Using Transfer-Resource Graph for Software-Based Verification of System-on-Chip

Xiaoxi Xu and Cheng-Chew Lim, *Senior Member, IEEE*

***Abstract*—The verification of system-on-chip is challenging due to its high level of integration. Multiple components in a system can behave concurrently and compete for resources. Hence, for simulation-based verification, we need a methodology that allows one to automatically generate test cases for testing concurrent and resource-competing behaviors. We introduce the use of a transfer-resource graph (TRG) as the model for test generation. From a high abstraction level, TRG is able to model the parallelism between heterogeneous interaction forms in a system. We show how TRG is used in generating test cases of resource competitions and how these test cases are structured in event-driven test programs. For coverage, TRG can be converted to a Petri net, allowing one to measure the completeness of concurrency in simulation.**

***Index Terms*—Concurrency, coverage, event-driven, resource-contention, simulation, system-on-chip, test-generation, verification.**

I. Introduction

THE SYSTEM-ON-CHIP (SoC) solution has become a popular very large-scale integration (VLSI) design paradigm. In addition to many advantages, including lower production cost and higher performance, this design paradigm has practically reduced the design process of a complex system into integrating some predesigned and reusable components. However, very thorough verification must be done to check the correctness of the SoC design. Multiple sources claim that verification complexity is growing exponentially [1], with about 50%–70% of the design time and efforts spent on design verification.

The soaring verification complexity stems from the large scale of hardware (HW) integration. HW components verified to be compliant with register-transfer-level (RTL) specification do not guarantee that they will work together as expected. To deal with this problem, an SoC must be hierarchically verified. That is, each component needs to be individually verified. Then, each subsystem made up of closely related components is verified. Finally, the whole system is verified. Each verification stage has its own emphasis and goal. At subsystem and system levels, component-to-component interactions (instead of the components themselves) should be the verification emphasis. The target bugs at this stage are those that are deeply buried in interactions between components and not the bugs that are local to one component.

Manuscript received August 2, 2007; revised November 15, 2007 and January 9, 2008. This work was supported by the Australian Research Council under Linkage Project LP0454838. This paper was recommended by Associate Editor R. F. Damiano.

The authors are with the School of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, SA 5005, Australia (e-mail: justinxu@eleceng.adelaide.edu.au; ccl@eleceng.adelaide.edu.au).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2008.923092

Besides the HW components, software (SW) also contributes to the full SoC functionality. Therefore, checking HW–SW interactions becomes an important part of the SoC verification process.

Separately checking the HW-to-HW and HW-to-SW interactions is still insufficient to fully verify a whole system because a system that concurrently runs heterogeneous forms of interactions is subject to various unforeseeable implementation bugs. Listed in [2], unique bugs at the system level include:

1) interactions between blocks that are assumed verified;
2) conflicts in accessing shared resources;
3) arbitration problems and dead locks;
4) priority conflicts in exception handling;
5) unexpected HW/SW sequences.

All these bugs are related to interactions, particularly to concurrent ones with resource competitions. Concurrency is the central characteristic of a system; therefore, concurrency is the key to system-level verification. Hence, a simulation-based verification methodology should provide the following: 1) a method to generate test cases of concurrency and 2) a method to quantify the concurrency completeness in terms of the temporal relations between concurrent interactions.

This paper proposes to use a model called transfer-resource graph (TRG) to deal with the aforementioned two issues. Test cases of concurrency can be generated from TRG. Furthermore, TRG can be transformed into a Petri net to allow one to quantify the temporal relations of concurrency.

We adopt the SW approach [3] of generating test cases in the form of SW [test programs (TPs)] to be executed by the SoC processor. The relations among an SoC, its test bench, and the SW are conceptually shown in Fig. 1. The SW bears the responsibilities of stimulating the SoC and maintaining a high level of concurrency; thus, the role of the test bench is reduced to observing (including error detection and event logging) the SoC design (the test bench still provides physical-level stimulation to the design but under the control of the SW). Therefore, by the SW approach, a considerable amount of time and effort could be saved in the test-bench development.

Although these practices (using the same model for the test-generation and coverage measures and using the SW to test the HW) are not new, the novelty of our approach is to generalize HW and SW behaviors and their cooperations at an elevated abstraction level so as to systematically organize the parallelism between heterogeneous interactions in TPs.

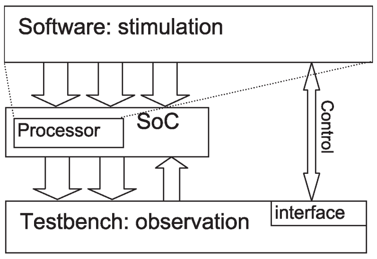


Fig. 1. Relations among an SoC, its test bench, and the test SW.

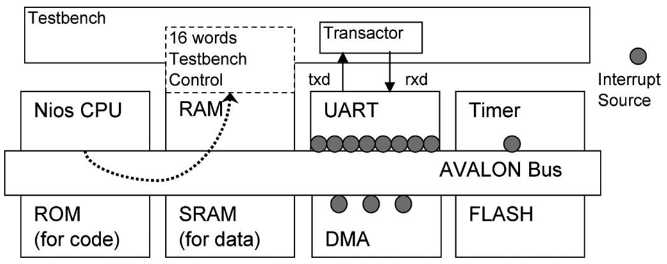


Fig. 2. System under demonstration—the Nios SoC.

The rest of this paper is organized as follows. Section II introduces some related works, focusing on the relations between the test-generation and coverage measures. Section III focuses on our interaction model transfer. Section IV formally introduces TRG and its use for test generation. Section V discusses the TP structures supported by TRG. Section VI discusses how TRG is transformed to a Petri net as the postsimulation stage coverage model. The experimental results in Section VII illustrate some quantitative aspects of the TRG method. Section VIII concludes this paper.

II. Related Works and Background

Test-generation and coverage measures are the two most important aspects of simulation-based verification. They can be regarded as two relatively independent tasks.

Tests can be generated without an apparent model of a SoC. Using randomization is the conventional way to produce stimulation to a design. This method has intrinsic problems in test quality, whereas complex designs, such as SoC, need a number of high-quality tests to check their functionalities and performances. Hence, tests generated without a system model cannot meet the SoC verification requirements. The main issue is that those tests do not take the fundamental characteristic of a system, namely, the concurrency, into account.

Model-based generation is more likely to produce high-quality tests of concurrency. XGEN [4], [5], which is the test generator used at IBM for system verification, models interactions as the building blocks for test-SW generation. An interaction is a series of communication stages known as "acts"; each act is performed by some HW components. With the user's interventions, XGEN is able to produce high-quality tests by arranging concurrent behaviors. The modeling stage of XGEN may need considerable efforts. In addition to modeling interactions, the HW components, with their functionalities, also need to be modeled in detail, which requires in-depth HW knowledge.

Coverage measures indicate verification completeness. At the system level, conventional RTL statement-based coverage measures give little information about concurrency. The completeness of concurrency can be quantified in terms of the temporal relations between events. Kwon *et al.* [6] propose that users first establish a hierarchical–temporal–event–relation graph to represent the interactions between communicating HW components; then, an algorithm based on the graph can calculate coverage space, which will be much smaller but more meaningful than a simple cross-product coverage model. This method can generate an accurate coverage space. To build such a graph, the users (verification engineers) must have an accurate view of signal-level timing dependences between components.

We realize that in verifying a complex design such as SoC, the requirement of in-depth knowledge about HW implementation is usually the bottleneck for building a consistent system model, which is crucial for improving the test quality or for accurately computing the coverage space. Our proposed TRG model (first introduced in [7]) addresses this issue at a higher level of abstraction. Compared with the "interaction" model in XGEN, our model, which is called "transfer," is more concise and atomic. Furthermore, no HW details need to be modeled in TRG. Instead, the TRG model naturally represents a programmer's view of a system. The test quality is maximized by fully exploiting the concurrency [8]. However, TRG has not been formally defined, and its application for coverage is not yet explored. This paper formally defines TRG and introduces its usage in defining coverage spaces.

We demonstrate our methodology on a Nios SoC [9], as shown in Fig. 2. This SoC is simple (~25 000 lines of Verilog code) but contains adequate features of a typical system, which are multiple components (including the CPU) interconnected by on-chip bus. The Nios CPU is a pipelined reduced instruction set computer. The direct memory access (DMA) can perform data transfer between any slaves on the Avalon bus. The full-duplex-capable universal asynchronous receiver/transmitter (UART) is the communication interface of the system. ROM and static RAM (SRAM) store instructions and data, respectively. RAM and Flash are the additional memory modules. A number of interrupt sources exist and will be serviced by the CPU. The test bench accepts the SW's control via an interface implemented in the RAM.

III. Transfer Model

A. Overview

For system-level verification, it is critical to have a system model at a suitable abstraction level.

We should focus on modeling the interactions between components rather than on the components themselves. Furthermore, the interaction being modeled should be more abstract than signal-level transactions. This is because we are to implement tests in an SW known as TP, which has little controllability and observability of signal-level events (e.g., Bus_Request, Bus_Acknowledge, etc). However, the abstraction level should not be too high because TP is supposed to closely stress the SoC HW. It is inappropriate for a TP to view devices as services provided through application program interfaces.

