Positioning Test-Benches and Test-Programs in Interaction-Oriented System-on-Chip Verification

Xiaoxi Xu and Cheng-Chew Lim and Michael Liebelt
The University of Adelaide
{justinxu;cclim;mike}@eleceng.adelaide.edu.au

(Topic Category: Simulation-Based Validation; Hardware/Software Co-Validation)

***Abstract*— In simulation-based system-on-chip (SoC) verification, in addition to the test-bench (TB) - the basic facility for stimulation and observation, software native to the SoC also plays a part in interacting with the SoC. This software is referred to as the test-program (TP). However, the relationship between the TB, the TP and the SoC is not always intuitive and can cause conceptual confusion. This paper discusses this confusion and shows how to address it by positioning the TB and the TP more naturally in the verification framework.**

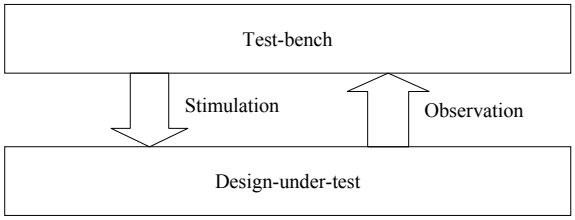
I. Introduction

While the system-on-chip design paradigm is gaining increasing popularity, the well-known verification bottleneck is becoming even more serious. At system-level, the *concurrency* among hardware (HW) components becomes a new verification dimension. *Resource-competition* is an inevitable consequence of concurrency. HW components could show functional problems when competing with each other for resources, even if they have already passed component-level verification. The nature of system level bugs is different from that of bugs local to a component. It is sometimes not possible to attribute such a bug to a *particular* HW component; instead, the bug may be caused by the ill-matched behaviours of multiple components[1]. These ill-matched behaviours only show up in tight resource-competitions.

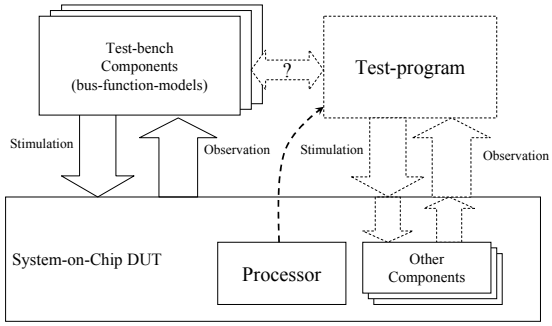
These bugs usually arise from *implementation details* of multiple components. Therefore formal methods, which work best on abstract models and at the component level, are not in a position to find those bugs. The industry is depending less on formal methods; system level verification substantially relies on a conventional simulation approach [2].

The simulation-based verification practice centers around test-bench (TB) development and automation. Figure 1(a) is the conventional model of verification. There is a TB and there is a design-under-test (DUT); the TB *stimulates* the DUT and *observes* the response from the DUT. While this TB-DUT "dualism" appears straightforward and proves fruitful in component-level verification, complications arise when verification needs to be performed at system level. The dualism raises two major problems.

1. The distinction between external and internal behaviours. The TB tends to treat the DUT as a black box. It applies stimulation/observation from the *exterior* of a DUT, so it is inherently problematic to force the TB to control/observe the DUT's internal behaviours. There are *white-box* approaches,



(a) TB stimulates and observes DUT.



(b) Both TB and TP could stimulate/observe DUT.

Fig. 1. The impact of introducing SW (TP) as a new entity.

i.e., adding control/observation points around components inside a DUT, to supplement the black-box approach. However, we could argue that this approach is still black-box natured in the sense that the similar controllability/observability issues still exist at component level.

2. The test generation problem. A TB is essentially the *vehicle* to transform abstract test-cases (TCs) into physical stimuli to a DUT. It is usually assumed that abstract TCs are already prepared. The TB does not really handle TC generation. The best stimuli-generation mechanism provided by state-of-the-art hardware verification languages (HVLs) is merely "constrained randomization", which is largely independent of the central characteristics of a system, namely, concurrency and the resource-competition. It is claimed that HVLs are simply the *tools*, but not really the *methodologies* [3]. It is insufficient to rely on TB alone, even if it is written in HVL, to battle the soaring verification complexity, which is growing at a "double exponential rate" [2].

For an SoC DUT, it is a common practice for a verification

engineer to write TCs in the form of software (SW). These SW tests are native to the DUT, not native to the simulation environment as the TB is. These TCs are typically program-snippets in which configurations are written to control registers, and the status is read back from the status-registers. This technique demonstrates SW's capabilities to control and observe a DUT. Although writing TCs in SW is often treated as an *ad-hoc* verification technique, we should realize that the introduction of SW as a third entity has *overturned* the traditional TB-DUT dualism. In the sequel, we will refer SW also as "test-program" (TP) to contrast with the term "test-bench" (TB).

As suggested in Figure 1(b), the relationship between the TB and the TP becomes problematic, since both of them can stimulate and observe a DUT. The TP, as the third entity, causes complications in addition to the dilemmas already caused by the TB-DUT dualism. Nevertheless, the TP also brings the potential to address those dilemmas and complications together. To achieve that, the TB and the TP should be work together in the most effective possible manner in the verification framework. This paper proposes to let the TP, instead of the TB, take the more *active* role of test-case *control*, especially parallelism management. The TB should take the relatively *passive observation roles*.

