FVP: A Formal Verification Platform for SoC*

Wen-Shiu Liao and Pao-Ann Hsiung

Department of Computer Science and Information Engineering,
National Chung Cheng University, Chiayi, Taiwan, ROC
E-mail: hpa@computer.org

Abstract

How to verify a System-on-a-Chip (SoC) has been an important issue in an SoC design process due to its complexity. The capacity of traditional verification techniques such as simulation or emulation is no longer suitable for SoC. However, formal verification that provides 100% coverage and counterexamples is expected to be a complementary solution. Several researches on formally verifying an SoC have demonstrated its feasibility and benefits. Nevertheless, there is no utility for platform based formal verification of SoC as yet. A Formal Verification Platform (FVP) is proposed to formally verify an Intellectual Property (IP) by providing a formal platform to create its environment. We will illustrate our modeling experiences using the model checker SGM.

Introduction

With the ever-increasing capacity of integrating gates into chips, *System-on-a-Chip* (SoC) design is getting more and more popular. However, verification has been a serious problem in the SoC design. The traditional verification methods suffer from huge numbers of test vectors and are becoming inefficient. Formal verification, in contrast to traditional verification methods, provides more confidences by exhaustively traversing whole system state spaces. When a system fails to satisfy a property, formal verification produces a counterexample that is very attractive to system designers. Several efforts on formally verifying hardware or SoC (1), (2), (3), (4), (5), (6), (7), (8) indicate that verifying a large system by using formal methods is feasible although formal verification has the state space explosion problem.

Often, in a platform-based SoC design, a system designer integrates some pre-designed and verified intellectual properties (IP) with one or more user designed IP. In order to efficiently reuse IPs in an SoC, an on-chip bus architecture such as AMBA or CoreConnect is essential. Fig. 1 illustrates a typical architecture of an SoC. The architecture is composed by two buses: the processor local bus (PLB) and the on-chip peripheral bus (OPB). Each bus has some IPs attached and a bus arbiter to control the bus. The bus bridge enables data communication between the buses. For formally verifying a user-designed IP, a designer needs to also model the SoC environment in which the IP will be embedded. Thus, it is desirable that we have a platform on which formal verification can be done.

To complement a system designer's effort in formally verifying an IP, a *Formal Verification Platform* (FVP) is proposed. Our goals are as follows:

1. Configurability: Through a system configurator in FVP, the user can create an SoC environment that interacts with a user designed IP of any scale and at different abstraction levels.
2. Flexibility and Extendibility: Due to the configurability of FVP, the user can decide how much detail can be verified and to customize the SoC environment in the verification process.
3. Verification Re-Use: When an IP model is formally verified, it can be re-used in the future by including it into the database of IP formal models in FVP.
4. Cycle Accuracy: FVP supports hardware behavior modeling techniques such that cycle accuracy is guaranteed in the SoC model and counterexamples can be generated by FVP.

The article organization is as follows. Section 2 describes our formal verification platform. Section 3 will discuss in detail how to model the components on the platform. Section 4 gives the final conclusions with future work.

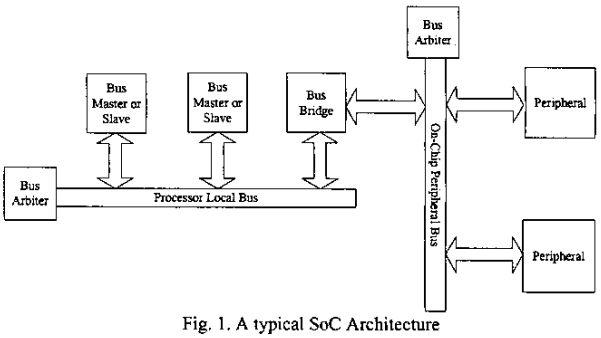

Fig. 1. A typical SoC Architecture

Formal Verification Platform

Due to the inefficiency of traditional hardware verification methods and the increase in design complexity, formal verification is getting more and more attention in recent years. As summarized below, there are some issues unresearched in the current state-of-art on formally verifying an SoC. (a) Some model checkers like SMV has only one implicit clock, a designer has to make some tricks when modeling an SoC with two bus frequencies or gated clocks (1). (b) Currently, there is no utility for platform-based formal verification of SoC. (c)

*This work was supported by project grant NSC 91-2215-E-194-008 from the National Science Council, Taiwan, ROC.

