Formal Verification of a Bus Structure Modeled in SystemC

Ali Habibi, Sofiène Tahar and Lazhar Halleb
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve, West,
Montreal, Quebec H3G 1M8
Email: habibi,tahar,halleb@ece.concordia.ca

Abstract—**In this paper, we present the formal verification of a bus structure modeled in SystemC. SystemC is an emerging system level design and verification language based on C++ object oriented paradigms. The verification approach is based on both abstract interpretation (for model reduction) followed by model checking of some of the bus properties. In the abstraction phase, we statically analyze the SystemC model considered as C++ code augmented by library constructors, components and entities. We also provide a graphical representation of the reduced model, suitable for debugging and verification purposes. We use the Cadence FormalCheck tool to verify designs properties on the abstracted (reduced) bus model translated into Verilog code. While the verification of the original model was not possible to perform, we succeeded in checking all properties on the reduced model.**

I. Introduction

A decade ago, the EDA industry went progressively from gate level to register-transfer level (RTL) abstraction. This is one of the basic reasons why this process gained a great increase in the productivity. Nowadays, an important effort is being spent in order to develop system level languages (SLL) and to define new design and verification methodologies at this level of abstraction. RTL hardware design is too low as an abstraction level to start designing multimillion-gate systems.

State-of-the-art SLL proposals can be classified into three main classes. First, reusing classical hardware languages such as extending Verilog to SystemVerilog [6]. Second, readapting software languages and methodologies (C/C++ [10], Java [2], UML [5], etc.). Third, creating new languages specified for system level design (Rosetta [1] for example).

SystemC [12] is among a group of design SLLs proposed to raise the abstraction level for embedded system design and verification. It is expected to make a stronger effect in the area of architecture, the co-design and integration of hardware and software [11].

The verification of a SystemC Design is a more serious bottleneck in the design cycle. Going further in complexity and considering hardware/software systems will be out of the range of the nowadays used simulation based techniques [7]. Classical verification techniques when used with SystemC will face several problems related to the object-oriented aspect of this library and to the complexity of its simulation environment.

For instance, the main trends in defining new verification methodologies are considering a hybrid combination of formal, semi-formal and simulation techniques. This kind of hybrid techniques can offer a partial answer to the question: "Is the verification task complete?" However, an answer to a question like "Is a property always true?" can be only answered by purely formal techniques such us theorem proving [8] and model checking [9]. This latter, despite its problem of state explosion, is gaining a lot of interest in both areas academic and industrial. A number of proposals offer to abstract the system in order to verify some of its properties using model checkers and then complete the verification process by classical simulation techniques.

In this paper, we present an approach to verify a bus structure using model checking. The bus represents a generic Master/Slave architecture included as part of the SystemC library. It supports a variety of modes: blocking, direct, non-blocking, fast memory and slow memory. Our objective is to abstract the bus's model in order to verify some of its critical properties (mainly: liveness and safety properties).

The rest of this paper is organized as follows: Section 2 presents an approach to verify SystemC designs. Section 3, describes the bus structure. Section 4 presents the verification steps of the bus structure and the experimental results. Section 5, finally, concludes the paper.

II. Verification Approach

We use the verification approach given in Figure 1, where the static code analyzer gets at its input a SystemC design and a set of reduction tactics (called abstraction library). It then generates a reduced hypergraph representation of the design. This latter is fed into a Hypergraph to Verilog converter. The conversion is seen as a concretization of the abstracted design (hypergraph) into the Verilog language. We did select Verilog because we will use the FormalCheck model checking tool [3].

A. SystemC Library

SystemC is a set of C++ class definitions and a methodology for using these classes [11]. The core language consists of an event-driven simulator as the base. It works with events and processes. The other core language elements consist of

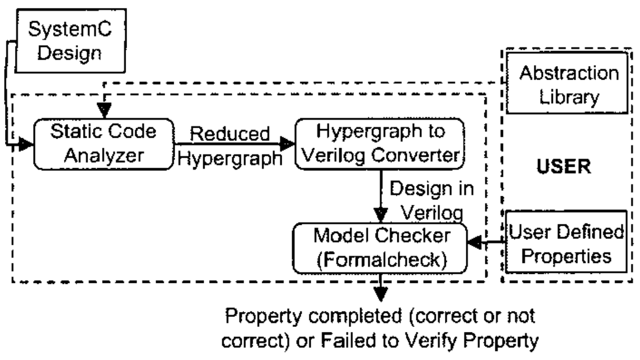
SystemC Design
Static Code Analyzer
Reduced Hypergraph
Hypergraph to Verilog Converter
Design in Verilog
Model Checker (Formalcheck)
Abstraction Library
USER
User Defined Properties
Property completed (correct or not correct) or Failed to Verify Property

Fig. 1. Cascading Model Checking with Abstract Interpretation.

modules and ports for representing structures. Interfaces and channels are used to describe communications. The primitive channels are built-in channels that have wide use such as signals, semaphores and FIFOs. SystemC provides data types for hardware modelling and certain types of software programming as well.