The rest of the paper is organized as follows. Section II discusses related works and background information on system level verification, especially on the problematic TB-TP relationship. Section III describes TP's various roles in system level verification; a TP-TB interface is also introduced to reshape the overall relationship among TB, TP and DUT. Section IV discusses what TB could do in TP-centric methodology, with our experience in using TB to profile TP. Section V presents our final solution to the TP-TB-DUT relation problem and concludes the paper.

II. Related Works and Background

A. Problematic TB-TP relation

The reality in verification community is that verification remains the "biggest single bottleneck in IC design" [4], taking 50-70% of the total time and effort of one design cycle. Much of the effort/time is spent in TB construction.

Simulation-based verification has been largely understood as constructing test-benches (TBs) and using them as the vehicles to apply stimuli to designs-under-test (DUTs) and observing DUTs' response. As the underlying DUT becomes more sophisticated, so is the TB. For instance, in order to meet the requirement of SoC verification, Synopsys has put forward a 4-layered TB structure [5], in which each layer is responsible for transforming more abstract tests into more concrete tests. A similar concept is also present in [6].

Hardware-description-languages (HDLs) are not suitable to compose such TBs. Building complex TB requires costly propriety "hardware verification languages" (HVLs), which provide convenient mechanisms to fill abstract test-cases with concrete parameters. Although layered TB in HVLs increases the abstraction level of test-cases, it has not resolved the basic problem of generating abstract test-cases. We further argue that the construction of sophisticated TBs has actually *defocused* our attention on the DUT itself. Verification engineers are easily trapped in the painstaking process of *TB development*, and distracted away from the more creative task of *test-case generation*. While each HVL could provide its unique strength [7], we should always bear in mind that the term "verification" does not refer to language-oriented technologies, but a rather generic concept [3].

If a DUT is an SoC, verification engineers could manually develop test-cases, not in HDLs or HVLs, but in the programming languages (mostly assembly or C) *native to the DUT* rather than native to the simulation environment. Using software-based self-tests is not a new idea [8], [9], [10], [11], [12], [13]. But the differences between SW-based tests and TB-based tests are seldom discussed explicitly. SW, as well as a TB, could provide stimulation and observation, so what should be the proper relations among a TB, SW and the DUT? What are their roles in verification?

In [11], the proposed relation is that "TB controls/observes TP". Here, "TP" refers to diagnostic subroutines for HW verification purpose. The behaviours of TPs are monitored and *intercepted* by a TB. For instance, the TB could intercept a TP's read-access at a certain memory location, and modify the read data with a different value generated by the TB's randomization mechanism. Thus, the TP always gets random data from this memory location, which could be treated by TP as a random data source to configure HW components. However, using the TB to control a TP is un-natural in the sense that a programmer's *original intention* encoded in the TP is damaged. Therefore the applicability of using the TB to control a TP would be narrow. For the above specific example, the TP could obtain a similar effect (without the TB's intervention) by accessing not a fixed memory location but an array of pre-decided random values.

The more natural TP-TB relation should be exactly the opposite: TP controls TB [12], [14], [15]. In [15], the TP-TB synchronization is manually (and statically) specified but automatically implemented. In a test file, test-cases in the form of SW snippets are *annotated* with desired TB behaviour. A custom parser reads the test file and associates snippets' program addresses with their annotated TB behaviours. At simulation, the TB is sensitive to the program-counter (PC) in the processor. Whenever the PC matches any address identified by the parser, the associated TB behaviour is triggered. In this sense, the TB is under the control of the TP. This approach is useful but may not be sufficient to support tighter TP-TB communication such as parameter-passing.

It is better to let a TP control the TB more explicitly. The TB and the DUT are essentially HW; indeed, the TB can be composed in HDL just like the DUT. Since a TP can control/observe the DUT through the DUT's "programming interface", namely, its control/status registers, it makes sense to allow the TP to communicate with the TB in a similar fashion. In [16], a patented idea is to connect all TB components using a central bus dedicated to verification, just like a SoC

is integrated around some interconnection mechanism.

B. Dualism: TB and DUT

If the TB is controlled/observed by a TP as the DUT is, from the TP's point of view, the *dualism* between "TB" and "DUT" is reduced – there is only *one* type of entity, namely, SW-controllable HW components. Indeed, differentiating a HW component on the "TB" side or on the "DUT" side is not absolutely necessary. A good example is a processor (in a SoC design), which could be represented either by an accurate but slow HDL model, or by a less accurate but faster instruction-set-simulator (ISS). Counting the processor's behaviour on the TB side or on the DUT side is simply a matter of interpretation.

In the path-based test-generation method [17] used in IBM's XGEN tool [10], [18], a test is an "interaction" between a set of components, no matter whether the component from the SoC DUT side or the TB side. In our previous research [14], we implemented the same idea. Our research is demonstrated on a Nios SoC [19]. Figure 2 shows this SoC, coupled with its TB.

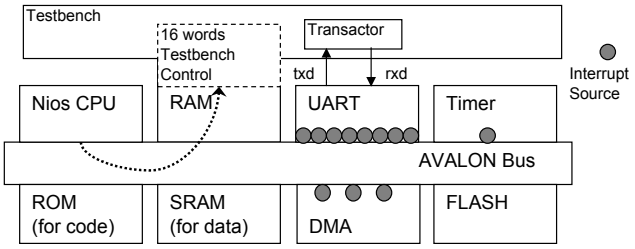


Fig. 2. System under demonstration – the Nios SoC.