To trade off the aforementioned considerations, the model should be readily comprehended by a device-level programmer who understands the HW functionalities and performances, but who may have little knowledge about HW implementation. We use the term "transfer" to represent the interaction at this specific abstraction level.

B. Definition of Transfer

Interactions become the focus of system-level verification. There is a challenging issue associated with modeling them, that is, interactions come in various forms, requiring different techniques to stimulate and observe them. Some interaction examples in the Nios SoC are the following.

1) A Flash-to-RAM DMA transfer. It is a series of read/write (R/W) operation driven by the dedicated HW—the DMA engine.
2) The execution of a `sort` subroutine. It can be viewed as a pattern of memory access performed by the CPU, which is driven by the execution of SW.
3) An incoming byte stream via the UART receiver. The stream finally reaches a memory buffer. This process is mostly driven by the interrupt mechanism.

Note that these three examples are data flows driven by heterogeneous mechanisms, which are, respectively, the DMA engine, `sort` subroutine, and interrupt subsystem/interrupt handler. While checking each of them is commonsense, checking their parallel execution will greatly improve the test quality, considering that we are able to observe not only interactions but also the interference between interactions. If the aforementioned three interaction examples take place in parallel, we will observe how the DMA engine and the CPU compete with each other for the bus access, how the UART will interfere with their competition by frequently interrupting the sort subroutine, and how the UART interrupt will be nested in the DMA interrupt.

The key to effectively constructing such parallelism is to generalize heterogeneous interaction forms into a common model. We call this model transfer type.

Definition 1: Transfer type is a set of programmer-controlled and data-intensive interaction patterns among SoC components. Its programmer-controlled feature means that a transfer type has the following properties.

1) Configuration: Transfer types have their own parameters, which can be configured by some instructions (an important part of a transfer type's configuration is the resources to be used).
2) Invocation: Transfers can be invoked by some instructions. Invocation instructions are allowed to have side effects of configuration.
3) Notification: Events of transfer completion can be notified to SW in some way (e.g., via interrupt), so that some SW flags can indicate these events.

Definition 2: Transfer instance (or simply transfer) is a specific configuration of a transfer type.

Note that configuration, invocation, and notification are the overhead of a transfer and that the main body of a transfer is the data flow. Fig. 3 shows the life cycle of a transfer, which includes a data and a control phase. Transfers embody the

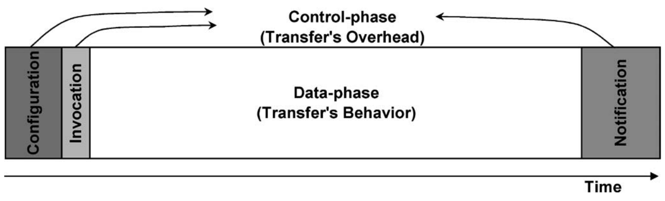


Fig. 3. Transfer life cycle.

interactions that the design allows and the application requires; therefore, transfers are simple test cases in their own right.

C. Expression Power of Transfer

In the early stage of an SoC design/verification cycle, the level of abstraction should be high enough to hide the differences between HW and SW behaviors [10]. Our transfer-type model meets this requirement. The three examples of interaction in Section III-B can be expressed as transfer types.

1) Transfer type "Flash-to-RAM DMA"
   a) Configuration: initial-source-address, initial-destination-address, width (8, 16, or 32-bit), and length;
   b) Invocation: set DMA engine control register go bit;
   c) Notification: DMA finish interrupt.
2) Transfer type "Sorting"
   a) Configuration: address, data type (signed/unsigned integer, etc), length, sort-algorithm, and reverse;
   b) Invocation: call subroutine `sort`(`address`, `type`, `length`, `algorithm`, `reverse`);
   c) Notification: the return of the subroutine.
3) Transfer type "UART-Rx-by-Interrupt"
   a) Configuration: end-of-packet character, max length, finish mode (by max length and/or end-of-packet character), and error-detection mode (`parity`, `frame`);
   b) Invocation: A STORE instruction to a special address—the test-bench/TP interface; when this address is written, the test bench starts to feed the SoC with a bit stream;
   c) Notification: The UART interrupt handler detects the finish conditions of the UART receiver.

More generally, the transfer-type model can describe three categories of data-intensive interactions, which are as follows.

1) **HW behaviors (hard transfers)**: The R/W operations on the bus are driven by master devices, whose behaviors are mostly hardwired. Therefore, they are categorized as hard transfers.
2) **SW behaviors (soft transfers)**: A processor in SoC is a valid master device, whose behaviors are programmable rather than hardwired. Therefore, its behaviors are called soft transfers.

   There is a subtle but crucial difference between the code in a soft transfer and the code in configuring a transfer. The former should be regarded as the payload code and is subject to verification, and the latter is treated as the overhead for verification. A consequence is that the former is application oriented and usually requires manual development. In contrast, the latter should be automatically organized in TP.

TABLE I
IMPLEMENT DIFFERENT CATEGORIES OF TRANSFERS

Transfer Category	Transfer-type Examples	Configuration	Invocation	Notification
Hard-transfers	Any DMA transfer	Setting control-registers	Setting control-registers	Master (or slave) interrupting CPU
Soft-transfers	UART polling; internal sorting; recursive subroutines; processor-self-test-subroutines	Setting control-registers; Passing arguments to subroutines	Calling subroutines	Subroutines' return
Virtual-transfers	UART `trdy/rrdy` ISRs; Timer `time-out` ISR	Setting control-registers; Setting global variables	Enabling interrupt sources	The virtual master (ISR) itself

One guideline to build soft transfers is to compose R/W intensive subroutines to stimulate the interactions between CPU and slaves. However, soft transfers do not have to literally transfer data; they can be computation-intensive operations to apply stress to different types of physical resources.

3) **HW/SW cooperation (virtual transfers)**: In the Nios SoC, the incoming UART byte stream is formed by the cooperation between the UART, the interrupt subsystem, and the UART-receiver-ready interrupt service routine (ISR). Although the byte stream is physically performed by the CPU, from a higher level of abstraction, it is functionally equivalent to perceive that a virtual master (also see Section IV-B) is conducting the stream between the receiver and a memory buffer. This virtual master is independent from the CPU which may be involved in another task (at a reduced performance). Transfers conducted by virtual masters are called virtual transfers. Unlike a soft transfer, which explicitly requires a real CPU as its resource, a virtual transfer just requires a virtual master; therefore, we can arrange multiple virtual transfers (and one soft transfer) to "concurrently" work on a single CPU. This concurrency is actually the parallel behavior of CPU and peripherals.

In a virtual transfer, the primary forms of interactions are interrupt request and response, whereas the R/W traffic on the bus is secondary.

Table I lists the three transfer categories and summarizes how to implement their configuration, invocation, and notification.

D. Transfer Complexity

To identify the transfer types of a given system, we need to discuss the complexity of the transfer type.

One transfer type's complexity is caused by its configuration. We use $\mathbf{T}$ to denote the set of transfer types in a system, and denote T_i as each transfer-type member. For T_i, each parameter has a set of values to select from. Parameters could be either totally independent of or coupled with each other in various ways. Therefore, T_i's parameter space is very application oriented. Hence, T_i requires an operation $P(\cdot)$ to perform its parameterization. Considering that each T_i has its own specific parameter space, its $P(\cdot)$ should be more accurately denoted as $P_{T_i}(\cdot)$ or, from the object-oriented programming point of view, as $T_i.P(\cdot)$. The complexity of $T_i.P(\cdot)$ can represent the complexity of T_i. To let $T_i.P(\cdot)$ deterministically traverse the whole parameter-space seems neither necessary nor practical. We implement $T_i.P(\cdot)$ using weighted and constrained randomization.

Our transfer model is flexible in the sense that defining a transfer type allows the tradeoff between the number of transfer types in a system and the complexity of their $P(\cdot)$. To one extreme, we could model only a single transfer type to represent all possible interaction patterns in a system but its $P(\cdot)$ needs to deal with a huge but also artificially constrained, parameter space. To the other extreme, we could create a transfer type for each possible interaction pattern of concrete parameters. In this case, we would have a huge number of transfer types, where their $P(\cdot)$'s all have trivial complexities.

In practice, it is natural to adopt this strategy, which is generalizing interaction patterns with a similar parameterization style (including resource allocation style) as one transfer type. Taking the example of the Nios SoC, we initially planned to model 12 transfer types to represent the DMA transactions among four source memory modules (ROM, RAM, Flash, and SRAM) and three destination memory modules (RAM, Flash, and SRAM). Later on, however, we have decided to merge them into one transfer type called "memory-to-memory DMA," with a single but stronger $P(\cdot)$ capable of assigning source and destination among all memory modules. Nevertheless, we consider it more appropriate to model UART-Rx-by-DMA and UART-Tx-by-DMA as separate transfer types, which have very different parameters.

E. Transfer Temporal Granularity

To further characterize transfers, we give an estimation of their life expectancy.