B. SystemC Abstraction

As a solution to the SystemC verification problem we use an abstract environment that can be used for: (1) the analysis and verification of SystemC programs, (2) abstract debugging and (3) possible interfacing with model checking and simulation. The analysis of the design is, as defined in [4], based on approximate semantics of programs to provide sound answers to questions about their run-time behaviors. The abstract debugging will be possible thanks to the abstraction of the memory (allocation blocks and the stack), the language simulation manager, component responsible for running the simulation, the events' stack and to the code of the program itself. The program execution environment as well as the simulation environment will be represented in order to allow abstract execution of the program.

In order to interface abstract interpretation with model checking (i.e. feed the abstracted code into a model checker) objects' and events' aspects of SystemC designs need to be translated into a procedural like code. Eventually this may seem to be not always feasible since we are starting from an object-oriented program structure. However, the approach can still be valid when restricted to some parts of the program to verify local properties.

In summary, the requirements for our abstract environment are:

- Construct an abstract environment for C++ as an object-oriented program components. This will include the code (instructions, expressions, operations, etc.) and memory (allocation blocks and stack) abstractions.
- Define a specific abstraction for the SystemC simulator (events manger and events stack) and for all the SystemC's language specific classes (modules, signals, channels, etc.)
- Consider some program analysis tactics to extract properties from the abstracted program or to concretize it into a code that can be fed into a model checker.
- Represent the abstract environment in a graphical structure in order to allow more efficient abstract debugging and possible test coverage hints.

In order to analyze statically SystemC designs, we considered an approach based on Abstract Interpretation [4] which is a formal technique that has proven to be efficient with object-oriented languages and large programs. The approach consists of:

- Construct collecting semantics: which defines statically the future domains that will serve for the analysis and their specific manipulations.
- Construct Abstract Semantics: which maps a property to a finite representation of the property more suitable for the analysis.
- Define Analysis Techniques: which analyzes the abstract representation of the system in order to extract properties and/or to reduce the program size.

At the end of the analysis the program will be represented in a graphical format called *hypergraph* [13]. This latter, can be seen as a general automata connecting its states by branches (also called hyper-branches). Theses branches can be seen as an extension to Binary Decision Diagrams (BDDs) more adapted to programs representation. In other terms, they offer a higher level of abstraction and flexibility by introducing the notion of confined hypergraph. This encapsulation property of the hypergraphs is very suitable to SoC where a system is a connection of modules using its input and output ports.

C. Applying Model Checking

Model checking [9] is one of the main formal verification techniques used in the EDA industry. It is concerned with properties verification mainly at the RTL.

Model checkers are the most adequate formal technique to be used at the system level design. With this technique there are no corner cases, because the model checker examines 100% of the state space without having to simulate anything. However, we note that model checking is typically used for small portions of the design only, because the state space increases exponentially with complex properties and on quickly runs into a "state space explosion".

For instance, there is no new model checkers adapted for system level design. Nevertheless, what is interesting about these techniques is the definition of hierarchical verification allowing the use of the checkers for small design portions and guiding the abstraction in order to verify some particular properties.

III. Bus Structure Model

This bus structure as described in Figure 2 uses an overall form of synchronization where modules attached to the bus execute on the rising clock edge, and the bus itself executes on a falling clock edge. Multiple masters can be connected to the bus. Each master is identified by a unique priority, that is represented by an unsigned integer number. The lower this priority number is, the more important the master is. Each master communicates with the bus via an interface which describes the communication between masters and the bus; three modes are possible:

- Blocking Mode: Data is moved through the bus in burst-mode. The transaction cannot be interrupted by a request with a higher priority.
- Non-Blocking Mode: Read or write a single data word. After the transaction is completed, the caller must take care of checking the status of the last request. The status of the request is one of: SIMPLE_BUS_REQUEST (request issued and placed on the queue), SIMPLE_BUS_WAIT (request being served but is not completed), SIMPLE_BUS_OK (request completed without errors) or SIMPLE_BUS_ERROR (an error occurred during processing of the request).
- Direct Mode: The direct interface functions perform the data transfer through the bus, but without using the bus protocol. They are usually used to debug the state of the memory.