If we stay with TB-DUT dualism, the bit-stream fed from the transactor in TB to the UART (universal asynchronous receiver and transceiver), an "external behaviour", is different from an "internal" behaviour, e.g., a "memory-to-memory-DMA-copy". In contrast, if we adopt the view that the TB and the SoC are unified as *one super-system*, and that this super-system is under the uniform control of TP, there would be no major difference for TP to control the above two behaviours. In fact, there is no distinction between"external" and "internal" behaviours. They become concepts subject to human interpretation.

C. Dualism: Test and Object-under-Test

Since the term "design-under-test" (DUT) is a *subjective* concept. We shall now *redefine* "object-under-test" (OUT).

At system integration stage, the focus should be put on the interaction between components, rather than the components themselves. This suggests that the *interactions, not components, should be treated as the object-under-tests*. This view is justifiable, since at integration stage a certain degree of confidence in HW quality should *already* be established. This assumption is especially valid for commercially available components.

Focusing on interactions (or communications) is already a common practice, such as transaction-based-verification (TBV) [20] and transaction-level-modelling (TLM) [21]. However, the verification community has not yet proceeded to migrate from the view in which interactions or communications are treated as the properties, or "capabilities", *attached* to HW components, to the view in which interactions themselves become a set of objects *independent* from HW components. We respectively refer to these two views as the "component-oriented" (CO) mindset and the "interaction-oriented" (IO) mindset.

In the CO mindset, there still exists a dualism: "tests" and "objects-under-test". Hence, *test generation* is an unavoidable task for a given OUT. While in the IO mindset, this dualism disappears – the terms "tests" and "objects-under-test" are now referring to *one* type of entities, namely, the *interactions*. The "test generation" issue now is reduced to OUT identification issue. The philosophical implication of IO mindset can be found in many areas of studies. Milnor observed that "it is reasonable to define the behaviors of a system to be nothing more or less than its entire capability of communication" [22]. The IO mindset is also coherent with the concept of *concurrency*, or *parallelism*, which is the central characteristic of a HW system.

The IO mindset is not merely an alternative way of thinking, but also yields practical solutions. In our recent work [23], a model called transfer-resource-graph (TRG) is formally defined. (We will briefly introduce it in the next section.) TRG can generate test-cases in the form of *event-driven* TP [14], [23], which naturally takes the role of *parallelism manager* – the very role that TB struggles to play (but hardly satisfactorily) in a TB-centric verification approach.

III. The TP's Roles in Interaction-oriented Verification

A. Overview

A TP-centric verification methodology is not necessarily to be interaction-oriented (IO). A TP-centric but component-oriented (CO) method implies that TPs are developed to diagnose HW components. Since test-case development for each HW component must be very custom, we see little opportunity for automation here.

If a TP-centric approach is combined with the IO mindset, the TP naturally takes the responsibility of managing the parallelism between *interaction-objects*. This reminds us that an *operating system* (OS) shares the same concept of *parallelism management*. However, a second thought is that an OS is *not* appropriate to be run on a system in its HW integration stage. Indeed, running OS with application SW on a system is necessary but would happen much later. And the owner of those SW is the SW development team, not the HW team; so running them is a "liability" rather than an "asset" to the HW team. Nevertheless, comparing TP with OS will shed some light on how to partition TP's responsibilities and where the opportunities for automation are. But first we briefly introduce our TRG model to understand what kind of interaction-objects we will be dealing with.

B. Transfer Resource Graph (TRG)

We use the term "transfer-types" to represent interactions at a specific abstraction level. A *transfer-type* is a set of *data*-intensive interaction patterns. A transfer-type also has three *control* elements: (i) configuration, (ii) invocation, and (iii) notification. They are instruction sequences that respectively (i) setup the parameters of the interaction pattern, (ii) trigger the interaction pattern, and (iii) and report the completion of the interaction. In addition, we discriminate between a "transfer-type" and its "transfer-instance". The latter is a specific configuration of the former. The term "transfer" is a shorthand for the latter; but it is often used to refer to both concepts when discrimination is insignificant. Figure 3 illustrate the concept of a transfer-type.

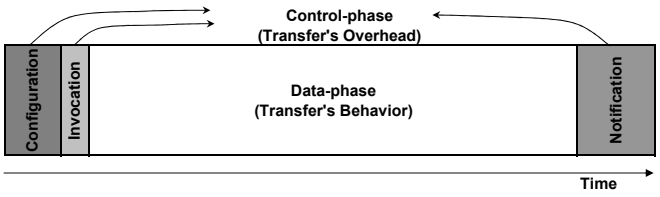


Fig. 3. Transfer life-cycle.

For example, a transfer-type named "UART-RX-by-Interrupt" in the Nios SoC (Figure 2) is a data-flow with the following properties:

- Configuration: *end-of-packet character*, *max-length*, *finish-mode* (by *max-length* and/or *end-of-packet-char*), *error-detection-mode* (`parity, frame`);
- Invocation: a STORE instruction to a special address – the TP-TB interface; when this address is written, the TB starts to feed the SoC with a bit stream.
- Notification: the UART interrupt-service-routine detects the finish conditions of the UART receiver.

The transfer model is generic enough to generalize HW behaviours (e.g., a memory-to-memory DMA copy) , SW behaviours (e.g., a `sort()` subroutine) and HW-SW collaboration (e.g., the above "UART-Rx-by-Interrupt" example) [13].