The state space explosion issue is still a major bottleneck in formal verification.

To handle above issues, we choose *State Graph Manipulators* (SGM) (9) as the verification kernel for our FVP. SGM is a high-level compositional model checker with multiple state-space reduction techniques for the verification of real time systems. In SGM, a system is described by a set of communication extended timed automata (TA) (9) and a property is specified by *Timed Computation Tree Logic* (TCTL) (9). We can model hardware behavior with multiple clocks or a gated clock in the timed automata model. In SGM, the global system state-space is computed iteratively by composing one timed automaton at a time. Thus, it is natural to treat an IP as a timed automaton in SGM.

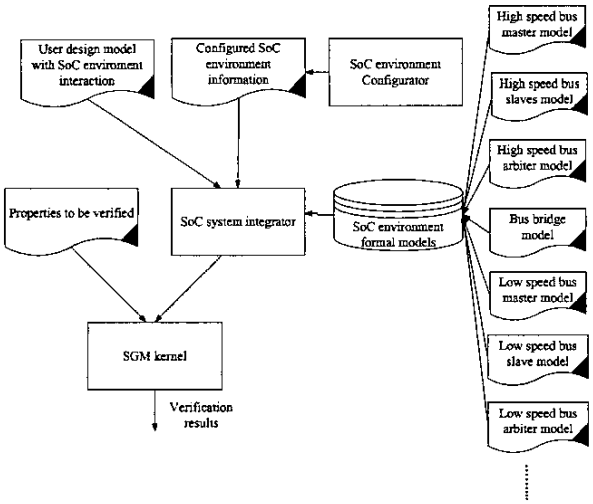

Fig. 2. Block diagram of FVP

Fig. 2 illustrates the block diagram of FVP. FVP consists of three components: *SoC environment configurator*, *SoC system integrator,* and *SGM kernel.* The given order is the sequence in which they are used. A user sets up an SoC environment using the *SoC environment configurator*. After the user finishes modeling a user-designed IP, the *SoC system integrator* is involved to integrate the user-design IP model with all the formal IP models that were specified through the *SoC environment configurator*. As soon as the whole SoC model is integrated, the user can formally verify the SoC using the *SGM kernel* against some user-specified properties.

The detailed functionalities of the *SoC Environment Configurator*, *SoC System Integrator,* and the *SGM Kernel* are described as follows.

SoC Environment Configurator: Through the *SoC environment configurator,* an IP designer can setup all the IPs that constitute an SoC environment for the designer's own IP model. This configured environment information is stored for later use during the actual SoC integration phase. For example, a user may select two buses: PLB and OPB, two PLB masters, one bridge, two PLB slaves, and three OPB slaves as an SoC environment for an image-processing IP to be attached on the CoreConnect Bus Architecture.

SoC System Integrator: According to the information saved by *SoC environment configurator*, the *SoC system integrator* composes an SoC consisting of the user-designed IP model and all the user-specified formal IP models. Because the TA model is a closed one, an external environment model is required, for which a *formal bench* (a-kind-of *testbench)* is generated by the *SoC system integrator.*

SGM Kernel: The formal model of each IP in SoC is expressed as a Timed Automaton. The *SGM kernel* reads these automata sequentially and merges them one by one to generate the global system behavior of the SoC. The user can use TCTL to specify the properties that they are interested. SGM then uses symbolic model checking technology to verify if the set of TA satisfies a given TCTL property. If the TA does not satisfy a TCTL, a counterexample is produced by SGM. The user can analyze the counterexample to locate the problem.