The slave interface describes the communication between the bus and the slaves. Multiple slaves can be connected to the bus. Each slave models some kind of memory that can be accessed through the slave interface. Two modes are possible:

- Direct interface: immediate read or writing of data without using the bus protocol.
- Indirect interface: read or write a single data element, pointed to by data in or from the slave's memory. The functions return instantaneously and the caller must check the status of the transfer.

To the bus more than one master can be connected. Each master is independent of the others, so each master can issue a bus request at any time. The arbiter selects the most appropriate request according the following rules:

- If the current request is a locked burst request, then it is always selected.
- If the last request had its lock flag set and is again 'requested', it is selected from the collection queue and returned, otherwise:
- The request with the highest priority is selected from the collection queue and returned.

IV. Formal Verification

In this section we will illustrate the verification approach on a bus structure offered as part of the SystemC distribution [12]. In fact, this structure includes several SystemC components and showed the principles of using SystemC at the transactional level. Besides some of the sample properties, e.g. liveness and safety, cannot be verified using simulation. They require the usage of formal techniques such as model checking.

A. Abstraction

A partial representation of the bus's hypergraph is given in Figure 3. It shows the first hypergraph generated from the bus code. It includes an events' environment containing several processes: masters, slaves, clocks, arbiter, etc. In parallel with the program environment, the events environment includes the list of all the system processes and their status. For simplification, we use only two status for each process: active (1) and not-active (0).

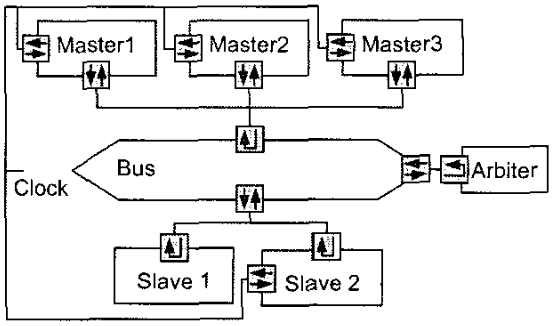

Fig. 2. Simple Bus Structure.

The simulation manager is presented as a box connected to the entries of the program hypergraph. It can be seen as a procedure that determines the structure of the system according to the list of active processes. For example, if the Master 1 is sending active, then, only its correspondent code is analyzed. Each small box from the program environment, (e.g., arbiter()) presents a confined hypergraph that includes the correspondent object members and methods.

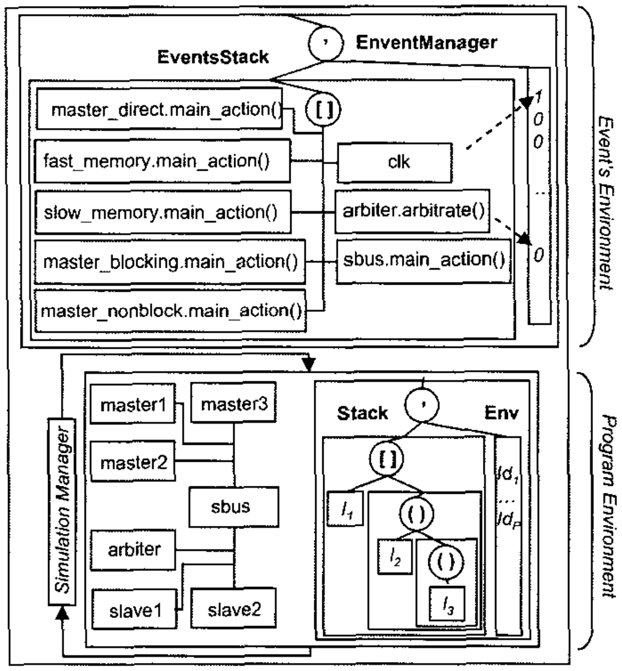

Fig. 3. Hypergrah of the Simple Bus Structure.

TABLE I
MODEL CHECKING RESULTS.

