7] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, Nusmv: A new symbolic model verifier, in *Proceedings of the 11th International Conference on Computer Aided Verification*, London, UK: Springer-Verlag, 1999, pp. 495–499. http://dl.acm.org/citation.cfm?id=647768.733923.

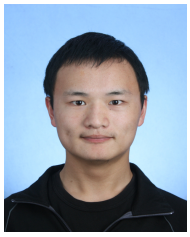
[8] L. Zuo, Q. Ding, and Y. Tang, Design and implement of mvb bus controller based fpga, in *Proceedings of the 2010 International Conference on Communications and Intelligence Information Security*, Washington, DC, USA: IEEE Computer Society, 2010, pp. 232–235. http://dx.doi.org/10.1109/ICCIIS.2010.36.

[9] Y. Jiang, Z. Li, H. Zhang, Y. Deng, X. Song, M. Gu, and J. Sun, Design and optimization of multi-clocked embedded systems using formal technique, in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA: ACM, 2013, pp. 703–706. http://dx.doi.org/10.1145/2491411.2494575.

[10] C. Courcoubetis, W. Damm, and B. Josko, Verification of timing properties of vhdl, *Lecture Notes in Computer Science*, vol. 697, pp. 225–236, 1993. http://dx.doi.org/10.1007/3-540-56922-719.

[11] K. Y. Rozier and M. Y. Vardi, Ltl satisfiability checking, in *Proceedings of the 14th International SPIN Conference on Model Checking Software*, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 149–167. http://dl.acm.org/citation.cfm?id=1770532.1770548.

[12] K. Y. Rozier, Survey: Linear temporal logic symbolic model checking, *Comput. Sci. Rev.*, vol. 5, no. 2, pp. 163-203, 2011. http://dx.doi.org/10.1016/j.cosrev.2010.06.002.



Dexi Wang received his BS degree from Tongji University, China in 2012. He is currently working toward the PhD degree in computer science at Tsinghua University, China. His research interests include formal verification, model checking, and their applications in embedded systems.

Fei He received his BS degree from National University of Defence Technology, China, in 2002, and PhD degree in computer science from Tsinghua University, China in 2008. He is currently an associate professor in School of Software, Tsinghua University. His research interests include formal verification, model checking, and their applications in embedded systems.

Chao Su received his BS degree from Tianjin University of Science & Technology, China in 2011, and MS degree in software engineering from Tsinghua University, China in 2014. His research interests include formal verification, model checking, and their applications in embedded systems.

Yangdong Deng received his BS and MS degrees from Tsinghua University, China in 1995 and 1998, respectively and PhD degree in electrical and computer engineering from Carnegie Mellon University Pittsburgh, PA, USA in 2006. His research interests include parallel electronic-design-automation algorithms, electronic-system-level design, and parallel program optimization.

Ming Gu received her BS degree from National University of Defense Technology, China in 1984, and MS degree in computer science from the Chinese Academy of Sciences in 1986. Since 1993, she has been working as a professor at Tsinghua University, China. Her research interests include formal methods, middleware technology, and distributed applications.

Jiaguang Sun received his BS degree in automation science from Tsinghua University, China in 1970. He is currently a professor at Tsinghua University and the Director of the School of Information Science and Technology of Tsinghua University.