Transfers need HW resources to run. Parallelism and resource-competition are constructed in the form of *concurrent* transfers. Here we should make a subtle differentiation between *resource contention* and *resource conflict*. The former is our verification interest; it refers to resource competition at the *physical* level that is supposed to be resolved by hardware mechanisms, including bus contention arbitration, interrupt arbitration and cache-memory coherence schemes. In contrast, "resource-conflict" is the conflict in *logical* sense, requiring a programmer's discretion to *avoid*.

In order to construct contentions without introducing conflicts, we treat programmer-accessible bits (regardless control, status or data) as *bit-resources*. These bit-resources aggregate to form logical resources, i.e., "masters", "registers" and "memory-ranges". Then, TRG can be formally defined as a set of transfer-types, a set of logical resources, and the constraints between them. A *scenario*, is defined as set of transfers which do not use logical resources in a conflicting manner.

TRG is an interaction-oriented model. HW components' functionalities are abstracted away in a TRG. (The only property left is the very generic concept of "resource availability".) In fact, HW component's properties are reorganized to be *transfers*' properties. This is a central step to migrate from a CO mindset to an IO mindset. For instance, in order to instantiate a "UART-RX-by-DMA" transfer in the Nios SoC, with the property that "the RX stream terminates on the `eop` (end-of-packet) signal", TP needs to:

1) write the EOP register in the UART with a specific `eop` character;
2) setup the transactor in the TB to feed a byte-stream whose last byte is the same `eop` character;
3) enable the DMA engine to be sensitive on the `eop` signal.

We do not say that these three operations are "configuring three HW devices" – this is the CO interpretation; instead, we say that they are "configuring a *single* property of a transfer" – this is the IO interpretation.

C. SW's Responsibility in IO verification

SW plays multiple roles in interaction-oriented HW-verification.

- **Role 1**: some SW components, i.e., interrupt service routines (ISR), should *cooperate* with raw HW components to fulfill their expected functionalities.
- **Role 2**: some SW components, called *soft-transfers*, should *stimulate* hardware to check if they work as expected. For example, a subroutine with intensive memory access could stress memory modules; a subroutine with intensive ALU operations could stress the ALU in the processor itself.
- **Role 3**: some SW should *manage* system-level concurrency by efficiently scheduling HW and SW behaviours.

These roles contribute differently to system-level verification. Role 1 is actually a part of the DUT, Role 2 represents some actual test-cases to the DUT, and Role 3 manages test-cases on the DUT. Role 3 serves as the backbone of the verification SW. It is relatively independent from an actual SoC DUT. In this section, we specifically regard the SW playing Role 3 as the "test-program" (TP).

Role 1, 2 and 3 SW components respectively resemble the SW components running on a general-purpose computer, i.e., (1) hardware drivers, which fulfill hardware functionalities, (2) user processes, which carry out the user defined tasks, and (3) operating system (OS), which schedules user processes.

Because of the similarities between (i) ISR/soft-transfer/TP and (ii) driver/user-process/OS, concepts applicable to the latter may lead to relevant considerations for the former.

- The interaction-objects that the OS manages are processes; the interaction-objects that the TP manages are transfers.
- Just like an OS that manages the parallelism of processes, a TP should manage the concurrency of transfers.

- An OS kernel use a data-structure called "process-control-block" (PCB) to record each process' scheduling information; similarly, a TP needs some data structure to record the running status of transfers.
- From computer users' point of view, the OS is the *overhead* while their user-processes are the *payload*; similarly, in a verification engineer's point of view, the TP (i.e., the SW in Role 3) is the *overhead*, while other SW components (SW in Role 1 and 2) are the *payload*.
- An OS is event-driven SW; a TP is better to be event-driven as well.

Nevertheless, a TP needs to be much simpler than a general-purpose OS. This is because a TP and an OS are dealing with parallelism at different levels, which could be quantified in terms of the *temporal granularity* of the interaction-objects they are managing. An OS maintains process-level-parallelism (PLP). The CPU-slice allocated of a user process running on a time-division-multi-task OS is at the millisecond level, or in the order of 10^6 CPU cycles. The *life expectancy* of a process is much longer. In contrast, a TP is supposed to maintain the concurrency between transfers, whose optimal *life expectancy* is in the order of 10^4 CPU cycles [23]. As a result, a TP enjoys substantial simplicities that are not shared by a general-purpose OS. The simplicities include (i) a transfer's resource usage is *static* (while a process uses resources dynamically), and (ii) concurrent transfers do not communicate with each other (while processes need to communicate each other).

These simplicities make TP implementation easy (and TP automation feasible). For instance, in principle, the data-structure for transfer scheduling purpose could be as simple as a *single* bit (1 for running and 0 for not-running). In the Nios SoC, we use an integer, called "running-flag", to store transfer status. Some running-flags carry more information such as error status and transfer result, while other running-flags do contain only one bit of information.

ISRs (Role 1) and soft-transfers (Role 2) are system-specific, so they basically require manual programming. However, the IO mindset provides useful guidelines to help programming. For instance, ISRs should map *physical interrupts* into *transfer event* and reflect these events in running-flags.

D. TP-TB Communication

In the "UART-RX-by-DMA" example in Section III-B, the TP must set up the RX transactor (which is in the TB), just as the TP does similar settings for the UART and the DMA engine (which are in the DUT). This shows that TP views TB and DUT essentially as *peers*. Hence, there should be a mechanism to allow the TP to communicate with the TB, just as the TP can communicate with the DUT through the control/status registers (usually referred as the "programming interface"). This mechanism will help to hide the difference between "TB-to-DUT stimulations", "DUT-to-TB responses" and "DUT internal behaviours". In fact, TB and DUT become one single *DUT-TB super-system*, under TP's unified control.