First of all, we discuss the necessity of comparable life expectancies of all transfers. The transfer model enables us to generalize data flows driven by various mechanisms, which could operate in a wide spectrum of data rates. The question is how to "match" concurrent transfers in order to achieve the desired verification quality, i.e., the parallelism and resource contention? For example, does it make sense to create a test case in which a 1000-B-long transfer T_1 at the speed of 10 MB/s runs alongside another 1000-B-long transfer T_2 at 10 KB/s? It appears to be a poor match, considering that the life of T_1 is only 1000th that of T_2, which means that the parallelism exists only in 0.1% of the simulation so that the competition on the shared resource (the bus) is very little. Therefore, it makes sense to configure all transfers to have comparable life expectancies, for example, within one order of magnitude of difference in length.

We now consider how to estimate the optimal life expectancy. Common sense tells us that the life expectancy should not be too long. This is because the simulation is a very time-consuming process. In the shortest time possible, we not only need to cover most configurations for each given transfer type but also to try its concurrent running with other transfers. On the other hand, neither can life expectancy be too short. We regard the data phase of a transfer as its main body, in which parallelism and resource competition are supposed to happen, whereas the transfer's control phase, namely, its configuration, invocation, and notification, is the overhead. Thus, it is natural to require the data phase to be at least one order of magnitude longer than the control phase; otherwise, a considerable portion of the simulation time will be spent on the overhead. Fortunately, the length of the control phase is predictable because all transfer types' configuration, invocation, and notification are made up of instruction sequences of similar length. Hence, we assume that the following quantities are available:

1) the average execution time of transfer configuration C;
2) the average execution time of transfer invocation I;
3) the average execution time of transfer notification N.

Then, we can reasonably conclude that the optimal transfer life expectancy is simply in the range of $(10\sim100)\times(C+I+N)$, which makes the overhead well under 10%.

In the Nios SoC example, $(C+I)$ requires 25 assembly instructions or 100 SoC clocks. Transfer notification is typically by interrupt, which includes the time spent in context switching and ISR execution; thus, the average N is about 350 SoC clocks. Therefore, the optimal transfer life expectancy is in the range of $(10\sim 100)\times(C+I+N)$ or 4500∼45 000 SoC clocks. This estimation guides us on how to model the transfer types and, particularly, on how to bias their $P(\cdot)$ behaviors.

From the earlier discussion, we can quantitatively sense the time granularity of "transfers." This is also the granularity of our proposed "system-level" tests. This granularity is coarser than that of signal-level transactions, which typically ranges from several to dozens of clock cycles. The granularity helps us understand the features and limitations of the system-level tests. For instance, it appears impractical and also unnecessary for a test generator to consider the temporal relations at clock-level accuracy.

IV. Resource and TRG

A. Resource Contentions and Resource Conflicts

The focus of system-level verification is parallelism. The main purpose of constructing parallelism is to observe interesting resource competitions. Resource competitions could happen in various domains, including the on-chip interconnection subsystem, interrupt mechanism, CPU time, and memory locations. An even more intriguing situation is that competitions in various domains can interfere with each other, which is as discussed in Section III-B.

The strength of the transfer model is that it allows these phenomena to be built naturally—we simply arrange multiple transfers to run concurrently. By managing transfer instances' configuration, invocation, and notification, a TP has considerable freedom in arranging parallelism. However, there should exist some principles to prevent the freedom from being reduced to unchecked randomness.

Our principle is to distinguish between resource contentions and conflicts. Resource contentions represent the physical-level competitions that are supposed to be resolved by HW mechanisms (e.g., bus protocol and interrupt handling scheme). These competitions are not just legal but also desirable. Resource contentions are then defined as physical-level resource competitions which a programmer has no direct controllability/observability. In contrast, resource conflicts are competitions at the logical level and require a programmer's discretion in order to avoid. For example, we should allow the DMA engine to compete with the CPU for a physical memory module; however, we require that the DMA transfer should never access the memory addresses that are currently involved in a `sort` subroutine because, otherwise, the results of both processes will not be predictable from their configurations.

Definition 3: Given a set $\mathbf{t}$ of transfers $t_1, t_2, \ldots, t_n$, which are, respectively, instantiated from transfer types $t_1.T$, $t_2.T, \ldots, t_n.T$, we assume that each $t_i.T$ is associated with a `pass`/`fail` Boolean function $t_i.T.Check(t_i.configuration, MemRegSpace)$, which, according to the configuration of t_i, checks if t_i has caused the expected changes (between when t_i is invoked and when t_i is finished) in the memory/register space. If, for all i, $t_i.T.Check()$ is constant regardless of the temporal relations of t_i (sequential, overlapping, etc) with all other transfers in $\mathbf{t}$, we state that $\mathbf{t}$ is free of resource conflicts; otherwise, $\mathbf{t}$ has resource conflicts.

This definition forces some "determinism"—the result of each t_i should be deterministically predicted; however, the determinism is also accompanied by "indeterminism"—the temporal relations between conflict-free transfers are allowed to happen in any way. If there are n conflict-free transfers, with each having a start and an end event, then we shall allow for $(2n)!/2^n$ possible event sequences, all of which shall yield the same results in the memory/register space.

Avoiding resource conflict is reasonable—if each transfer's result can be predicted by its configuration (and the contents in memory/register space), high-level functional checkers, i.e., $T.Check(\cdot)$, can be easily implemented in the test bench. Not enforcing this restriction on resource conflicts is still an option; in that case, the test generator simply has more freedom, but it loses the capability to predict correct results. Therefore, the burden of predicting correct test results is left to the users.

Once the test generator is able to avoid resource conflicts, no other restrictions are preventing it from constructing parallelism. In this way, resource contentions at the physical level are implicitly constructed.

B. Logical Resources

Considering that resource conflict is a logical concept, we only need to model the logical resources in the system. Therefore, there is no need to model HW's specific functionalities. With this simplification, we only model three categories of resources: masters, registers, and memory ranges. We will see that this modeling is not as *ad hoc* as it may seem.

1) **Master:** Master is defined as anything that can conduct a transfer type. Examples of master in our Nios SoC include the read and the write masters of the DMA engine and the data master of the Nios CPU. Once modeled, a master is a trivial resource—the test generator only needs a single bit to indicate its status: available or unavailable. However, the concept of virtual master requires a little more insight into how to interpret system behaviors.

   A virtual master is an ISR that cooperates with the HW to perform data-intensive operations, e.g., the UART receiver-ready ISR is a virtual master performing transfer-type "UART-Rx-by-Interrupt" (see Section III-C). A virtual master is usually capable of only one transfer type; however, we can model as many virtual masters as necessary for a SoC. The number of virtual masters is independent from the number of physical CPUs. Once modeled, the test generator does not distinguish between virtual and real masters. This way, the resource contention on CPU time can be implicitly constructed.

2) **Register:** Registers are also simple resources. We only need to model data-intensive registers visible to programmers. Examples are the UART rxdata and txdata registers.

   Considering that control/status registers across a SoC are not suitable to be treated as data, they are not modeled as register resources. However, in fact, many control/status bits are already implicitly abstracted as masters.

3) **Memory range:** Memory ranges are flexible resources that are dynamically maintained by the test generator. A memory range is an object with properties of base address, size, subword granularity, and R/W mode. From within one free memory range, a test generator can dynamically allocate ranges of suitable size/location to the transfers; meanwhile, the unused fragments become free memory ranges. Allocated memory ranges can reside in the same physical memory module and can even overlap if they are all read-only. In this way, the test generator is able to construct resource contentions on physical memory modules.

   In our current implementation on the Nios SoC, memory ranges do not cross physical memory boundaries. However, this restriction can be lifted if we view the whole memory space as a single free memory range and allocate subranges to transfers. In that case, the corner cases, in which a transfer crosses physical boundaries, can be naturally built. However, this implementation needs to take into account miscellaneous constraints such as the following: ROM cannot be written, memory-mapped registers should be excluded from the address space, and different memory modules may accept different granularities.

Just as transfer types are the generalization of similar transfer instances, the previously discussed logical resource types are the generalization of the bit resources, namely, all bits in memory and registers that are accessible by a programmer. Bit (regardless of data, control, or status bit) is the finest resource object to a programmer; the master, register, and memory range are simply different aggregations of bits. For instance, a physical master device's behavior is controlled/observed by the bits in its control/status registers; it is actually those bits that are abstracted as one logical "master" resource. Therefore, the granularity of a master resource is a few control/status bits. Similarly, a register's granularity is several data bits, and a memory range's granularity consists of numerous (continuous) data bits.

C. Formal Definition of TRG and Scenario

Transfers and resources can be interlinked to form TRG. TRG can be formally defined in terms of transfer instance and bit resource.

Definition 4: A flat TRG is a triple $G = (\mathbf{t}, \mathbf{r}, u)$, where the following are defined.

1) $\mathbf{t}$ is a set of concrete transfer-instances in a system.
2) $\mathbf{r}$ is a set of bits accessible to a programmer.
3) Function $u : (\mathbf{t} \times \mathbf{r}) \rightarrow \{n, s, e\}$, where n, s, and e, respectively, represent no use, shared use, and exclusive use. Notation "$u(t, r) = n/s/e$" means, respectively, that transfer t will not use, share, or exclusively use bit r.

