FORMAL VERIFICATION FOR SPACEWIRE LINK INTERFACE USING MODEL CHECKING

Session: SpaceWire Test and Verification (Poster)

Long Paper

Zhiquan Dai

College of Information Engineering, Capital Normal University, Beijing, China

Limin Tao

Beijing Engineering Research Center of High Reliable Embedded System

Beijing, China, 100048

Liya Liu

Dept. of Electrical and Computer Engineering, Concordia University

1455 de Maisonneuve W., Montreal, Quebec, H3H 1M8, Canada

Yong Guan, Weigong Zhang, Yuanyuan Shang, ShengZhen Jin

College of Information Engineering, Capital Normal University, Beijing, China

E-mail: guanyong@mail.cnu.edu.cn, woyun_23@163.com, liy_liu@ece.concordia.ca, zwg771@yahoo.com, syy@bao.ac.cn

ABSTRACT

The design of the SpaceWire based satellite onboard system circuits was a part of the job in the development of Space Solar Telescope (SST) project, which has been completed by National Astronomical Observatories, Chinese Academic of Sciences. In order to prove the circuit was faithfully implements the SpaceWire protocol's specification, formal verification techniques were applied during the process of development of the circuits and automated model checking approach was employed. The implementation designed as VHDL models on the FPGA for SpaceWire link interface circuit under investigation has an extension state (Error Analysis) in the state diagram providing link initialization, normal operation and error recovery services between transmitter and receiver on exchange level. Some properties were checked successfully on the original model by using Cadence SMV tool and some properties were verified to false. The results of the verification showed we have to update the design according to the counterexamples to guarantee the circuit design implemented on FPGA is reliable and can be integrated in the SST project.

1 INTRODUCTION

The correctness of design is one of the key problems to large-scale complex digital system design, namely, design verification. Unfortunately, the complexity of the verification exponentially increases with the increasing scale of chip. Particularly, as complex the state of the art is, the cost of trial-produce is quite expensive. Safety is the first place for many very important systems, for instance, the railway signal, nuclear power station, aerospace, national security and large communication system [1]. Any mistake of design possibly causes huge economic losses or catastrophic consequences as personnel casualties. The design of the SpaceWire based satellite onboard system circuits was a part of the job in the development of Space Solar Telescope (SST) project, which has been completed by National Astronomical Observatories, Chinese Academic of Sciences. In order to prove the circuit designed for the highly reliable communication based on SpaceWire protocol was faithfully implements the SpaceWire protocol's specification, this study aimed to verify the SpaceWire link interface, which was one of the important elements of the SpaceWire. Formal verification techniques were applied during the process of development of the circuits and automated model checking approach was employed.

Techniques for automatic formal verification of finite state transition systems have developed in the last 30 years to the point where major chip design companies are beginning to integrate them in their normal quality assurance process. The most widely use of these methods is called Model Checking [9]. In model checking, the design to be verified is modelled as a finite state machine, and the specification is formalized by writing temporal logic properties. The reachable states of the design are then traversed in order to verify the properties. In case that the property fails, a counter example is generated in the form of a sequence of states [7]. In general, properties are classified to "safety" and "liveness" properties. The former declares what should not happen or what should always happen; the latter declares what should eventually happen. Specification is a process to briefly express the design system and its properties with formal language. Formal specification description language has strict syntax and semantics, which are used to express the functional behavior of the system, such as timing characteristics or internal structure.

The main fault of traditional testing and simulation verification is that they are incomplete. In another words, they can only prove that the design has error but can not guarantee the design has no error. So, they are often suitable to find the vast or obvious errors in the initial verification, but not to find complex and subtle errors [1]. The main advantage of the formal verification is completeness. Through model checking, a method of the formal verification, we can find complex or subtle design mistakes which other methods cannot find. So, model checking is an effective way of the computer system design verification.

2 APPROACH

2.1 VERIFICATION FLOW

The overall flow of our approach is depicted in Figure1. The verification flow is also applicable for other classes of circuit verified by model checking methods.

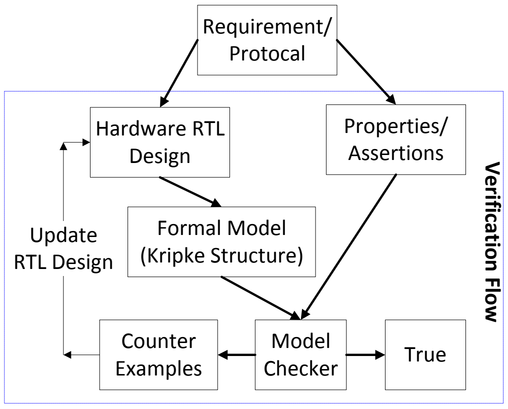

Figure1. Formal Verification Flow

The inputs to the process are the RTL description of the circuit, a formal specification (possibly comprising many properties/assertions). We elaborate on the latter point, for Properties/Assertions of SpaceWire control, in Section 3.2. The formal SpecaWire control model is automatically compiled into a finite state machine [11].