However, the fact that TB components are *not native* to DUT applies some constraints to the implementation of the mechanism:

- the TP cannot communicate with each TB component *directly*, because TB components don't have their own programming interface;
- TB-to-TP communication is harder than TP-to-TB communication, because TB components cannot interrupt CPU as many DUT components can.

These constraints suggest the need to implement a *central control* in the TB and let TP communicate with it through a centralized interface. Figure 4 illustrates this TP-TB communication mechanism. Using reserved memory locations as the *TP-TB interface*, TP issues logical-level commands and parameters to the *TB central control* (TBCC), which in turn manages the rest of TB *without TP's further intervention*, or, as we call it, *autonomously*.

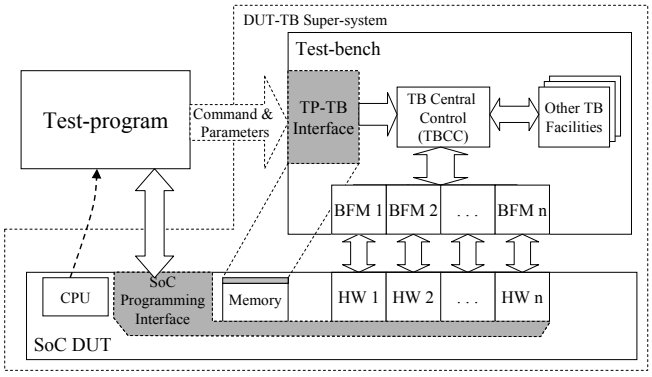


Fig. 4. With TP-TB interface, TP views TB and DUT as a single TB-DUT super-system.

This communication mechanism is mostly used for configuring bus-functional-model (BFM) behaviours; but it also allows the TP to gain some controllability over the HW simulation environment. For instance, if the TP detects an unexpected SW failure, it may choose to instruct the TB to pause the simulation. This useful feature provides valuable debugging opportunities *without* the support of an application SW debugger, which is unavailable (and also inappropriate) at the HW integration stage.

The cost to implement a centralized TP-TB interface is low. For simulation, it simply consumes a few memory locations; if TB and DUT are in the form of HW prototypes, the interface requires a couple of dedicated registers.

IV. The TB's Roles in TP-centric Verification

A. The TB's Role in TP-centric Verification

The control and observation capabilities provided by the TP and the TB have their respective strengths and limitations, as listed in Table I. It should not be too surprising to see that the TB's strengths exactly complement the TP's limitations and vice versa, since the TP is SW, slower but flexible, and the TB is essentially HW, faster but less flexible.

What kind of responsibilities should the *TB* take in a *TP-centric* verification approach? Some of the TB's responsibilities are already shown in Figure 4:

	DUT Control	DUT Observation
TP	•Logical level write-operation; •Native to DUT; •Sequential but highly orchestrated.	•Limited to read-operation and interrupt mechanism; •Affecting DUT's behaviour.
TB	•Physical level signal-feeding; •Brute-force to DUT; •Parallel but poorly coordinated.	•Ubiquitous observation; •Non-instructive to DUT's behaviours.

TABLE I
TP AND TB'S CONTROLLABILITY AND OBSERVABILITY.

- The TB-central-control (TBCC) module should implement the so-called "autonomous control" over various TB components. TBCC does not need too much intelligence, since it simply performs whatever the TP instructs it to do. This kind of autonomous control is much simpler than the controllability required in a TB-centric verification approach.
- Each TB component performs whatever the TBCC tells it to do, probably filling up some details. For example, the UART RX transaction should feed stimuli to the UART with properties specified by the TP; the transactor itself determines any details not specified by TP. Again, these mechanical tasks do not need much intelligence.

Besides the rather mechanical "control", the TB is especially suitable for "observation" tasks. The term "ubiquitous observation" in Table I refers to TB's potential to observe anything (i.e., any flip-flop or wire) at anytime (i.e., at any simulation cycle). This type of observation is *non-intrusive* to the DUT's behaviours. In contrast, SW observes HW only through read operations; and these operations modify DUT's behaviours.

Traditional observation tasks TB performs include:

- Test result checking;
- HW property assertion;
- HW event dumping.

While it makes sense to let the TB observe the DUT at physical level, we might wonder whether the TP can act as a proper logical-level observer. For instance, one transfer "DMA-copy 1,000 bytes from address 0xa000 to address 0xf000" took 2,000 DUT cycles to complete; in order to check the transfer result, is it appropriate to compare 1000 destination and source locations, *byte-to-byte in SW*, which may consume 20,000 cycles? Obviously, SW is not in the position to perform such an observation task on a large amount of data. One typical memory-read operation native to a SoC not only consumes 1 to 10 DUT cycles (the *simulated time*), but also wastes *simulation time* (i.e., the simulator's computation time). In contrast, a behavioural checker implemented in the TB reads data two to three orders of magnitude faster than SW in terms of *simulation time*, and consumes strictly zero *simulated time*. SW-based checking may supplement TB-based checking in special cases where no massive data operations are needed (or find its applicability when TB and DUT are prototyped as true HW).

To summarize, in a TP-centric verification approach, the TB essentially plays passive roles. Since the more active role of parallelism management is taken by the TP, the TB is not required to have too much intelligence in itself. Nevertheless, the TB should provide powerful observation mechanisms. These requirements match the overall profile of nowadays HDLs/HVLs.