Then, a scenario can be formally defined.

Definition 5: Given a flat TRG $G = (\mathbf{t}, \mathbf{r}, u)$, a scenario is a subset $\mathbf{s}$ of $\mathbf{t}$ satisfying the following: 1) $|\mathbf{s}| = 1$, or 2) $|\mathbf{s}| \geq 2$ and for any two distinct t_i, $t_j \in \mathbf{s}$, for all $r \in \mathbf{r}$, $(u(t_i, r), u(t_j, r)) \notin \{(s, e), (e, s), (e, e)\}$.

However, implementing a flat TRG is impractical due to the huge number of concrete transfers and bit resources in a system. In order to visualize a TRG and to practically generate scenarios, we use a different TRG definition based on transfer types and master, register, and memory-range resource models.

Definition 6: A TRG is $G = (\mathbf{T}, \mathbf{R}, U)$, where the following are defined.

1) $\mathbf{T}$ is a set of transfer types in a system; each transfer type is a set of transfer instances.
2) $\mathbf{R}$ is a set of logical resources; each resource is a set of bits.
3) Function U: $(\mathbf{T} \times \mathbf{R}) \rightarrow \{n, s, e\}$. For each pair $(T, R) \in (\mathbf{T} \times \mathbf{R})$, if all instances of T exclusively use all bits in R, then $U(T, R) = e$; if all instances of T do not use any bits in R, then $U(T, R) = n$; otherwise, $U(T, R) = s$.

Fig. 4 shows an abridged TRG for the Nios SoC. Arrows represent the transfer types, the blocks represent the resources, and the letter e and s represent the access mode. Note that some ISRs are treated as master resources.

D. Implement TRG for Test Generation

We implement TRG as a couple $(\mathbf{T}, \mathbf{R})$, where members in $\mathbf{T}$ and $\mathbf{R}$ are all intelligent objects that are aware of resource usage. A transfer type T has a resource-allocation operation. This allocation is an important part of the parameterization operation $T.P(\cdot)$ of T, and it is denoted as $T.P.A(\cdot)$. The allocated exclusive and total resource usages are denoted, respectively, as $T.\mathbf{U_e}$ and $T.\mathbf{U_t}$, where $T.\mathbf{U_e} \subseteq T.\mathbf{U_t} \subseteq \mathbf{R}$.

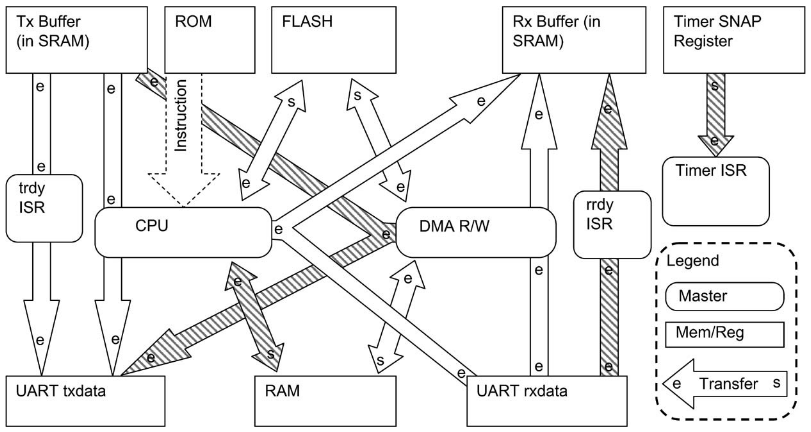


Fig. 4. Simplified TRG of the Nios SoC. The shaded transfers form a scenario.

Before we give a scenario generation algorithm, we need to introduce another internal operation of transfer type T. Once $T.P(\cdot)$ has decided on the concrete parameter values, the test generator needs to interpret them into actual configuration/invocation instructions. This interpretation operation is denoted as $T.I(\cdot)$. Its input comes from the output of $T.P(\cdot)$, and its output is SoC instructions that implement the configuration/invocation.

Given a TRG $G = (\mathbf{T}, \mathbf{R})$, let $\mathbf{R_S}$ and $\mathbf{R_E}$ represent, respectively, the current resources available for shared and exclusive accesses. The following algorithm constructs a scenario and maximizes the number of transfers.

1) $\mathbf{R_S} = \mathbf{R}$; $\mathbf{R_E} = \mathbf{R}$.
2) Randomly select a transfer type T_x from $\mathbf{T}$.
3) Issue $T_x.P(\cdot)$, which, in turn, issues $T_x.P.A(\cdot)$ to parameterize/allocate resources to T_x so that
   a) $T_x.\mathbf{U_e} \subseteq \mathbf{R_E}$.
   b) $(T_x.\mathbf{U_t} \setminus T_x.\mathbf{U_e}) \subseteq \mathbf{R_S}$.
4) Issue $T_x.I(\cdot)$ to interpret the configuration and output the configuration/invocation instructions.
5) $\mathbf{R_S} = \mathbf{R_S} \setminus T_x.\mathbf{U_e}$; $\mathbf{R_E} = \mathbf{R_E} \setminus T_x.\mathbf{U_t}$.
6) In $\mathbf{T}$, drop any transfer type that cannot obtain sufficient resources from the reduced $\mathbf{R_E}$ or $\mathbf{R_S}$.
7) If $\mathbf{T}$ is empty, one scenario with maximal transfers has been generated; otherwise, repeat from step 2).

The four shaded transfers in Fig. 4 form a legal test scenario. Although they appear to be loosely distributed in the TRG, the test quality is high because all HW components are supposed to concurrently behave in simulation, where the CPU is sorting data in RAM, the DMA is transferring data from a buffer to the UART, the Timer is counting, and the UART is working in full duplex mode (besides all these transfers, the instruction flow is also active. The instruction flow is not as manageable as data flows; thus, we treat it as "noise"). Therefore, a high degree of resource contentions will be achieved on various physical resources such as the bus, the slave interfaces, the interrupt mechanisms, and CPU time.

In our implementation, users can also intervene with the test generation by specifying a bias file, which biases most randomization operations in the test generator, including:

1) the behavior of transfer-type selection, i.e., step 2) in the aforementioned algorithm;
2) the behavior of transfer-type parameterization, i.e., $T.P(\cdot)$;
3) other control variables called environment parameters, which globally affect concurrent transfers (e.g., the UART Baud rate and the data/instruction cache mode).

The bias file will also be used in the test generation with feedback information from postsimulation analysis. Section VII-D provides further information.

E. Features and Limitations

As a model at high abstraction level, TRG has the following features/limitations.

1) TRG decouples two levels of complexity for test generation—the complexity of each transfer type and the complexity of generating parallelism. The former is system-specific, whereas the latter is relatively independent from an actual SoC, which makes TRG applicable to a wide range of designs.
2) Most effort is required in modeling each transfer type [e.g., manually composing $T.P(\cdot)$]; the task of generating a legal scenario is left to the test generator.
3) TRG is a method that is independent from the simulation/emulation platform. HW components are allowed to be modeled at different abstraction levels. It is even possible to apply TRG to generating manufacturing tests.
4) The target bugs are not the bugs inside each HW component, but hard-to-detect bugs caused by close resource competitions. Therefore, HW components are preferably free of obvious internal bugs.
5) Result checking is by means of checking the contents in memory and registers, which can be implemented as high-level functional checkers in test benches. However, a failed transfer gives limited indication of the physical location of the bug. Therefore, other error-detection mechanisms (e.g., assertions) should also be implemented in test benches.

Owing to TRG's high level of abstraction, SoC designers are allowed to plan test cases of parallelism and resource contentions long before a system is actually integrated.

V. TP Structure

A. Overview

We identify three roles that SW plays at system-level verification.

1) **Role 1**: Some SW components, i.e., ISRs, should cooperate with raw HW devices to fulfill their expected functionalities. This role extends a system from a collection of raw HW to a collection of usable functionalities.
2) **Role 2**: Some SW components should stimulate HW to check if HW works 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.
3) **Role 3**: Some SW should manage system-level concurrency by efficiently scheduling HW and SW behaviors.

These roles contribute differently to system-level verification. Role 1) is actually a part of design-under verification (DUV), Role 2) represents some actual tests to DUV, and Role 3) manages test cases on DUV. Role 3) serves as the backbone of the verification SW. It enhances the test quality by arranging parallelism on DUV and is relatively independent from an actual SoC. We now specifically regard the SW playing this role as the TP. TP generation should be automated, implying that TP should be regularly structured.

Our differentiation between these roles is not *ad hoc*. The components of Roles 1), 2), and 3), respectively, resemble the SW components running on a general-purpose computer, i.e., 1) HW drivers, which fulfill HW functionalities, 2) user processes, which carry out the user-defined tasks, and 3) operating system (OS), which schedules user processes. These two sets of components (ISR/Testcase/TP and driver/user process/OS) have different purposes and work on different levels. However, there are also similarities between them.

