Mixing ATPG and Property Checking for Testing HW/SW Interfaces [*]

Alessandro Fin
University of Verona
Strada le Grazie, 2
37134 Verona, Italy
fin@sci.univr.it

Franco Fummi
University of Verona
Strada le Grazie, 2
37134 Verona, Italy
fummi@sci.univr.it

Graziano Pravadelli
University of Verona
Strada le Grazie, 2
37134 Verona, Italy
pravadelli@sci.univr.it

ABSTRACT

A critical part of the design of HW/SW systems concerns the definition of the HW/SW interface. Such interfaces do not directly map a functionality of the system description, but they are inferred by the characteristics of the selected programmable device (CPUs, DSPs, ASIPs, etc.). Their addition to the design can modify the behavior of the original system, thus their verification is a hard task. The proposed verification methodology joins functional verification and property checking in order to avoid their respective limitations. The methodology is focused on SystemC descriptions that can be automatically synthesized. This is particularly important since commercial model checking tools work on structural hardware descriptions, which can be obtained by performing rapid prototyping of both HW and SW parts of SystemC models. The proposed approach has been verified on the SystemC model that is the reference synthesis example of one of the most powerful SystemC synthesis environment.

Categories and Subject Descriptors

H.8.1 [**Harware**]: Testing

General Terms

Design, Verification

Keywords

ATPG, Fault simulation, Model Cecking

1. INTRODUCTION

Basic tasks of designing HW/SW systems concern their modeling and verification. By adopting the SystemC philosophy, the modeling activity can be performed by using a unique description language (C++), which can represent both software and hardware modules [1].

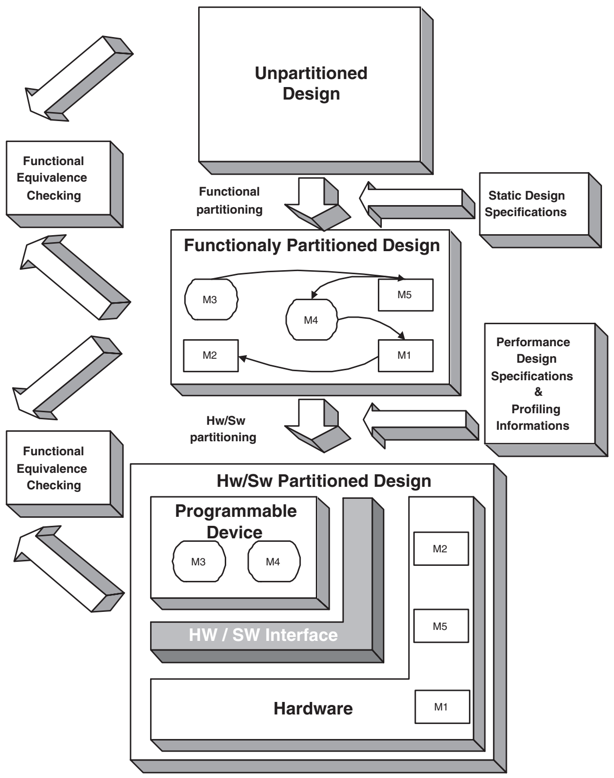
Unpartitioned Design
Functional Equivalence Checking
Functional partitioning
Static Design Specifications
Functionaly Partitioned Design
M3
M5
M4
M2
M1
Performance Design Specifications & Profiling Informations
Hw/Sw partitioning
Functional Equivalence Checking
Hw/Sw Partitioned Design
Programmable Device
M3
M4
M2
M5
HW / SW Interface
Hardware
M1

Figure 1: Design flow with HW/SW interface definition.

Under this assumption, the design flow becomes a continuous refinement of an initial system description (see Figure 1). Such a description is partitioned, at first, into components, which are then characterized as HW or SW parts by adopting, for instance, performance estimation metrics [2]. This characterization (last step in Figure 1) introduces in the design new parts so called as interfaces [3]. Verification of interfaces is a particularly hard task [4], since they do not directly map a functionality of the system modeled by the initial description.

This paper is focused on the definition of a verification methodology able to validate the insertion of interfaces into a system description partitioned between HW and SW components. The aim of the proposed verification methodol-

[*]Research activity partially supported by the European Community IST-2001-34607 project: SYMBAD

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'03, April 28–29, 2003, Washington, DC, USA.
Copyright 2003 ACM 1-58113-677-3/03/0006 ...$5.00.

ogy is to combine functional verification based on simulation with model checking. This joined approach is performed in order to improve the fault coverage achieved by using a functional ATPG on the HW/SW interface of a digital device. Undetected faults are analyzed in depth by using model checking in order to characterize their functional interpretation either as hard to verify configurations or as dangerous design errors. The methodology is applied to synthesizable SystemC descriptions, thus a rapid hardware synthesis of the software part allows to apply standard model checking tools (e.g., *SMV* [5] and *RuleBase* [6]) working on structural hardware descriptions only. This allows to characterize undetected faults of the HW/SW interface.