B. Experiment: TB for SW Profiling

A TB's powerful observation capabilities are directly applied to HW behaviours. But since we are using the TP-centric approach, it is very natural to inquire TP's behaviour, or more generally, SW's behaviour. This kind of observation is both (a) *necessary* since the SW's behaviours could suggest how much it is stressing HW, and (b) *possible* since the so-called "SW's behaviour" is derived from CPU's behaviour, which is definitely under the TB's ubiquitous observation.

On a general-purpose computer, SW *profiling* is a powerful tool that observes SW's behaviours, helping SW engineers to improve SW quality. Profiling provides critical SW performance information, including CPU-time spent in user mode and in system-calls, and the calling relations among functions. This kind of SW profiling needs support from the OS. Likewise, in TP-centric verification, if a certain profiling facility is provided, verification engineers can evaluate TP's performances, improve TP quality, find coverage holes and/or anomalies, and compare TPs obtained through different methodologies. However, to perform TP profiling through OS support is infeasible. Also, SW profiling should not be confused with the HW profiling facility [24] provided by the simulation environment, which gives information on the real-world time spent in simulating each HW component. The profiling information we are expecting includes 1) function calling relations, 2) function execution time, and 3) interrupt-related information. .

Another type of SW behaviour refers to SW's accesses to critical data structures, such as transfers' running-flags (see III-C), whose dump-file contains coverage information regarding transfers' concurrency completeness [23]. We focus on profiling in this paper.

To observe TP's behavior, TB simply collects data from critical registers/memory locations during simulation. It is possible to let the TB also analyze these data at *run time*; but for efficiency reasons, analysis is usually performed by powerful *post-simulation* utilities, which can be regarded as extension of the TB. Figure 5 shows the way we implement the SW observation mechanism. Essentially, by comparing with (or referring to) the address-symbol table (generated by TP's compilation tool), the TB monitors the program counter and specific memory locations, and dumps critical events, including context-switching and value modification, into dump-files. It is worth mentioning that both monitors' behaviours in Figure 5 could be constructed automatically using the information from the address-symbol table.

Figures 6 to 8 demonstrate that various SW performances can be extracted by profiling. The DUT in the experiment is a

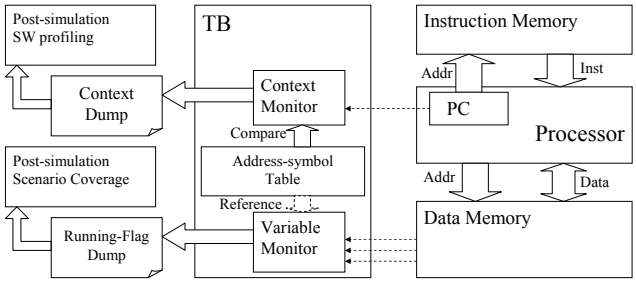


Fig. 5. TB observes SW behaviour.

Nios SoC (see Figure 2) in register-transfer-level Verilog. The TP is made of a set of functions, total size 49 KB-executable, including a 37 KB, event-driven function `scheduler` (automatically generated using the TRG method). The TP took 42 million DUT-cycles to run, simulated in 45 minutes on a 3+GHz workstation; it produced a 12 MB dump in text format, containing 355k context-switchings, analyzed in 25 seconds by our profiling utility written in Python language. The main commercial tools include (i) Synopsys's VCS [24] for Verilog simulation and (ii) the Nios SoC SW development tool-chain for TP compilation.

Time-unit: Simulation-cycle (20 Simulation-cycles = 1 DUT-cycle)						
Function	**Entry**	**INT'd**	**Calls**	**Exec w/ calls**	**Exec w/o calls**	**Prof (%)**
main	1	2654	141	826947860	163721980	19.80%
memoryblkrevbyCPU	1098	5714	235	174466360	130329820	15.76%
do_logic	233	5632	0	160023220	111371640	13.47%
uartISR	17489	894	1106	168592040	108945940	13.17%
recursive_fibo	129356	4501	128678	17610769620	102495880	12.39%
CWPUnderflowTrapHandler	1194	0	0	60685220	60685220	7.34%
scheduler	2169	629	2051	662509920	36686160	4.44%
CWPOverflowTrapHandler	1194	0	0	26475040	26475040	3.20%
uartISR_wrapper	17489	894	17489	213993300	25007320	3.02%
txdata1pollingbyCPU	32	618	0	27560660	22703540	2.75%
rxdata1pollingbyCPU	10	318	0	19655500	15876180	1.92%
TimerISR	3202	730	1123	37719960	10296280	1.25%
TimerISR_wrapper	3202	564	3202	69227060	4597580	0.56%
Unknown	75	0	0	2419900	2419900	0.29%
dmaISR	269	39	259	4640460	1356700	0.16%
fibonacci	678	26	678	238660720	1350020	0.16%
TerminateRunningRX	389	0	0	856200	856200	0.10%
PrepareNextScenario	70	0	84	3206680	740260	0.09%
__mulsi3	235	21	0	719140	576720	0.07%
dmaISR_wrapper	269	114	269	6828020	408960	0.05%

Fig. 6. Function profiling.

Figure 6 provides basic information associated with each function. The rightmost three columns report each function's CPU time, including/excluding time spent in callings and interrupts, and percentage of CPU time. This report can tell how the profiling of payload and overhead (Role 1–3 functions, see III-C) matches our expectation. This particular report says that the Role 3 (function `scheduler`) takes 4.44%, while Role 1 (all trap handlers, ISRs and ISR wrappers) amounts to 28.75%, and Role 2 (all remaining functions except `main`) accounts for 47.01%. So we conclude that the paylaod/overhead profiling basically matches our expectation.