Computer users always hope that their user processes occupy most CPU time and the OS kernel consumes just a little fraction of CPU time as the overhead to maintain interprocess parallelism. Similarly, from the HW-verification point of view, TP [Role 3)] is only the control overhead to manage the user-defined tests; the SoC processor should distribute the most time executing the payload code—the code of Roles 1) and 2). Therefore, the structure of TP becomes extremely critical, considering that it directly influences simulation efficiency. Although all simulation-based verification methods suffer the same intrinsic shortcoming—simulation consumes a lot of time—we can alleviate the problem using an efficient TP structure. This alleviation is orthogonal to other efforts such as various simulation accelerating technologies. We present three versions of TP structures based on TRG.

B. Polling-Based TP

Our first structure is the polling-based TP [7]. The test generator identifies legal scenarios in the TRG, then it outputs transfers' configuration and invocation instructions in the TP. Consecutive scenarios are separated by polling statements to avoid resource conflicts. In simulation, these statements keep polling until all transfers in the current scenario have finished,

```
void main(){
......
/* Scenario x: Transfer T1 and T2 */
/* Configure T1 and T2: */
The configuration instructions of T1;
The configuration instructions of T2;
/* Invoke T1 and T2: */
T1_Running=True; The invocation
instruction of T1;
T2_Running=True; The invocation
instruction of T2;
/* Poll T1 and T2's notification: */
while (T1_Running OR T2_Running)
do_nothing;
/* Scenario x+1: */
......
```

Fig. 5. Pseudo code of a scenario in a polling-based TP.

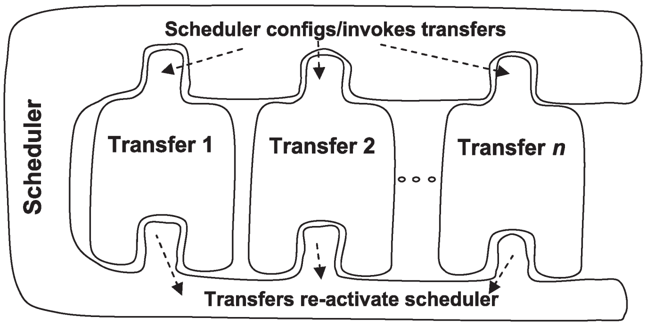

Scheduler configs/invokes transfers
Scheduler
Transfer 1
Transfer 2
Transfer n
Transfers re-activate scheduler

Fig. 6. Scheduler and transfers.

and then, the TP can proceed to the next scenario. The execution of the polling-based TP is shown in Fig. 7(a). Fig. 5 is a TP fragment, which submits one scenario made up of transfers T_1 and T_2.

C. Event-Driven TP

The second structure is the event-driven TP also called scheduler, in which polling statements are canceled [8]. Fig. 6 conceptually shows the relation between some transfers (shown as the jigsaw pieces) and the scheduler. The scheduler invokes some transfers and then exits. In turn, transfers can reactivate the scheduler at their completion events; again, the scheduler may submit new transfers because some resources must have been released by the completed transfer.

In this scheme, the test generator does not predetermine the scenarios in the TP. Instead, the generator generates a collection of transfers, detects their resource conflicts according to the TRG, and encodes these conflicts as submission conditions of each transfer. The transfers' configuration and invocation instructions and their submission conditions are stored in an "action table." In the simulation, whenever a transfer is finished, the scheduler is invoked. The scheduler will access the action table to check (not poll) a waiting transfer's submission conditions (i.e., whether any resource-conflicting transfer is already running) before submitting that transfer. The execution of the TP is shown in Fig. 7(b).

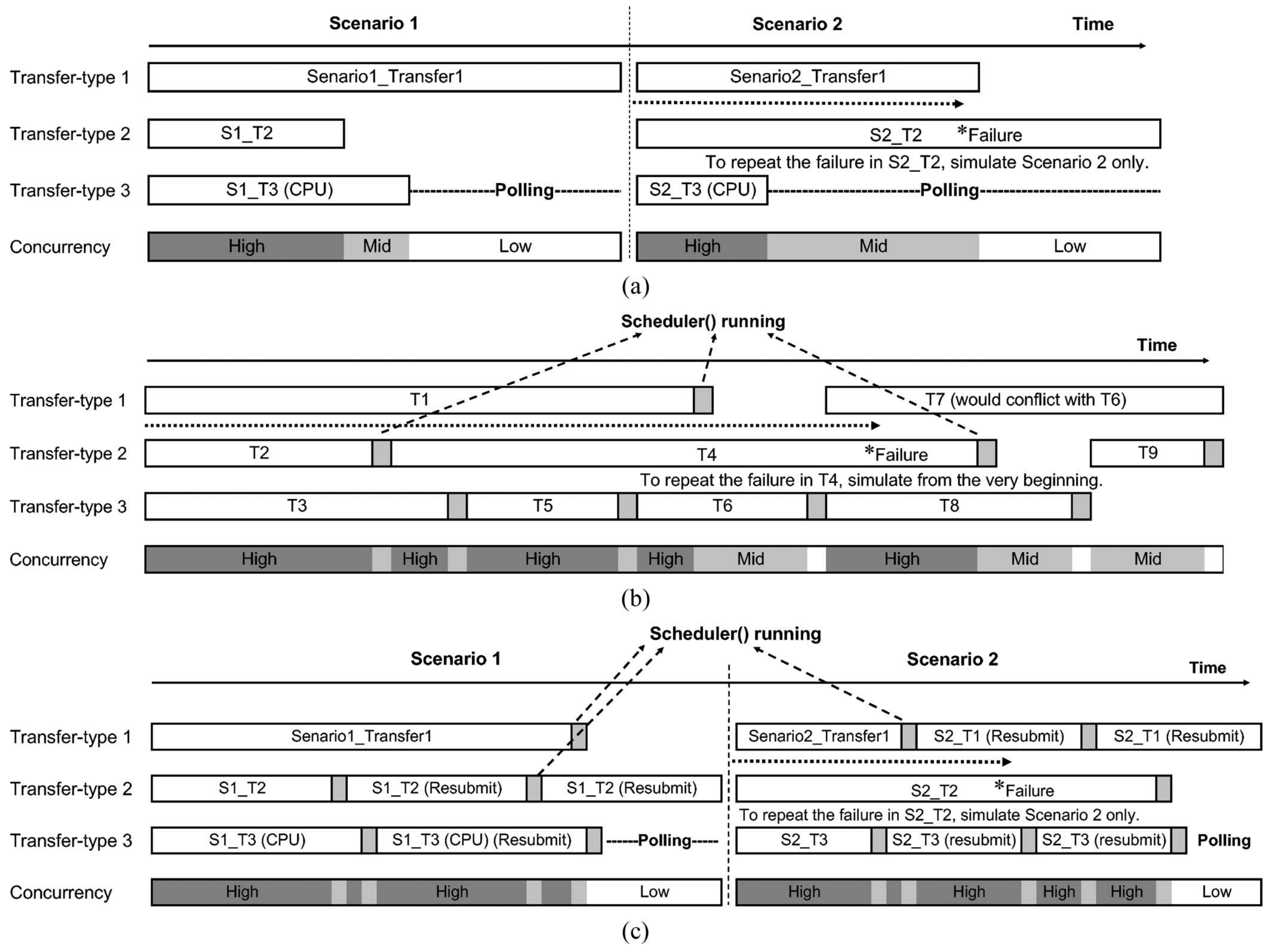


Fig. 7. Execution of TPs. In each subfigure, the first three rows represent three transfer types that can potentially run concurrently. The shading in the fourth row indicates the degree of concurrency. (a) Polling-based TP. Scenarios are separated by polling. (b) Event-driven TP. Scheduler attempts to submit new transfers when an old one finishes. (c) Hybrid-mode TP. Scheduler resubmits transfers in a scenario.

Compared with the polling-based program, the event-driven TP is more advantageous because it avoids inefficient polling statements so that the CPU could devote more time to stimulating HW. Meanwhile, the degree of concurrency will be enhanced, considering that the scheduler may submit new transfers as soon as an old one finishes. Moreover, the event-driven TP requires a robust interrupt mechanism. In this sense, we shall no longer simply view TP as the management overhead; instead, the TP directly plays a value-added part in HW verification.

The event-driven scheme does have a shortcoming in some cases. The simulation-time scenario is not predetermined at the generation time so that whenever we want to repeat a failure due to resource contentions among some specific transfers, we have to rerun the whole simulation from the very beginning, even if the failure occurs near the end of the simulation. The polling-based TP does not always have such a problem [compare Fig. 7(a) and (b)], considering that scenarios are explicitly written in the polling-based TP—when a failure happens in a scenario, we can comment out all the irrelevant scenarios in the TP to directly repeat the failed scenario.

D. Hybrid TP

To overcome the above shortcoming, we mix the two structures into a hybrid scheme. The test generator still predetermines scenarios, and the TP runs each scenario in an event-driven manner. That is, whenever a transfer in one scenario has finished, the scheduler is invoked and simply resubmits the finished transfer. This process is repeated until a certain condition is met (e.g., each transfer in one scenario has completed at least once). The TP execution is shown in Fig. 7(c). The polling mechanism is reserved but only works at the end of a scenario.