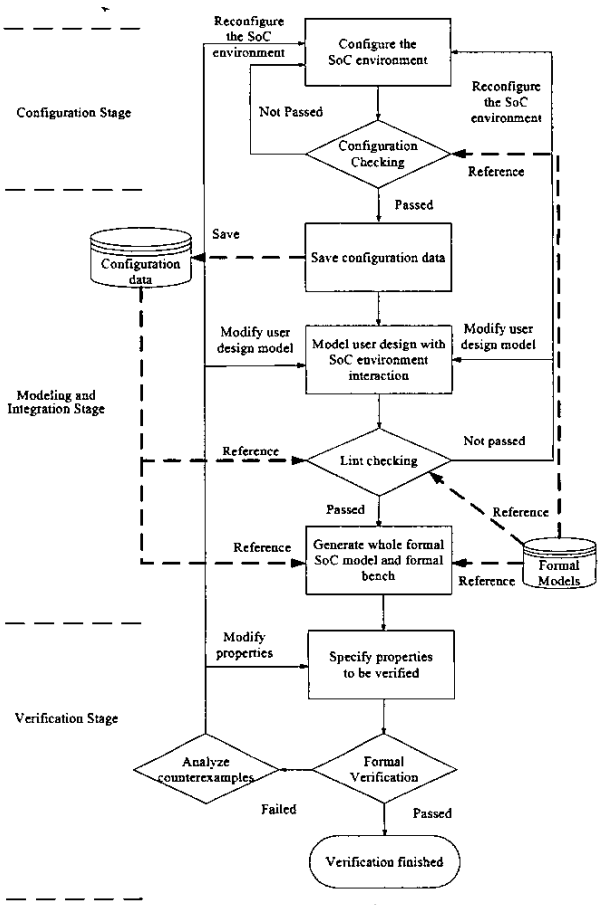

Fig. 3. Verification flow of FVP

The verification flow of FVP is shown in Fig. 3. There are three stages in the verification flow of FVP: configuration stage, modeling and integration stage, and verification stage. In the configuration stage, an FVP user begins by specifying his/her SoC environment. After an SoC environment is specified, the *SoC environment configurator* will check if there is any

conflict among the IPs in the assigned SoC. Before integrating the SoC in the beginning of model and integrate stage, lint checking is provided to avoid any syntax error in the user-designed IP model and to avoid any inappropriate interactions of the user IP model with the formal models. If lint checking fails because of syntax errors, the user must then modify his/her model. The user may also re-configure the SoC environment to correct any inappropriate use of the formal models. After lint checking has passed, the *SoC system integrator* will generate the final SoC models and a formal bench according to the configuration data, existing formal models, and the user design model. In the verification stage, the user can start verifying the whole SoC model using the SGM kernel. Either a wrong property specified in SGM, or a wrong configuration, or some bugs in the user-designed IP model may cause the formal verification to fail in SGM. If verification fails, the user must analyze counterexamples manually to decide how to re-formulate properties, to re-configure the SoC environment, or to modify the user-designed IP model.

Modeling Details

In configuring and integrating an SoC environment for the formal verification of an IP in FVP, the timed automata model which is *closed* one must be extended into an open model by associating an interface with each automaton such as that in timed modules (10), interface automata (11), and I/O automata (12). Formal IP models in FVP can communicate with each other through the interface that is provided by SGM. The interface of SGM is composed of shared variables, and synchronization labels. The shared variables represent meaningful data in the global system. There are two types of variables: register variables and clock variables. Register variables provide numeric information in the global system and clock variables provide timing information. The synchronization labels model the synchronous behavior among different TA, that is to say, two or more transitions must be taken at the same time. When there exist two transitions that share the same synchronization label in two different TA, SGM will merge these two transitions into one transition in the product state graph.

In the rest of this section, we illustrate how four signal-related behavior are modeled in FVP using SGM, namely, multi-rate clocks, gated clocks, wait-states and timeout in an IBM CoreConnect bus arbiter, and a timed behavior of signal.

Bus clock is the key issue in the formal modeling of IPs. Many formal verification tools have limits in modeling hardware system with multiple bus clocks or gated clock because they only have one implicit clock, e.g. SMV (1). We show how SGM can handle system models with multiple bus clocks or a gated clock. Fig. 4 illustrates an example of modeling two bus clocks. We use clock variables X and Y to represent the progress of time in the two bus clocks, and register variables *CLK_A* and *CLK_B* to represent the signal output of the two bus clocks. In the timed automata A and B, $A0$ ($B0$) and $A1$ ($B1$) represent the states of the bus clocks driven low and high, respectively. The transitions from $A0$ ($B0$) to $A1$ ($B1$) represent the change of bus clock level from low to high, which is the positive edge of the bus clock signal. Similarly, the transitions from $A1$ ($B1$) to $A0$ ($B0$) represent the negative edge of the bus clock. The different rates of multiple bus clocks in SoC can be modeled by setting different values to the clock variables X and Y on the transitions. For example, the rate of clock *CLK_B* is twice that of clock *CLK_A* in Fig. 4. The initialization of bus clocks can be modeled by setting the initial state of the automata. If transitions in some formal IP models need to be triggered at the positive edge of *CLK_A*, the user can synchronize these transitions with the clock transition from $A0$ to $A1$ in order to guarantee that these transitions will be triggered at the same time.

