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

Xiaoxi Xu
The University of Adelaide
justinxu@eleceng.adelaide.edu.au

Cheng-Chew Lim
The University of Adelaide
cclim@eleceng.adelaide.edu.au

***Abstract*— The verification of a 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 automatic generation of test-cases for testing concurrent and resource-competing behaviors. We introduce the use of 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 the measurement of completeness of concurrency in simulation.**

I. Introduction

The system-on-chip (SoC) solution – designing a whole system ready for application in a single chip – has become a popular VLSI design paradigm. In addition to many advantages including lower production cost and higher performance, this design paradigm practically has 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 integration. Hardware components verified to be compliant with RTL (register-transfer-level) specification do not guarantee that they will work together as expected. To deal with this problem, an SoC must be verified hierarchically. That is, each component needs to be verified individually, then each sub-system made up of closely related components is verified, and finally the whole system is verified. Each verification stage has its own emphasis and goal. At sub-system 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 buried deeply in interactions between components, but *not* the bugs local to one component.

Besides hardware components, software also contributes to the full SoC functionality. Therefore checking hardware-software interactions becomes an important part of the SoC verification process.

Checking hardware-to-hardware and hardware-to-software interactions *separately* is still insufficient to fully verify a whole system, because a system capable of running heterogeneous forms of interactions *concurrently* is subject to various unforeseeable implementation bugs.

Listed in [3], unique bugs at system-level include:

- Interactions between blocks that are assumed verified;
- Conflicts in accessing shared resources;
- Arbitration problems and dead locks;
- Priority conflicts in exception handling;
- Unexpected hardware/software sequences.

All these bugs are related to interactions especially 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: (a) a method to generate test-cases of concurrency, and (b) 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 above two issues. Test-cases of concurrency can be generated from TRG. Furthermore, TRG can be transformed into Petri-net to allow the temporal relations of concurrency to be quantified. In this sense, TRG serves as the model for both test-generation and coverage, saving the time and effort to establish a generation model and a coverage model separately.

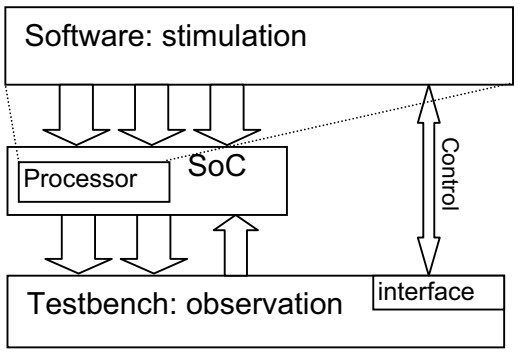


Fig. 1. Relations among an SoC, its test-bench and the test-software.

We adopt the software approach [2] of generating test-cases in the form of software (test-programs) to be executed by the SoC processor. The relations among an SoC, its test-bench and the software are conceptually shown in Figure 1. The software bears the responsibilities of stimulating the SoC and maintaining 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 software.) Therefore, by the software approach, a considerable amount of time and effort could be saved in test-bench development.

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

The rest of the paper is organized as follows. Section II introduces some related works, focusing on the relations between 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 test-program structures supported by TRG. Section VI discusses how TRG is transformed to Petri-net as the post-simulation stage coverage model. Experimental results in Section VII illustrates some quantitative aspects of the TRG method. Section VIII concludes the 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 an SoC. Using randomization is the conventional way to produce stimulation to a design. This method has intrinsic problems in test quality, whereas complex designs like 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 into account the fundamental characteristic of a system, namely, the concurrency.

Model-based generation is more likely to produce high quality tests of concurrency. XGEN [6], [7], the test-generator used at IBM for system verification, models *interactions* as the building blocks for test-software generation. An *interaction* is a series of communication stages known as "acts"; each act is performed by some hardware 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, hardware components with their functionalities also need to be modeled in detail, which requires in-depth hardware knowledge.

Coverage measures indicate verification completeness. At 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. In [8], Kwon *et al.* propose that users first establish a hierarchical-temporal-event-relation (HiTER) graph to represent the interactions between communicating hardware 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 accurate coverage space. To build such a graph, the users (verification engineers) must have an accurate view of signal-level timing dependencies between components.

Test-generation and coverage measures can also work very closely with each other. This approach usually applies a common model to test-generation and coverage, therefore, strong expertise is indispensable in the modeling process. Geist *et al.* show their experience in coverage-directed test-generation [9]. For a given design, an FSM (finite-state-machine) in the form of BDD (binary-decision-diagram) is manually derived from its specification and implementation. The coverage space is then defined as the transitions of the FSM. For each transition, a fake assertion that this transition is not reachable is forced and then the assertion is evaluated by a model-checking tool, which yields a counter-example to that fake assertion. The counter-example will be eventually developed into a test-case to activate the target transition. This approach ensures that all defined coverage tasks are covered. But a considerably large amount of manual preparatory work is needed in the upfront modeling stage.