Resubmitting a finished transfer is necessary because more temporal relations among the concurrent transfers can be traversed. At the logical level, temporal relations specify the order of logical events experienced by the concurrent transfers, such as which transfer starts first or finishes first. At the physical level, temporal relations even have finer granularity (up to a single clock cycle). Rare temporal relations imply high-quality tests. For example, simultaneous bus accesses and nesting interrupts are relatively rare relations; however, they are excellent circumstances to verify whether the HW and SW can behave correctly.

Another advantage of the hybrid TP is that the scheduler has less overhead than its counterpart in the event-driven scheme. This time, the scheduler does not have to check resource conflicts when it resubmits one transfer because the current scenario has already been identified as conflict-free in TRG.

VI. TRG for Coverage

A. Overview

The simulation-based verification of a complex VLSI, such as SoC, requires multiple coverage models. Each model measures the simulation effectiveness from a specific perspective. At the system level, considering that the system's behaviors can be described as concurrent interactions, one coverage model is needed to enumerate all concurrent interactions and the temporal relations between them. However, the widely used statement-based coverages (line, toggle, conditional and local state machine, etc) cannot give such information.

The temporal relations open up an enormous coverage space, requiring a mathematical model to deal with it. We choose Petri net [11], [12] as the model because its semantics describe concurrency which is constrained by resources.

B. TRG and Petri Net

TRG and Petri net share some similarities in describing a system. Both formally define concurrency and conflict.

The TRG model does allow us to specify the system-level concurrency. However, TRG lacks the capability to describe the dynamics of the system. As a high-level test-generation tool, TRG cannot, and does not need to, deterministically specify temporal relations between concurrent transfers. The rich possibilities of the temporal relations can only be realized during simulation. For example, TRG does not (and cannot) specify at which moment in the life of transfer T_1 that another running transfer T_2 will finish. The timing that T_2 finishes is a complex function of its configuration, its submission timing, and the contentions on resources between T_1 and T_2.

A scenario generated from TRG only represents a snapshot of data flows in a system. In contrast, the execution of a Petri net captures the temporal aspect of a system's behavior at the logical level; the reachability graph derived from a Petri net can be used to describe the possible execution sequences. Therefore, a Petri-net model is suitable for postsimulation analysis of the temporal aspects of a system. A desirable feature of TRG is that it can be readily converted to a Petri net. Assuming that any transfer in TRG contributes two transitions in Petri net, which are start and end, we can construct a Petri net from a TRG by the following steps.

1) **Converting Resources.** For each resource R in TRG, create a place P_R to represent the resource.
2) **Converting Transfers.** For each transfer type T in TRG, perform the following.
    a) Create two transitions T_{start} and T_{end}.
    b) Create a state-place $T_{running}$ (cf. resource-place P_R).
    c) Create flows of weight 1 from T_{start} to $T_{running}$ and from $T_{running}$ to T_{end}.
3) **Connecting Transfers and Resources**
    a) First, for each transfer-resource pair (T, R) that satisfies $U(T, R) = s$, the following are performed.
        i) Add one token into P_R.
        ii) Create one flow of weight 1 from P_R to T_{start}.
        iii) Create one flow of weight 1 from T_{end} to P_R.
    b) Then, for each transfer-resource pair (T, R) that satisfies $U(T, R) = e$, the following are performed.
        i) If P_R has no token, put one token in it.
        ii) Create one flow of weight $n(R)$ from P_R to T_{start}, where $n(R)$ is the number of tokens in P_R.
        iii) Create one flow of weight $n(R)$ from T_{end} to P_R.

Once the Petri net is generated, its reachability graph is obtainable from a Petri-net tool. Fig. 8 shows a Petri net constructed from the TRG of the Nios SoC.

It should be noted that both the TRG and the Petri net converted from TRG are high level abstractions of a SoC (with its application). Most resource contentions at the physical level are invisible, which is simply because the physical resources are not present in the models. Nevertheless, the Petri net can provide useful temporal information regarding HW–HW and HW–SW interactions at the granularity level, as discussed in Section III-E. The Petri net could include even more temporal information, provided that each transfer contributes more internal states and events other than simple "start," "running," and "finish."

C. Use of Petri Net

The derived Petri net can indicate the total number of (unparameterized) scenarios (see Section IV-C) because each state in the reachability graph represents a scenario. This number contributes to the total complexity of scenario-generation algorithm in Section IV-D.

The most practical use of the Petri net is to define the coverage space. The coverage space is based on the reachability graph associated with the net. There are several options to define the space:

1) All states in the graph (i.e., markings);
2) All state-state transitions in the graph;
3) All paths in the graph;
4) All cycles in the graph.

These options represent the different levels of temporal details. In [12], a number of coverage-space definitions based on the reachability graph are proposed. For example, the path coverage space could be just too enormous due to the graph size and connectivity; however, some modifications can be made, such as limiting the length of the path.

To check the coverage, we need to collect the transfers' start/finish event history from the simulation trace. This history can be easily collected because each transfer has an SW flag indicating whether it is running. The Petri net reads the event history to replay the transition firing sequence. Its reachability graph is traversed in this manner. The traversed states, transitions, and other coverage points (cycles/paths) are counted and compared with the coverage space size, then the percentages are reported.

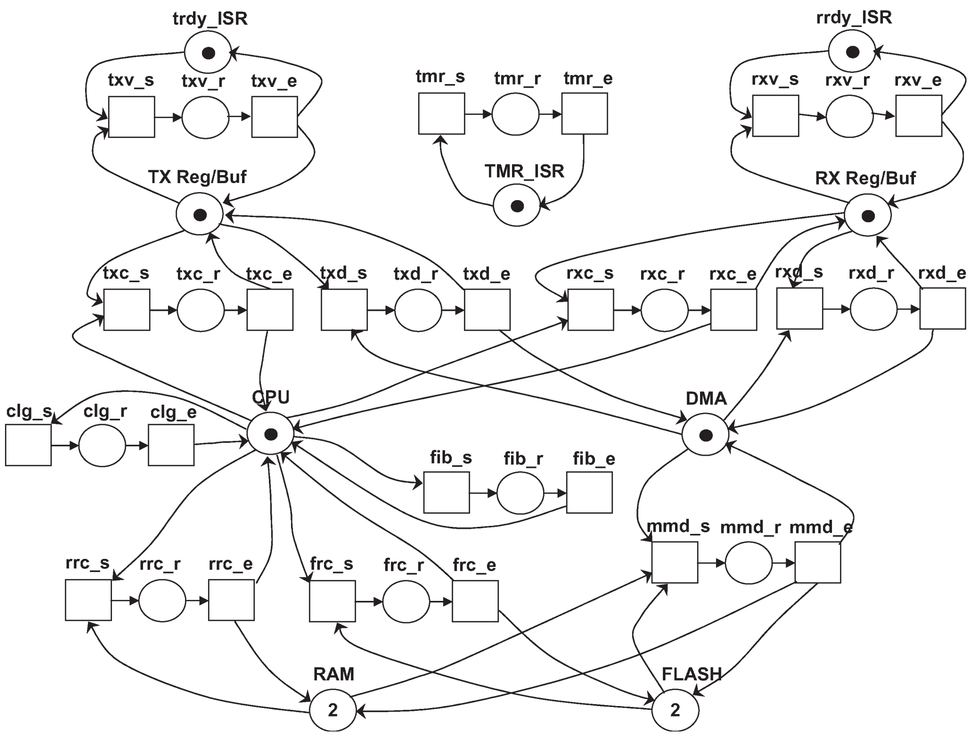
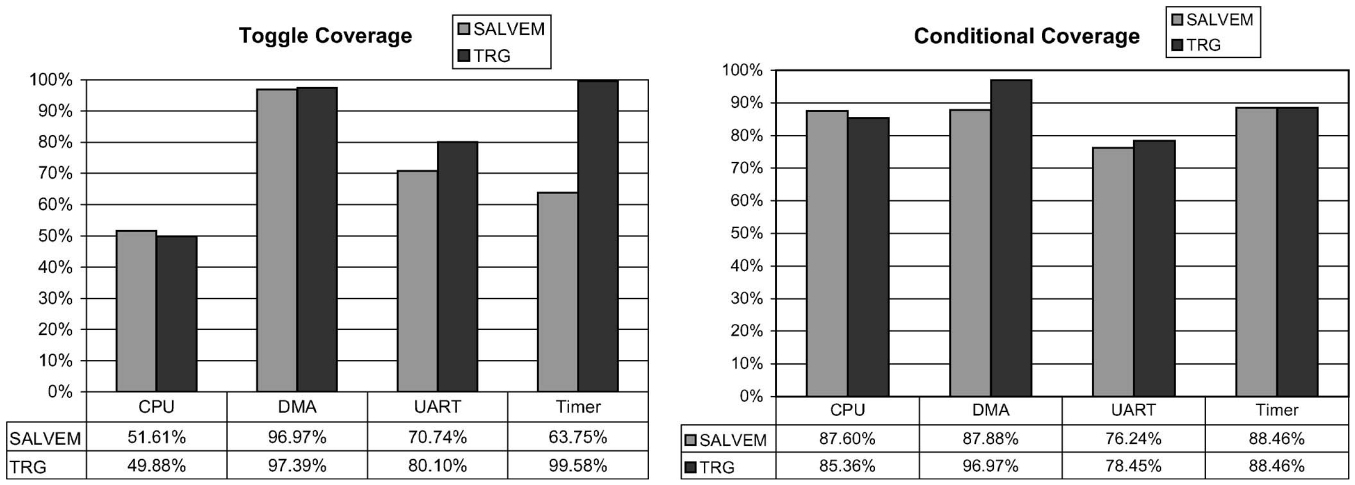


Fig. 8. Petri net derived from the TRG of the Nios SoC. Square nodes are transitions, round nodes are places, and dots and numbers are tokens.