This profiling could guide us to further improve SW structure. For instance, the `main` function accounts for 19.8%, in which the CPU is basically locked (until interrupted) in a strictly infinite loop of doing nothing (C code: "`while 1;`") to emulate CPU *idling*. The idle time can be exploited to contribute to HW verification. We could replace "`while 1;`" with "`while 1 CPU_self_test();`". (Looping the self-testing is meaningful, since each loop will experience different interrupt situations.)

Although the CPU is essentially a sequential device, due to interrupts, we still can obtain information regarding parallelism from SW profiling. Figure 7 indicates how many times each function is preempted by ISRs. From this report we can identify a coverage hole in the parallelism: the DMA ISR had never preempted function `__mulsi3`. (Many other zeros are reasonable and don't represent coverage holes.)

Report 2: Function-Interrupt Cross
(Warning: Function names may be truncated)

Func/INT	**dmaISR**	**TimerI**	**uartIS**	**CWPUnd**	**CWPOve**
main	34	111	2509	0	0
memoryblkrevbyCPU	77	1205	4432	0	0
do_logic	62	866	4704	0	0
uartISR	0	0	0	894	0
recursive_fibo	69	694	3398	170	170
CWPUnderflowTrapHandler	0	0	0	0	0
scheduler	11	84	534	0	0
CWPOverflowTrapHandler	0	0	0	0	0
uartISR_wrapper	0	0	0	0	894
txdata1pollingbyCPU	9	188	421	0	0
rxdata1pollingbyCPU	7	30	281	0	0
TimerISR	0	0	608	122	0
TimerISR_wrapper	0	0	442	0	122
dmaISR	0	7	24	8	0
fibonacci	0	2	24	0	0
TerminateRunningRX	0	0	0	0	0
PrepareNextScenario	0	0	0	0	0
__mulsi3	***0***	6	15	0	0
dmaISR_wrapper	0	9	97	0	8
UpdateEnvironment	0	0	0	0	0
caching	0	0	0	0	0

Fig. 7. Function-interrupt cross.

Figure 8 reports more detailed information: all exact interrupt nesting sequences that occurred in the simulation. The report says that the deepest interrupt nesting sequence first occurred at simulation cycle 640220650, when a recursive function was preempted by the DMA ISR, which was in turn preempted by the Timer ISR, which was in turn preempted by the UART ISR, which was finally preempted by a register-window underflow handler. That this extremely rare corner-case has happened three times in this simulation-run is eventually the result of our TP-centric and interaction-oriented TRG method.

```
Report 5: Interrupt Nesting Sequences

Overview: 94 Interrupt Sequences Encountered.
Warning: Function names in the history may be truncated.

Times  Dep      Debut@ Stack_History(INT'EE>INT'ER)..
 2509    1     6158890 main>uartISR
 3398    1    43254430 do_logic>uartISR
  609    2    43256390 recursive>uartISR>Underfl
  609    2    43288350 recursive>uartISR>Overflo
   69    1    63326090 do_logic>dmaISRa
    . . .
    3    3   640195150 recursive>dmaISRa>TimerISR>Overflo
    3    4   640220650 recursive>dmaISR>TimerISR>uartISR>CWPUnder
    3    4   640249890 recursive>dmaISR>TimerISR>uartISR>CWPOverf
    3    3   640164930 recursive>dmaISRa>TimerISR>CWPUnderfl
    3    3   640185950 recursive>dmaISRa>TimerISR>uartISR
```

Fig. 8. Interrupt nesting.

To summarize, by extracting rich information about SW performances, the profiling mechanism provides valuable opportunities to monitor/improve simulation quality. Furthermore, since this mechanism is *independent* of the TP-generation method, it can serve as a *fair comparison* of different TP-generation methods. TB's *ubiquitous observation capability* has shown its power in this profiling mechanism.

V. Conclusion

In this paper the problematic relationship between TB, TP and DUT is identified. We propose a TP-centric, interaction-oriented verification methodology to resolve the problem. With confusing dualisms (TB-DUT, test-OUT) removed, TB, TP and DUT now enjoy natural and harmonious relations, in which each of TB and TP is exerting its strengths to make up the other's limitations. As suggested in Figure 9, TB and DUT form a "DUT-TB Super-system", which is under TP's *unified control*; while TP and DUT form a "HW-SW Super-system", which is under the TB's *ubiquitous observation*.

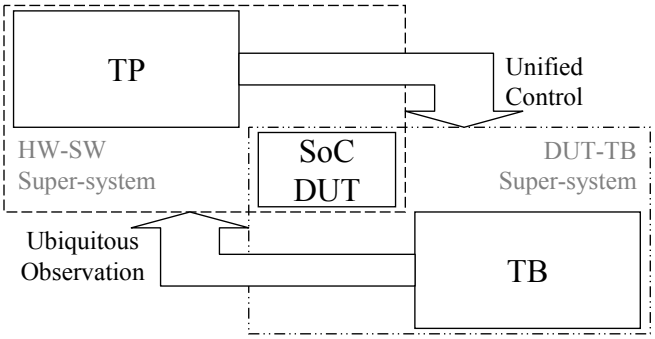


Fig. 9. Position TP and TB in SoC verification.

Methodology	Component-oriented	Interaction-oriented
TP-centric	•Not very exciting common practice •HW diagnostics	•Emerging, this paper •SoC verification
TB-centric	•Traditional •Component-level verification	?