The RTL is translated into a formal model of the circuit, either manually or using automated tools. For the work in this paper, this is a description in the input language of a model checker. (The model checker Cadence SMV [2] includes an automated translator from Verilog to its input format.) However, in general it would depend on the formal verification (FV) tool that is used. If the verification result is false, then update the RTL design according to the counterexamples generated by the model checker automatically.

2.2 FORMAL MODEL

The hardware engineer's design is usually some sort of a finite automaton. Independent of the concrete design language, this finite automaton can be represented by a kripke structure, which is the standard representation of models in the model checking literature [9]. The kripke structure is a quintuple K=(S, S_0, R, AP, L), where S is the finite state set of all the Boolean state variables $\{s_1, s_2 \cdots s_n\}$, $S_0 \subseteq S$, denoting the set of initial states in which the circuit can begin operation, R is the transition relation of the system defining how the system evolves over time, AP is the set of all the atomic proposition and its negative proposition, and L is the marking function which maps the state $s \in S$ into the true atomic proposition set of S [1]. We can also regard K as a marked directed graph with a root, S is the vertex set of the graph, R is the edge set of the graph, L is marking function of the vertex, and the root is s_0.

Given an RTL-level circuit designed with hardware description language, for example Verilog, a formal model will be created automatically by the X-HDL tool as mentioned above [10]. Timing-related details in the RTL are modelled using non-determinism, so that the resulting formal model exhibits a superset of the actual system behaviours. Any verification performed on the formal model will then be faithful with the specification.

3 SPACEWIRE CONTROL MODULE

SpaceWire[8] is a network for space applications composed of nodes and routers interconnected through bi-directional high speed data links. According to the SpaceWire website hosted by the ESA, it has been used in missions of the ESA as well as space agencies NASA and JAXA. The SpaceWire standard [8] describes 6 protocol levels: physical, signal, character, exchange, packet, and network. In this paper, we concern with the exchange level that defines the protocol for link initialization, flow control, and link error detection and recovery (similar to the more widely known Transmission Control Protocol, TCP). Our main case study is the SpaceWire control module of a node in the SpaceWire network, which is implemented by our group in VHDL description language. Unfortunately, as VHDL is not the required input language of any model checking tool available to us at this moment, the design is translated from VHDL into Verilog by X-HDL [10]. With aid of this tool, code in Verilog was automatically translated into the input code with the acceptable language for the Cadence SMV model checker. In the mean time, English language specifications from the standard document [8] were translated into formal specifications in linear temporal logic and inserted into the SMV file as assertions to be checked [3].

3.1 MODEL

A SpaceWire end node comprises three modules: a transmitter (TX), a receiver (RX), and a state machine that sends control signals to them (FSM). We abstracted the control module code from our whole design. Generating a SMV model from Verilog involved straightforward transition for the most part, retaining the control structure, and only abstracting away some data and timing in the Cadence SMV checking tool [2]. FSM module indicating how state was abstracted to be the SMV model is briefly described the following part [4]. Further details may be found in the standard document [8].

The FSM controls the overall operation of the end node. Its operation is shown in Figure2. The sequence of state ErrorReset, ErrorWait, and Ready provide a mechanism of initializing the SpaceWire node, either coming from a whole system reset or triggered by an error. During this sequence of operation, RX is enabled to receive, but TX is prohibited from sending. In the Started state, TX can send NULL signals to the other end, to establish a connection. Next, the FSM enters the Connecting state where TX is enabled to send flow control tokens (FCTs). When RX receives FCTs, it indicates that the other end has space in its receive buffer for data. The Run state is the state for normal operation where packets flow freely in both directions across the link. The node remains in the Run state until an error occurs or until the link is disabled [8]. An ErrAnalysis_DataSave state was added in order to improve the error analysis and process ability. When an error occurs or the link is disabled in the run state, FSM enter into ErrAnalysis_DataSave state. In the same time, FSM save and analyze the error and the data. If the data has been saved and the error has been read, then the FSM enter into ErrorReset state, or still in the ErrAnalysis_DataSave state.