Property	CPU Time	Memory (in MB)
P1	6:59:12	93.59
P2	15:23:02	183.91
P3	17:46:54	293.63

TABLE II
VERIFICATION PLATFORM.

FormalCheck version	3.2
Main Memory	4.0 GB
CPU	2 CPUs (Run 900MHz)
Architecture	Sparc
OS Version	5.8

B. Model Checking

After applying reductions tactics on the hypergraph of Figure 2, the generated reduced hypergraph is concretized into a Verilog code. This latter is fed into the FormalCheck tool [3] in order to verify some of the design's properties. In fact, FormalCheck verifies that a design model exhibits specific behaviors (*properties*) that are required by the design specification. Properties that form the basis of a model checker's query fall into two categories: *safety* and *liveness*. Safety properties can be expressed using one of two formats: The *always* format and the *never* format. Liveness properties describe behaviors that are *eventually* exhibited.

For instance we considered the following properties:

```
Property 1:
  NEVER( (simple_bus.request==ture)
      &&(simple_bus.status!=BUS_OK))

Property 2:
  AFTER (simple_bus.request==true)
  && (simple_bus.request.block==true)
  EVENTUALLY (simple_bus.status==BUS_BLOCK)

Property 3:
    EVENTUALLY (simple_bus.status==BUS_OK)
```

Property 1 means that a master generates request only when the bus is ready to handle new requests (i.e. bus status set to BUS_OK). Property 2 says that if the bus receives a new blocking request, then, in the future, its status will change to blocking (i.e. bus status set to BUS_BLOCK). Property 3 proves the bus status will always return to ready to receive new requests.

C. Experimental Results

We first started by translating the bus code from SystemC to verilog (without abstraction). We modelled the SystemC simulator as a new module. Although this simplification reduces effectively the complexity of the code, the verification of all the previous properties failed after few minutes with the same problem of "memory exceeded". Then, when using the abstracted code all the properties were verified as it can be seen in Table I. The verification platform is described in II.

V. CONCLUSION

In this paper, we used an approach based on abstract interpretation to verify, using model checking, a bus structure modelled in SystemC. We used abstract interpretation to reduce the code. Our choice was guided by the well known performances of this technique when dealing with a variety of languages and complex systems. The reduced code is represented in a graphical structure, hypergraph, in order to allow more flexible yet efficient analysis environment.

We translated the reduced hypergraph structure to Verilog in order to use model checking in FormalCheck to verify some of the system's properties. The results obtained on a bus structure showed the feasibility of the approach. In future work, we will investigate (1) the proof of soundness of the concretization of the hypergraph into a Verilog code and (2) the application of model checking directly on the hypergraph structure.

REFERENCES

[1] P. Alexander, R. Kamath and D. Barton. System Specification in Rosetta. In Proc. *IEEE Engineering of Computer Based Systems Symposium*, UK. April 2000.

[2] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. *IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Computer Society Pres, Los Alamitos, CA, pp. 175-184, 1998.

[3] Cadence. Formal Verification Using Affirma FormalCheck. version 2.4, Auguest 1999.

[4] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511-547, August 1992.

[5] D B. P. Douglass. Real-Time UML. Addison-Wesley, 2000.

[6] IEEE Std. 1364-2001, IEEE Standard for Verilog Hardware Description Language 2001. ISBN 0- 7381-2827-9. IEEE Product No. SH94921-TBR.

[7] M. Kantrowitz and L. Noack. I'm Done Simulating: Now What? Verification Coverage Analysis and Correctness Checking of the DECchip21164 Alpha Microprocessor. In Proc. *ACM/IEEE Design Automation Conference*, 1996.

[8] C. Kern and M. Greenstreet. Formal Verification in Hardware Design: A Survey, *ACM Transactions on Design Automation of E. Systems*, Vol. 4, April

[9] P. C. Pixley. Integrating model checking into the semiconductor design flow, *Computer Design's Electronic Systems journal*, pp. 67-74, March 1999.

[10] R. Roth and D. Ramanathan. A High-Level Hardware Design Methodology Using C++, In Proc. *4th High Level Design Validation and Test Workshop*, San Diego, pp 73-80, 1999.

[11] Systemc 2.0.1 language reference manual. open systemc initiative, 2003.

[12] Systemc website: http://www.systemc.org, 2004.

[13] F. Vedrine. Analyses totales de programmes par interpretation abstraite. PhD thesis, Ecole Polytechnique, Paris, France, 2000.