2. METHODOLODY

The low cost and the flexibility of programmable devices as CPUs and DSPs allow the definition of modern digital designs as a mix of hardware and software modules. The critical functionalities are developed as *ad-hoc* hardware components, whereas the general functionalities run on a programmable device. This approach decreases the cost and the development time of the new design, since it uses fully tested programmable devices, from chip vendors, and developers have to focus their effort only on the required *ad-hoc* hardware components. Adopting a system level design language, e.g. SystemC, for the initial design description, the development of the software modules require a minor effort.

2.1 Hw/Sw partitioning

System level design languages (e.g. SystemC) and software languages (e.g. C++, C) can be adopted to describe the whole system at early stage of development (see Figure 1). Design specifications and profiling sessions allow to identify the critical sections of the design. Designers have to refine these results in order to choose which functionalities will become a hardware module. The general adopted methodology is to develop as hardware modules the most time consuming functionalities [2].

2.2 Test set inheritance

Every time a design description, at an identified abstraction level, is converted to the description at the lower level, there is the need of a validation phase. This test is performed by applying the whole set of test patterns, identified at the higher abstraction level, on the new description at the lower level (see the leftmost part of Figure 1). Moreover, this set of test patterns is extended at each development phase with some new level-specific test vectors that are simulated on the previous-level representation to identify correct responses.

2.3 Interface extraction

The HW/SW partitioning step requires the definition of an interface between hardware and software functionalities. The goal of the defined interface is to rebuild the data and control dependencies between hardware and software functionalities. Several alternatives can be considered with respect to this functional constraint:

- parallel interface with polling;
- serial interface with polling;
- parallel interface with interrupt;
- serial interface with interrupt.

2.4 Interface verification

The interface is a new hardware component of the developed architecture. It could not be completely sketeched in the initial design specification, because it depends on the adopted implementation details. Thus, it has to be accurately tested and verified. Test sets, previously generated for the design before the HW/SW partitioning, represent an excellent initial test set to check the new design definition. These test sequences can test the main feature of the interface, but the full interface verification can require a specific test generation phase. The proposed methodology for HW/SW interface verification consists of the following three steps as reported in Figure 2:

- ATPG on the unpartitioned design;
- fault Simulation and ATPG on the HW/SW partitioned design;
- interface fault coverage improvement by using model checking.

This verification methodology focuses the use of the powerful, but computational expensive, model checking analysis, to the only hard to detect faults, which can really represent design errors. Moreover, this is a general methodology, which can be efficiently applied to test all kind of interfaces (HW/SW, HW/HW, data format transfer, etc.)

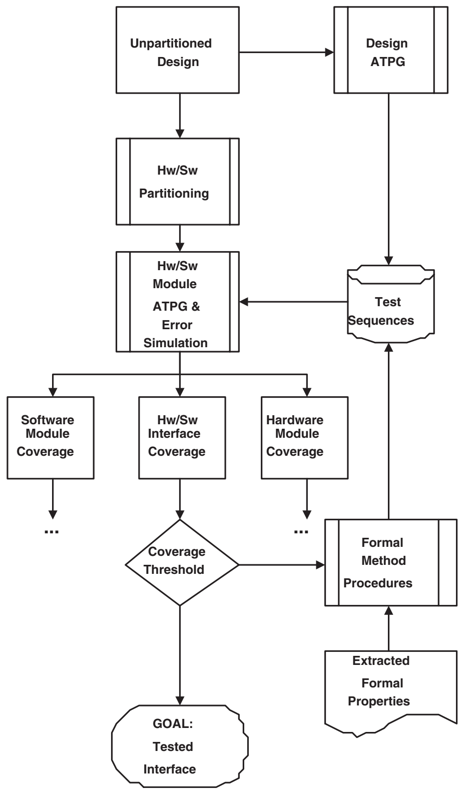

Figure 2: HW/SW interface verification methodology.

3. FUNCTIONAL TPG

Every time a design description, at an identified abstraction level, is converted to the description at a lower level, a validation phase is mandatory. This task is usually performed by using a simulation-based approach whenever formal verification techniques cannot be applied. This test is performed by applying the whole set of test patterns, identified at the higher abstraction level, on the new description at the lower level. Moreover, this set of test patterns is extended at each development phase with some new level-specific test vectors that are simulated on the previous-level representation to identify correct responses. By using a common description language covering all phases of a design hierarchy, this testing strategy could be repeated from the system level to the structural level. SystemC could be this common description language.