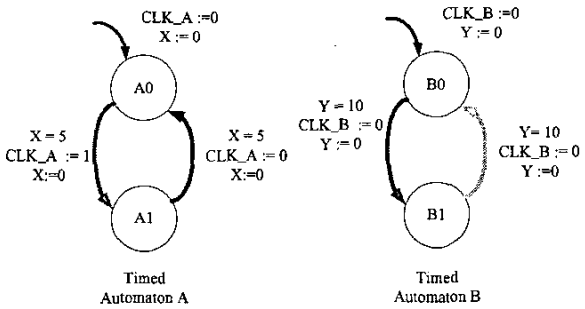

Fig. 4. Two bus clocks model

Fig. 5 is a gated clock model. Gated clock is a common technique for low power design in hardware. Register variable G is a control signal that enables/disables the clock. When G is one the clock operates normally. When G becomes zero, the timed automata will eventually stay in Low-power mode until G becomes one, which represents the gated condition of a clock.

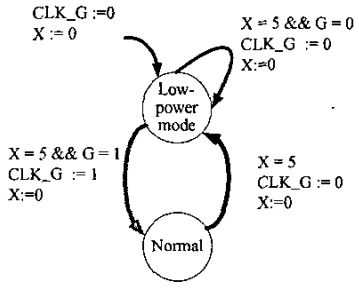

Fig. 5. Gated clock model

Fig. 6 illustrates transfer phase and termination phase in non-address pipelining of an IBM CoreConnect PLB (13) bus arbiter. The PLB address cycle consists of three phases: request, transfer, and termination. The termination conditions of termination phase are either a bus slave asserts SI_addrAck, or SI_rearbitrate, or a bus master asserts Mn_abort, or a bus arbiter checking if the transaction has timed-out. We only illustrate time-out and SI_addrAck of termination phase in Fig.

6. A PLB_PAValid signal will be asserted high when the arbiter grants the bus to a master at the beginning of the transfer phase. The maximum length of the transfer phase is controlled by the slave's SI_wait signal and by the PLB arbiter address cycle time-out mechanism. The slave that a master wants to access may assert SI_wait signal high to indicate that it is unable to latch the address and all of transfer qualifiers at the end of current cycles. If the SI_wait is asserted, the bus arbiter will continue to drive PLB_PAValid as well as the address and transfer qualifier signals until the slave asserts the SI_addrAck signal. A register variable C counts the number of bus clock cycles in Fig. 6. C will start counting as soon as the bus arbiter grants the master's request by asserting PLB_PAValid to high. If the slave does not assert SI_wait and has not responded within 15 bus clock cycles, then the arbiter will assert PLB_mnAddrAck high to indicate time-out. Then the transfer will be terminated at the 16^{th} bus clock cycle. If a slave acknowledges the bus arbiter by asserting SI_addrAck, the address cycle is terminated and C will be reset.

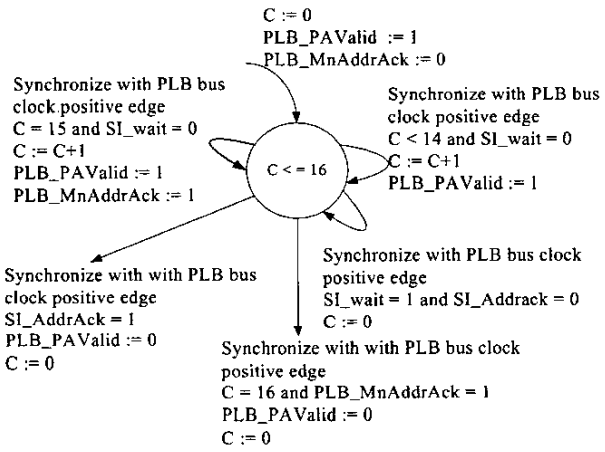

Fig. 6. Transfer phase and termination phase in non-address pipelining of an IBM CoreConnect PLB bus arbiter