Toggle Coverage

	CPU	DMA	UART	Timer
SALVEM	51.61%	96.97%	70.74%	63.75%
TRG	49.88%	97.39%	80.10%	99.58%

Conditional Coverage

	CPU	DMA	UART	Timer
▨ SALVEM	87.60%	87.88%	76.24%	88.46%
■ TRG	85.36%	96.97%	78.45%	88.46%

Fig. 9. Comparison of toggle and conditional coverages.

Besides indicating the completeness of temporal relations, the coverage information can be further used to guide test generation. We have implemented test generation with feedback at state and transition level (see Section VII-D for the details).

VII. Experimental Results

A. *Statement Coverages*

We have observed that reasonable statement coverages can be achieved by TPs generated from TRG. Considering that another SW-based test-generation methodology called SALVEM [13] is demonstrated on the same Nios SoC, we compare the results of the SALVEM tests with the test results of TRG. Fig. 9 shows the comparisons of the statement (toggle and conditional) coverages.

The TRG method has higher coverages on some components but is lower (but comparable) on the CPU, which has 11 000 lines of code and is the most complex component in the system. The lower coverages on CPU using the TRG method may be attributed to the fact that we have not put too much effort in manually creating subroutines stressing the processor itself, which, we believe, need another level of automation beyond the scope of this paper. We believe that the TRG method imposes no restrictions on achieving reasonably high statement-based coverages.

B. *State Space Traversing*

The statement-based coverage measures give little information regarding system-level concurrency and resource

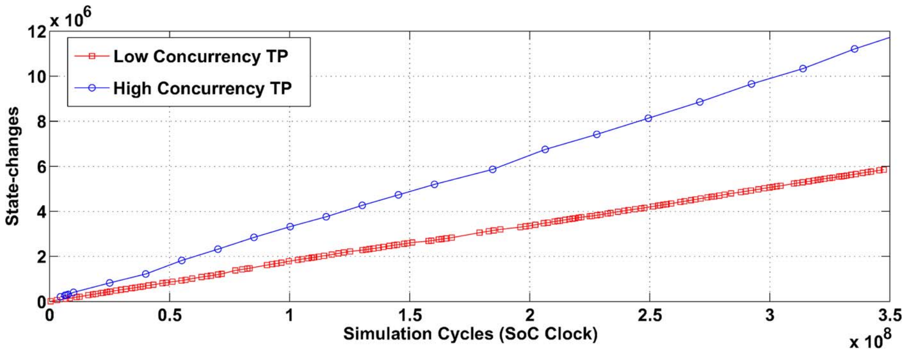


Fig. 10. State changes against simulation cycles.

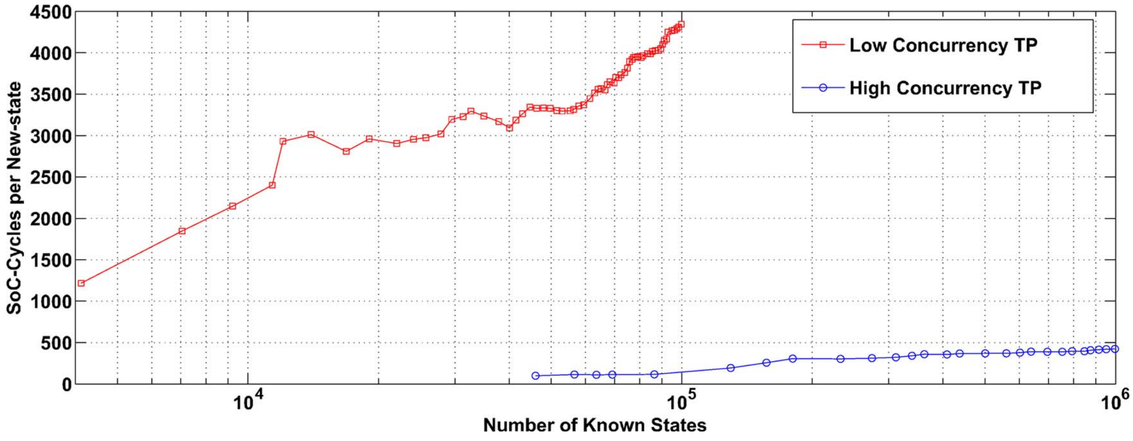


Fig. 11. New-state emergence rate against the number of known states.

contention. Therefore, we attempt to indicate this information using "system state space." We define the state space as the space made by the concatenation of the major control/status registers in the SoC components (CPU, DMA, UART, and Timer). The concatenation is 64-b long; thus, theoretically, the size of the space is 2^{64}, which makes it impractical for any tests to traverse it exhaustively. However, we can statistically measure how fast states can change and how fast new (i.e., unprecedented) states will emerge. These values are useful, considering that system states can give information regarding concurrency. For example, from the traversed states, we can tell if all peripherals have simultaneously requested interrupt.

We compare the capabilities to traverse the state space between two sets of TPs. One set contains TPs of scenarios of one or two transfers, and the other set contains TPs of scenarios of a maximum number of (i.e., three, four, or five) transfers. Fig. 10 shows the rate of state change. The high-concurrency TPs have a state-change rate that is roughly two times the rate of the low-concurrency TPs. A faster state-change rate implies that more events are happening simultaneously.

However, a faster state-change rate does not necessarily mean efficient state-space traversing. This is because states may recur many times. We further compare how fast unprecedented states emerge in simulation. Our experiments show that low-concurrency TPs have traversed about 10^5 distinct states in 420 million SoC clocks (12 computing hours on a 3-plus-GHz workstation). In comparison, high-concurrency TPs can traverse 10^6 distinct states in the same simulation duration. In Fig. 11, each data point represents one simulation of a TP. We observe that high-concurrency TPs produce new states at a much faster speed and that the speed is more insensitive to the number of known states. This encouraging comparison implies that concurrency is the key to efficiently exploring the state space.

C. *TP Efficiency*

We further compare different TP structures' performances in terms of execution time. Regarding the execution of the TPs as the overhead to build scenarios and the execution of other handwritten SW components as the payload, we profile the execution time of the three TP structures by monitoring the program counter (PC) in the Nios CPU. The profiling requires two tasks.

1) A monitor module is inserted in the test bench to record the calling/interrupt/return events by comparing PC with the addresses in the symbol table generated by the TP compiler.
2) A postsimulation analyzer is developed to extract various information from the record, including the average execution time of each function (with/without callings and interrupts), interrupt distribution among functions, and nested interrupts.

Fig. 12 shows the proportion of the TP execution time versus the TP size in terms of transfer quantity. We can see that the polling-based TP wastes a high percentage of the simulation

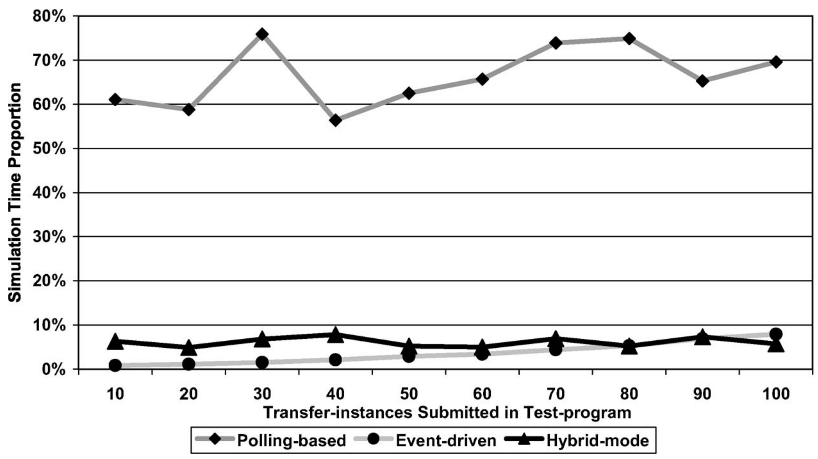


Fig. 12. TP profiling comparison.