The proposed testing strategy is based on the *bit-coverage* fault model [7], which shows a high correlation between faults modeled on descriptions of different abstraction levels. This fault model has been used to correlate test patterns between the behavioral, RT and gate-level for hardware components only. In this paper, a similar approach is applied to embedded systems, thus testing both hardware and software parts. Each bit of each variable, condition and input/output port is set stuck-at zero or one to obtain erroneous SystemC descriptions that are compared to the fault-free description in order to identify functional test patterns.

The current implementation of the TPG is based on a genetic engine. The test generation algorithm evolves a population of test sequences. The fitness of the generated sequences is proportional to the sequence fault coverage. Two operators have been developed for intra-sequence and inter-sequence crossover. A classic mutation operator can flip each bit of the generated test sequences.

Test sequences are applied to both fault-free design and faulty design. The output checker compares the output coming from two designs. A fault is detected every time the former comparison on primary output fails. Each detected fault is dropped from the fault list and any new generated test sequence is applied on all the undetected faults. Each bit-fault in the design is identified by a unique fault code. An extra input port is added to the original design module to control the simulation of the target fault.

The used fault injection method differs from the original version presented in [8], since it is performed directly on SystemC code instead of on the in-memory intermediate representation of the design. This fault injection schema produces only one erroneous SystemC design, which simulates all the erroneous behaviors by driving the error port.

Finally, three halt conditions can be set for the GA test generation engine:

- fault coverage threshold attainment (the default value is 100%);
- maximum generation time threshold;
- maximum number of generated sequences.

These conditions, in conjunction with the fitness function, allow to control the evolution of the GA TPG.

4. PROPERTY CHECKING

The application of the ATPG phase of the proposed methodology generally produces quite high fault coverage, however, some faults remain undetected, particularly on the HW/SW interface due to the verification problems described in the introduction. Are these faults hards to detect or redundants? In this work, we propose to use model checking in order to investigate the nature of these undetected faults.

Two kinds of formal properties can be written in order to resolve the ambiguity about the nature of undetected faults:

Specific design dependent properties. They are generated with respect to the specification and the expected functionality of the system to prove the correctness of the design under test. This kind of properties is suitable to prove the overall correctness of an implementation with respect to the corresponding specification. Indeed, we want just to focus the attention on some particular faults, those not detected by the ATPG.

General output comparison properties (OCP). They depend only on the primary outputs of the system and not on its functionality, thus no knowledge of the system behavior is required and automatic generation becomes feasible. The main idea is the same used for fault simulation. We assume the presence of a stuck-at fault in the faulty description of the system and we compare its outputs with the ones obtained by the fault-free system. The model checker should refutes a simple property as one represented in Figure 3 in order to generate a test pattern identifying the analyzed fault. Figure 3 represents the main file written to prove the `OCP` property by using the *Cadence SMV* Model Checker [5] on the `VD_iq` design (see Section 5). This kind of properties could be very hard to be proved, because the model checker must manage a finite state machine (FSM) with a very high number of states to complete the verification of general output comparison properties.

```
#include "VD_iq.smv"

MODULE main() {
-- port declarations
-- fault-free system instantiation
-- faulty system instantiation
-- error_port activates fault 2938
error_port := 2938;
-- formula to be proved
OCP: assert G((po_1=po_1_e)&...&(po_N=po_N_e));
}
```

Figure 3: Example of `OCP` property. `po_i` are primary outputs of the fault-free system, `po_i_e` are primary outputs of the faulty system

In despite of computational complexity problems, general output comparison properties are more suitable for our purpose, than specific design dependent properties since the possibility of an easy automatic generation make them more attractive. In order to overcame complexity problem, a good choice is represented by **Bounded Model Checking**. BMC is a technique that uses a SAT solver to search for counterexamples of a bounded length. Finding counterexamples in this way is sometimes significantly faster than using symbolic model checking. We used the *zchaff* [9] SAT solver joined with the *SMV* model checker since it allows to solve problems with more than one million variables and ten million clauses.

5. EXPERIMENTAL RESULTS

The proposed methodology has been applied to an inverse quantizer of the MPEG decoder described in [10]. This system is the synthesis reference example of SystemC Compiler. Figure 4 shows the example architecture, which is composed of three main modules.

The profiling analysis indicates the `inverse quantization` submodule as the most critical part of the whole design. In fact, from a performance point of view, it requires the ma-

Design configuration	SystemC code line	VHDL code line	PI#	PO#	Proc.#
Unpartitioned	557	17565	67	22	2
Partitioned	884	27534	67	22	4

Table 1: Characteristics of the example.

Design Configuration	Fault#	ATPG F.C.%	ATPG Time	Fault Sim. F.C.%	Total F.C.%	Seq.#	Vect.#
Unpartitioned	3262	59.1	5405 s.	-	59.1	47	30936
Partitioned	3364	60.8	7329 s.	54.2	62.1	46	32110

Table 2: ATPG and fault simulation results.