TABLE II
Methodology Differentiation.

We should realize that "TB-centric/TP-centric" and "component-oriented/interaction-oriented" are two orthogonal differentiations. We have gone through three out of total four combinations, as suggested in Table II. An interesting question, leading to our future research, is that what features a TB-centric but interaction-oriented methodology will have and what will be its application. The TP-centric method we have proposed in this paper would shed some light on that question.

References

[1] G. Mosensoson, *Practical Approaches to SoC Verification*, Proceedings of DATE User Forum, 2002, access at http://www.cecs.uci.edu/ics259/05-08-03/verisitySOCverify.pdf

[2] F.Bacchini, G.Moretti, H.Foster, J.Bergeron, M.Nakamura, S.Hehta, L.Ducousso, *Is Methodology the Highway Out of Verification Hell?*, Proceedings of the 39th conference on Design automation, p521-522, 2005

[3] VHDL Cohen Publishing, *Transaction-Based Verification in HDL*, accessed at http://members.aol.com/vhdlcohen2/vhdl/veriflang.pdf

[4] R.Goering, *Ten 2008 trends in system and chip design*, SCD Source Online Article, Feb 2008, accessed at http://www.scdsource.com/article.php?id=68

[5] M. Karppanen, P. Nivalainen, H. Talesara, *SoC Verification: A Layered Testbench Architecture, A System for Regression Management, A Coverage Methodology*, Synopsys User Group, 2003

[6] T.Anderson, J.Bergeron, E.Cerny, A.Hunter, A.Nightingale, *SystemVerilog reference verification methodology: RTL*, EETimes Design News, 2006, accessed at http://www.eetimes.com/news/design/showArticle.jhtml;jsessionid=AWR4UEJSI1KZWQSNDLOSKHSCJUNN2JVN?articleID=187001913&pgno=1

[7] B.Bailey, *Verification Languages and Where They Fit*, in Proceedings of EdaForum 2003, 11, 2003, accessed at http://www.edacentrum.net/edaforum/downloadables/dateien/mentor-seminar.pdf

[8] A.Cheng, A. Parashkevov, C.C. Lim, *Verifying System-on-Chips at the Software Application Level*, in Proceedings of IFIP-WG Conference on Very Large Scale Integration System-on-Chip, Perth, p586-591, Oct 2005

[9] G. Berry and L. Blanc and A. Bouali, J. Dormoy, *Top-level validation of system-on-chip in Esterel Studio*, HLDVT '02: Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop, 2002

[10] R. Emek, I.Jaeger, Y.Naveh, G.Bergman, G.Aloni, Y.Katz, M.Farkash, I.Dozoretz, A.Goldin, *XGEN: A Random Testcase Generator for Systems and SoCs*, in Proceedings of the 7th IEEE International High Level Design Validation and Test Workshop (HLDVT02), p145-150, 2002

[11] J. Andrews, *Unified Verification of SoC Hardware and Embedded Software*, Chip Design Online Magazine, 2007, accessed at http://www.chipdesignmag.com/display.php?articleId=1267

[12] J. Kenney, *Using a processor-driven test bench for functional verification of embedded SoCs*, Embedded.com online document, access at http://www.embedded.com/columns/showArticle.jhtml?articleID=193104351

[13] J. Xu, C.C.Lim, *Modeling Heterogeneous Interactions in SoC Verification*, in Proceedings of International Conference on Very Large Scale Integration and System-on-Chip, Nice, France, p98-103, Oct 2006

[14] J. Xu, C.C.Lim, *Exploiting Concurrency in System-on-Chip Verification*, in Proceedings of IEEE Asia Pacific Conference on Circuits and Systems, Singapore (APCCAS 2006), p836-839, Dec 2006

[15] S. Ezer, S.Johnson, *Smart Diagnostics for Configurable Processor Verification*, DAC '05: Proceedings of the 39th conference on Design automation, p789-794, 2005

[16] C.Johns, D.Mihal, D.Pierce, *Architecture for Simulation Testbench Control*, US Patent 6,651,038 B1, Nov 2003

[17] S.Copty, I.Jaegel,Y.Katz, *Path-Based System Level Stimuli Generation*, Lecture Notes in Computer Science, Vol 3875/2006, p1-13, 2006

[18] R.Emek, Y.Naveh, *Scheduling of Transactions for System Level TestCase Generation*, in Proceedings of the 8th IEEE International High Level Design Validation and Test Workshop (HLDVT03), p149-154, Nov 2003

[19] Nios Hardware Development Tutorial ver 1.2, 2004, Altera Inc. access at http://www.altera.com/literature/tt/tt_nios_hw.pdf

[20] D.Brahme, S.Cox, J.Gallol,M.Glasser, W.Grundmann, C.Ip, W.Paulsen, J.Pierce, J.Rose, D.Shea , K.Whiting, *The Transaction-Based Verification Methodology*, Technical Report, Cadence Berkeley Labs, 2000

[21] L.Cai and D.Gajski, *Transaction Level Modeling: An Overview*, in Proceedings of International Conference on Hardware-Software Codesign and System Synthesis, Newport Beach, California, Oct, 2003

[22] R. Milnor, *Communication and Concurrency*, p12, New York: Prentice Hall, 1989

[23] X.Xu, C.C.Lim, *Using Transfer-Resource Graph for Software-Based Verification of System-on-Chip*, IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems, Vol. 27, Issue. 7, July 2008

[24] *VCS/VCSi User Guide*, Ver 7.0, Synopsys.inc, 2003