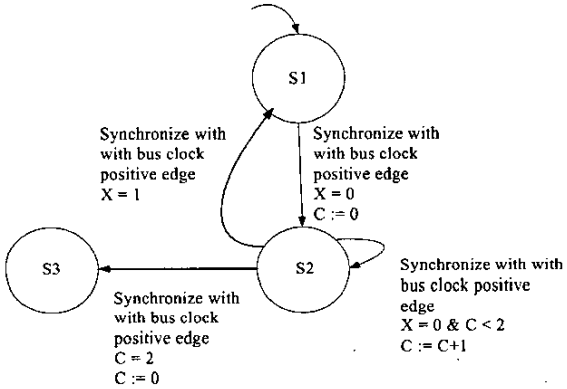

Fig. 7. Timed input signal behavior

To check if a signal X is asserted low for at least three clock cycles such as a reset signal in most systems. We can model this system behavior as in Fig. 7. If the system reaches state S3, then the condition is satisfied.

Conclusion

A formal verification platform (FVP) is proposed for the rapid prototyping and verification of a timed automata-based SoC environment such that any user-designed and modeled IP can be integrated into the environment and its behavior verified against on-chip bus protocols such as AMBA, CoreConnect, and Wishbone. Work on enhancing FVP is still undergoing. Future directions include a more cycle-accurate model of the components, transaction-based modeling and verification, assertion-based formal bench development, and formal bus wrapping technology.

References

(1) H. Choi, B. Yun, Y. Lee, and H. Roh, "Model checking of S3C2400X industrial embedded SOC product," *Design Automation Conference*, pp. 611 –616, 2001.

(2) P. Chauhan, E. M. Clarke, Y. Lu, and D. Wang, "Verifying IP-core based system-on-chip designs," *Proc. of the 12th IEEE International ASIC/SOC Conference*, pp. 27-31, 1999.

(3) B. Wang and Z. Lin, "Formal verification of embedded SoC," *ASIC Proc. of the 4th International Conference*, pp. 769 –772, 2001.

(4) A. Goel and W. R. Lee, "Formal verification of an IBM coreconnect processor local bus arbiter core," *Design Automation Conference*, pp. 196 –200. 2000.

(5) K. Takayama, T. Satoh, T. Nakata, and F. Hirose, "An approach to verify a large-scale system-on-a-chip using symbolic model checking," *Proc. of the International Conference on Computer Design (ICCD)*, pp. 308-313, 1998.

(6) P.-A. Hsiung, "Embedded software verification in hardware-software codesign," *Journal of Systems Architecture*, Vol. 46, No. 15, pp. 1435-1450, Elsevier Science, December 2000.

(7) P.-A. Hsiung, "Hardware-software timing coverification of concurrent embedded real-time systems," *IEE Proceedings — Computers and Digital Techniques*, Vol. 147, No. 2, pp. 81-90, March 2000.

(8) P.-A. Hsiung and S.-Y. Cheng, "Automating formal modular verification of asynchronous real-time embedded systems," *Proc. of the 16th International Conference on VLSI Design*, IEEE CS Press, pp. 249-254, January 2003.

(9) F. Wang and P.-A. Hsiung, "Efficient and user-friendly verification," *IEEE Transactions on Computers*, Vol. 51, No. 1, pp. 61-83, January 2002.

(10) R. Alur and T.A. Henzinger, "Modularity for timed and hybrid systems," *Proc. of the 9th International Conference on Concurrency Theory*, LNCS 1243, pp. 74-88, 1997.

(11) L. de Alfaro and T. A. Henzinger, "Interface automata," *Proc. of the 9th Annual ACM Symposium on Foundations of Software Engineering (FSE)*, 2001.

(12) N. Lynch and M. Tuttle, "Hierarchical correctness proofs for distributed algorithms," *Proc. of the 6th ACM Symposium on Principles of Distributed Computing*, pp. 137-151, 1987.

(13) Processor Local Bus (32-bit), http://www-3.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_32-bit_Implementation.