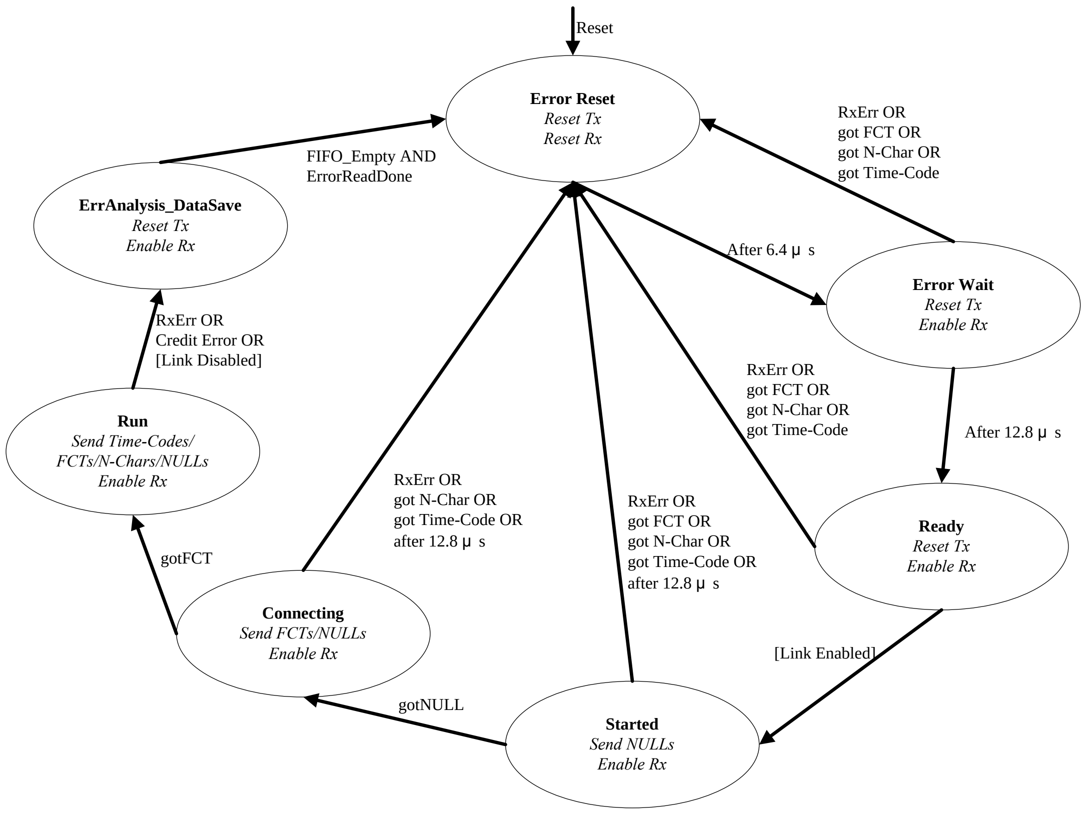

Figure2. SpaceWire Control Module State Graph

The end nodes communicate over a channel that was modelled in SMV to be capable of dropping or creating parity errors in both control and data packets. (Appropriate "fairness" constraints [6] were imposed on the channel to ensure that a packet would eventually get to its destination, even if it is dropped several times.)

3.2 FORMAL SPECIFICATIONS

25 SMV assertions were added in linear temporal logic corresponding to the specification written in nature language in the protocol [8]. Temporal logic formulas presented as formulas in ordinary Boolean logic, except that true value of a formula in temporal logic is a function of time. Some new operators are added to the traditional Boolean operators "and", "or", "not" and "implies", in order to specify relationships in time. The new operators are termed as tense operator consists of *G* (global), *F* (future), *X* (next) and *U* (until). *G p* means that *p* will keep true all the times in the future and is

read as "eventually *p*". The formula *F p* express *p* must hold true at some time in the future and is read as "eventually *p*". In addition, we have the "until" operator and the "next time" operator. The formula *X p* means that *p* will be true at the next time and is read as "next *p*". The formula *p U q* means that *q* is eventually true, and until then, *p* must always be true and is read as "*p* until *q*" [3].