Simulation		Property Checking					Total		
Faults	Det.	Faults	Det.	Untestable Faults	Def. Properties	Time	Det.	F.C.%	Efficiency%
82	79	3	1	2	18	1132 s.	80	97.5	100

Table 3: Joint verification technique for the HW/SW interface.

jority of the resources. Thus, it is the best candidate to become a hardware component, whereas the `saturation` and the `mismatch control` modules will become two software routines to be executed on the selected programmable device. Figure 5 shows the post synthesis architecture. Table 1 shows the features of both partitioned and unpartitioned architectures.

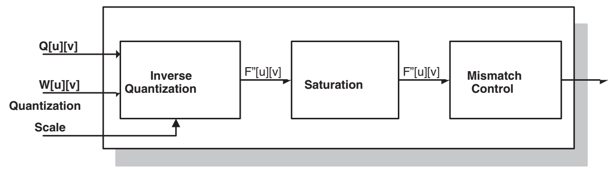

Figure 4: System architecture.

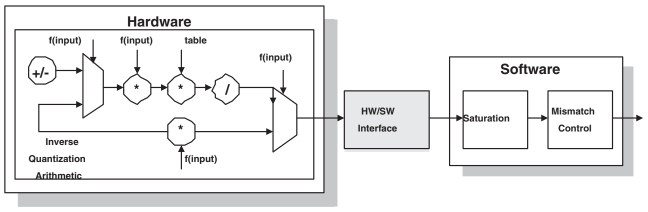

Figure 5: Post-synthesis architecture.

A parallel interface has been designed to allow the comunication between HW and SW blocks. This interface model introduces the lowest delay and it requires a very simple control logic to enable the data transfer from the HW blocks to the SW ones.
The HW/SW partition increases both the number of SystemC code lines and the number of processes. ATPG and fault simulation results are presented in Table 2. The total fault coverage for both architectures highlights the complexity of the considered example. The partitioned design contains more faults than the unpartitioned architecture. It includes the fault on the defined interface plus the faults on the the added signals to connect the three functional blocks.

The joint methodology results for the interface test are summarized in Table 3. The ATPG and the fault simulation phases have not been able to detect 3 interface faults. The property checking phase has been accomplished defining 18 properties. In order to manage the computational resources required by SMV to elaborate the example, a property for every output port has been written rather then a single general rule. The defined properties generated test sequences for 1 of the ATPG undetected faults. No counter example has been found by SMV for the other two. One is related to a constant value, while the last one identifies a design error due to an unconnected signal. Note that, the SMV process requires a larger amount of time with the respect to the TPG process, thus the simple analysis with SMV of an interface fault would be not feasible. This emphasizes the efficiency of the proposed joint methodology.

6. ADDITIONAL AUTHORS

U.Rossi (ST Microelectronics, Via C. Olivetti 2, 20041 Agrate Brianza Italy, email: `umberto.rossi@st.com` and F.Toto (ST Microelectronics, Via C. Olivetti 2, 20041 Agrate Brianza Italy, email: `franco.toto@st.com`).

7. REFERENCES

[1] SystemC User's Guide. *Synopsys, CoWare, Frontier Design, version 2.0*, 2001.

[2] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli and K. Suzuki. Hardware-Software Co-Design of Embedded Systems, The POLIS Approach. *Kluwer Academic Publishers*, April, 1997.

[3] J. Rowson and A. Sangiovanni-Vincentelli. Interface-Based Design. *ACM/IEEE Design Automation Conference (DAC)*, pp. 178–183 97.

[4] D. Panigrahi, C. Taylor, and S. Dey. Interface based Hardware/Software Validation of a System-on-Chip. *IEEE High Level Design Validation and Test Workshop (HLDVT)*, pp. 53–58 2000.

[5] K.L. McMillan. Symbolic Model Checking. *Kluwer Academic Publishers*, 1993.

[6] I. Beer, S. Ben-David, C. Eisner and A. Landver. RuleBase, an Industry-Oriented Formal Verification Tool. *ACM/IEEE Design Automation Conference (DAC)*, pp.655–660 1996.

[7] F. Ferrandi, F. Fummi, L.Gerli, and D. Sciuto. Symbolic Functional Vector Generation for VHDL Specifications. *Proc. IEEE Design Automation and Test in Europe Conference (DATE)*, pp. 442–446, 1999.

[8] A. Fin, F. Fummi, and G. Pravadelli. AMLETO: A Multi-Language Environment for Functional Test Generation. *Proc. IEEE International Test Conference (ITC)*, pp. 821–829, 2001.

[9] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: Engineering an Efficient SAT Solver. *ACM/IEEE Design Automation Conference (DAC)*, pp. 530–535 2001.

[10] CoCentric SystemC Compiler Behavioral Modeling Guide. *Synopsys version 2000.11-SCC1*, 2001.