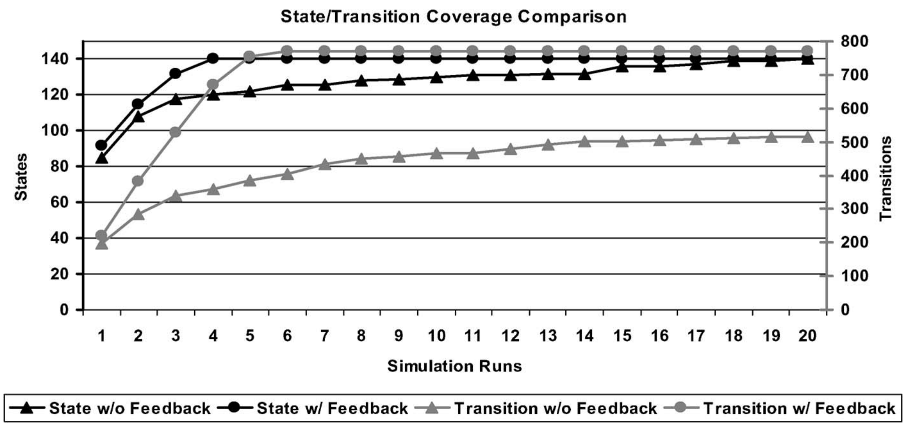


Fig. 13. State coverage and transition coverage with and without feedback.

time executing the polling statements but that the percentage is independent from the number of transfers in the TPs; the event-driven TP consumes substantially less time. However, as the transfer number in TPs grows, there exists more resource dependence between the transfers. As a result, the TP (scheduler) has to consume more time checking submission conditions. That explains the increasing execution time. For the hybrid-mode TP, both event-driven and polling mechanisms are used so that the percentage is comparable to the pure event-driven case but is marginally higher. Once again, the percentage is still independent from the transfer number because no submission condition is needed. Hence, the hybrid-mode TP is the most efficient among the three TPs when the transfer number is large. In addition, because of its advantages for debugging, we extensively use the hybrid-mode TP in this paper.

D. Test Generation With Feedback

We model a TRG with 12 major transfer types for the Nios SoC. Our generator can exhaustively (but randomly) produce 139 transfer subsets to be the (unparameterized) scenarios. This is well predicted by the reachability graph, which has 140 states, with the additional state representing the empty scenario. The reachability graph also contains 772 transitions.

We have achieved a test generation with feedback at state and transition levels.

A simulation-trace analyzer is developed. The analyzer is responsible for the following tasks:

1) count states and transitions in the trace log file;
2) compare the counts with the total states and transitions in the reachability graph;
3) identify the target (i.e., uncovered or less frequent) states/transitions;
4) in a bias file (see Section IV-D), adjust the randomization arguments about transfer selection and parameterization.

The state-level feedback is straightforward because a state in the reachability graph simply represents a scenario in the TRG. Once a target scenario is identified in the bias file, we simply increase the selection weights of the transfer types which make up the target scenario. Thus, the test generator will be more likely to generate the target scenario.

The transition-level feedback requires additional consideration. A transition in the reachability graph is a transfer-start T_s or a transfer-end T_e event, which separates two scenarios S_1 and S_2, i.e., $S_1 \xrightarrow{T_s} S_2$ or $S_1 \xrightarrow{T_e} S_2$. Thus, the analyzer needs to manage both the target scenario (S_1 or S_2) and the target event (T_s or T_e).

First, we identify the target scenario.

1) In case of $S_1 \xrightarrow{T_s} S_2$, the target scenario is S_2.
2) In case of $S_1 \xrightarrow{T_e} S_2$, the target scenario is S_1.

Once the target scenario is identified, we can apply the same mechanism as that used for state-level feedback in order to make it more possible for the target scenario to happen.

Second, we need to make the target event happen earlier in the current scenario (in order to enter or leave the target scenario, otherwise, the current scenario changes). For each transfer type, we define one of its parameters as its life expectancy, which controls how long a transfer will be running. For example, for a transfer-type "RAM-to-Flash DMA," the parameter "DMA length" is the life-expectancy parameter. The analyzer then adjusts the randomization ranges of the life-expectancy parameters in the bias file, where it reduces the life expectancy of T and/or extends the life expectancy of the rest transfers in the target scenario. Therefore, in simulation, the target event has more chance to fire earlier in order to enter or leave the target scenario.

Fig. 13 shows the accumulative state-coverage and transition-coverage comparisons between two sets of 20 simulation runs, where one is with feedback and the other is only with randomly generated scenarios (each set needs approximately 15 computing hours on a 3-plus-GHz 1-G RAM workstation). The figure shows that with feedback, all states and transitions are covered in the first several runs. For the 20 runs without feedback, the state-coverage space is traversed five times slower, and the transition coverage space cannot be traversed in 20 runs.

It should be noted that the fast traversing on states and transitions does not mean that the whole verification process is complete. If more detailed temporal relations (e.g., path/cycle) and other variations, such as transfer parameterization, are taken into account, more scenarios are needed. The fast traversing does give us a chance to focus on other coverage areas. Similar to all feedback techniques, our feedback scheme only targets at one type of coverage.

VIII. Concluding Remarks

TRG is a natural view of a system from the perspective of a device-level programmer, where a system is made up of programmer-controlled data flows, which are constrained by programmer-controlled resources. Therefore, TRG puts itself into a unique position between the SW and HW domains in verifying a system.

TRG has been successfully demonstrated on the single-processor Nios SoC. The basic idea of combining data flows with resource contentions is generic, making it applicable to a wide range of SoCs. In our future work, we will apply the model to verifying more sophisticated SoCs with multiple processors and multiple bus hierarchies. Another research area is the coverage model of parallelism and resource contention. Although the current Petri-net model derived from TRG can represent a certain level of resource-constrained parallelism, we may need to incorporate the domain knowledge about a real-world system to capture enough information regarding fine-grained resource competitions. One of the research directions to be taken is about how much domain knowledge is required to make a coverage model accurate and scalable.

References

[1] Cadence, *It Is About Time—Requirements for the Functional Verification of Nanometer-scale ICs*, 2005. Whitepaper.

[2] G. Mosensoson, "Practical approaches to SoC verification," in *Proc. DATE User Forum*, 2002. [Online]. Available: http://www.cecs.uci.edu/ics259/05-08-03/verisitySOCverify.pdf

[3] F. Hunsinger, S. Francois, and A. A. Jerraya, "Definition of a systematic method for the generation of software test programs allowing the functional verification of system on chip (SoC)," in *Proc. 4th Int. Workshop MTV*, 2003, pp. 11–16.

[4] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, "X-GEN: A random test-case generator for systems and SoCs," in *Proc. IEEE Int. HLDVT Workshop*, 2002, pp. 145–150.

[5] R. Emek and Y. Naveh, "Scheduling of transactions for system-level test-case generation," in *Proc. HLDVT*, 2003, pp. 149–154.

[6] Y. Kwon, Y. Kim, and C. Kyung, "Systematic functional coverage metric synthesis from hierarchical temporal event relation graph," in *Proc. 41st Annu. DAC*, San Diego, CA, 2004, pp. 45–48.

[7] J. Xu and C. C. Lim, "Modelling heterogeneous interactions in SoC verification," in *Proc. Int. Conf. VLSI-SoC*, Nice, France, 2006, pp. 98–103.

[8] J. Xu and C. C. Lim, "Exploiting concurrency in system-on-chip verification," in *Proc. IEEE APCCAS*, Singapore, Dec. 2006, pp. 836–839.

[9] *Nios Hardware Development Tutorial ver 1.2*, Altera Inc., San Jose, CA, 2004. [Online]. Available: http://www.altera.com/literature/tt/tt_nios_hw.pdf

[10] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[11] J. L. Peterson, *Petri-Net Theory and the Modeling of Systems*. Upper Saddle River, NJ: Prentice-Hall, 1981.

[12] H. Zhu and X. He, "A methodology of testing high-level Petri nets," *Inf. Softw. Technol.*, vol. 44, no. 8, pp. 473–489, Jun. 2002.

[13] A. Cheng, A. Parashkevov, and C. C. Lim, "Verifying system-on-chips at the software application level," in *Proc. IFIP-WG Conf. Very Large Scale Integr. Syst.-on-Chip*. Perth, Australia, Oct. 2005, pp. 586–591.

Xiaoxi Xu received the B.S. degree in automation from Shanghai Jiao-Tong University, Shanghai, China, in 1996. He is currently working toward the Ph.D. degree in the School of Electrical and Electronic Engineering, University of Adelaide, Adelaide, SA, Australia.

He had nine years of work experience, with Advantest and KLA-Tencor, in very large scale integration manufacturing and postsilicon test before he joined the Centre for High Performance Integrated Technologies and Systems, University of Adelaide, as a Research Student. His research interests include system-on-chip and processor design verification and testing and on-chip computation and communication architectures.

Cheng-Chew Lim (M'82–SM'02) received the B.Sc. degree in electronic and electrical engineering and the Ph.D. degree from Loughborough University, Leicestershire, U.K.

He is currently with the University of Adelaide, Adelaide, SA, Australia, as an Associate Professor and the Acting Head of the School of Electrical and Electronic Engineering. He is an Associate Editor of the *Journal of Industrial and Management Optimization*. His research interests include system-on-chip verification and very large scale integration architectures for array processors for matrix-based computations and wireless communications.