No	Reference in[9]	Assertion
1	Sec. 8.5.2.2(c)	assert G(!Reset & After64 &SpacewireControllerCurrentState = 0 -> X(SpacewireControllerCurrentState = 1)); --When the reset signal is de-asserted the ErrorReset state shall be left unconditionally after a delay of 6,4 μs (nominal) and the state machine shall move to the ErrorWait state.
2	Sec. 8.5.2.3(b)	assert G(SpacewireControllerCurrentState = 1 -> X(!RX_Reset & TX_Reset)); -- In the ErrorWait state the receiver shall be enabled and the transmitter shall be reset.
3	Sec. 8.5.2.4(a)	assert G(X(SpacewireControllerCurrentState = 2) ->(SpacewireControllerCurrentState = 1 | SpacewireControllerCurrentState = 2)); --The Ready state shall be entered only from the ErrorWait state.
4	Sec. 8.5.2.5(g)	assert G(SpacewireControllerCurrentState = 3 & (DisconnectionError | FirstNULLreceived_internal & (RX_Error | RX_GotSomethingWrong)) -> X(SpacewireControllerCurrentState = 0)); --If, while in the Started state, a disconnection error is detected, or if after the gotNULL condition is set, a parity error or escape error occurs, or any character other than a NULL is received, then the state machine shall move to the ErrorReset state.

Table1. Selected Formal Specification

Table1 lists the representative assertions. Specifications are classified into four categories, and each category is represented in the table. The first set of specifications is on the FSM operation, indicating how and when the system can move between FSM states, as shown in Figure2. The second set is related on the cases whether the transmitter and receiver are enabled or not. The third presents the situations that current state is transferred from itself or the previous state. The fourth is based on the interaction between FSM, TX, and RX, exemplified by row 4 in the table that deals with error handling. Our formal specification is as comprehensive as the corresponding English language specifications in the standard documents.

3.3 RESULTS

An SMV model with 333 lines code (including assertions) was generated based on a design with 511 lines code Verilog language. According to the SpaceWire protocol, the formal specifications are created, the state transition properties indicating that the current state is from itself or the previous state is verified to be true. The error handling properties are verified to be false. For instance, the assertion, indicating that if any error occurs, the ErrorWait state will move into the ErrorReset state in the SpaceWire standard document section 8.5.2.3 (e), is verified to be false. The counterexample of it shows that the error occurs, but the FSM do not move into the ErrorReset state. It might be the result of the case that our assertion is not stated as the specification of SpaceWire protocol or the RTL designed by our group really has some error. We will continue to consummate our formal verification and update our RTL design according to the verification results.

4 CONCLUSIONS AND FUTURE WORK

The translation from VHDL to SMV is automatically. Some properties of SpaceWire control module were verified successfully, and the remaining properties will be verified in the future. Although the system of assertions is quite comfortable, we have to study the informal descriptions of parts of the design and often formulate our own assertions to verify the design, because hardware designers are often not aware of all presumptions they use to believe that their source codes are correct. Although the model checking verification is complete, it is easy to generate the state space combination explosion problem. Users of model checking tools typically consider it a compliment to the traditional methods of testing and simulation, and not as an alternative.

ACKNOWLEDGMENTS

We appreciate Kenneth L. McMillan for help with Cadence SMV. And we would like to acknowledge our group for providing the SpaceWire RTL design and Maning Wan for helping us with understanding the SpaceWire protocol. This research was supported in part by the Space Solar Telescope (SST) project.

REFERENCES

1. Han Jungang, Du Huiming, “Digital hardware formal verification”, Peking University press, 2001.
2. Cadence SMV model checker, http://www.kenmcmil.com/smv.html.
3. K.L.McMillan, “Getting started with SMV”, SMV Reference Manual, Cadence Berkeley Labs, Berkeley, 1999.
4. K.L.McMillan, “The SMV language”, SMV Reference Manual, Cadence Berkeley Labs, Berkeley, 1999.
5. K.L.McMillan, “The Model Checking System” SMV Reference Manual, Cadence Berkeley Labs, Berkeley, 2002.

6. K.L.McMillan, “Symbolic Model Checking”, Kluwer Academic Publishers, 1992.

7. E. M. Clarke, O. Grumberg, D. A. Peled, “Model Checking”, MIT Press, 2000.

8. European Cooperation for Space Standardization, “Space engineering. SpaceWire - links, nodes, routers, and networks”(ECSS-E-50-12A), http://www.ecss.nl/forums/ecss/dispatch.cgi/standards/showFile/100302/d20090722143301/No/ECSS-E-50-12A(24January2003).pdf, 2003.

9. Sanjit A. Seshia, Wenchao Li, Subhasish Mitra. “Verification-guided soft error resilience”, Design, Automation, and Test in Europe, 2007, Pages: 1442 – 1447.

10. Tomáš Kratochvíla, Vojtěch Řehák, Pavel Šimeček, “Verification of COMBO6 VHDL Design”, CESNET, 2003.

11. Tahar, “Temporal Logics and Model Checking”, http://users.encs.concordia.ca

    /~tahar/coen7501/notes/3-mc-02.05.pdf.