We realize that in verifying complex design like SoC, the requirement of in-depth knowledge about hardware implementation is usually the bottleneck for building a consistent system model, which is crucial for improving test quality or for accurately computing coverage space. Our proposed TRG model (firstly introduced in [11]) addresses this issue at a higher level of abstraction. Compared with the "interaction" model in XGEN, our model called "transfer" is more concise and atomic. Furthermore, no hardware 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 concurrency [12]. But TRG has not been formally defined, and its application for coverage is not yet explored. This paper is to formally define TRG and introduce its usage in defining coverage spaces.

We demonstrate our methodology on a Nios SoC [4] as shown in Figure 2. This SoC is simple ($\sim 25,000$ lines of Verilog code) but contains adequate features of a typical system: *multiple* components (including the CPU) interconnected by on-chip bus. The Nios CPU is a pipelined RISC, the DMA can perform data transfer between any slaves on the Avalon bus, the full-duplex capable UART is the communication interface of the system, ROM and SRAM store instructions and data respectively, and 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 software's control via an interface implemented in the RAM.

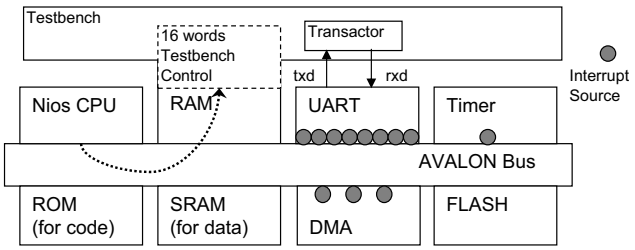


Fig. 2. System under demonstration – the Nios SoC.

III. Transfer Model

A. Overview

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

At system-level, the focus of verification shifts from checking each hardware component to checking the *interactions* between components. Therefore, 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 software known as test-program (TP), which has little controllability and observability of signal-level events (e.g., Bus_Request and Bus_Acknowledge, etc). However, the abstraction level should not be too high because TP is supposed to *closely* stress the SoC hardware. It is inappropriate for a TP to view devices as API *services*.

To trade-off the above considerations, the model should be readily comprehended by a device-level programmer, who understands the hardware functionalities and performances, but may have little knowledge about hardware implementation. We use the term *transfer* to represent interaction at this *specific* abstraction level. In the following subsections we will discuss the concept of transfers and their characteristics.

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:

(i) A Flash-to-RAM DMA transfer. It is a series of read/write operation driven by the dedicated hardware – the DMA engine;
(ii) The execution of a `sort` subroutine. It can be viewed as a pattern of memory-access performed by the CPU, driven by the execution of software;
(iii) 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 DMA engine, `sort` subroutine and interrupt subsystem/interrupt handler. While checking *each* of them is commonsense, checking their *parallel* execution will greatly improve test quality, because we are able to observe not only interactions, but also *interference between interactions*. If the above 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 UART interrupt will be nested in DMA interrupt.

The key to effectively construct 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:

(i) 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.)
(ii) Invocation: Transfers can be invoked by some instructions. Invocation instructions are allowed to have side-effects of configuration.
(iii) Notification: Events of transfer completion can be notified to software in some way (e.g., via interrupt), so that some software flags can indicate these events.

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

Note that configuration/invocation/notification are the *overhead* of a transfer and that the main body of a transfer is the data-flow. Figure 3 shows the life-cycle of a transfer, which includes a data-phase and a control-phase. Transfers embody the interactions that the design allows and the application requires, therefore, transfers are simple test-cases in their own right.

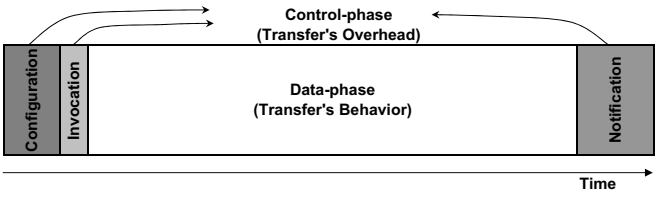


Fig. 3. Transfer life-cycle.

C. Expression Power of Transfer

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

(i) Transfer-type "Flash-to-RAM-DMA":
- Configuration: *initial-source-address, initial-destination-address*, *width* (8, 16, or 32-bit) and *length*;

- Invocation: set DMA engine control register `go` bit;
- Notification: DMA finish interrupt.

(ii) Transfer-type "Sorting":
- Configuration: *address*, *data type*(signed/unsigned integer, etc), *length*, *sort-algorithm*, *reverse*;
- invocation: call subroutine `sort(address, type, length, algorithm, reverse);`
- Notification: the return of the subroutine.

(iii) Transfer-type "UART-Rx-by-Interrupt":
- 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 test-bench/test-program interface; when this address is written, the test-bench starts to feed the SoC with a bit stream.
- 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:

(i) **Hardware Behaviors (Hard-transfers):**
The read/write operations on the bus driven by master-devices, whose behaviors are mostly hardwired. So they are categorized as *hard-transfers*.

(ii) **Software Behaviors (Soft-transfers)**:
A processor in SoC is a valid master device, whose behaviors are programmable rather than hardwired. So 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 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.
One guideline to build soft-transfers is to compose read/write intensive subroutines to stimulate the interactions between CPU and slaves. However, soft-transfers do not have to transfer data *literally*; they can be computation-intensive operations to apply stress to different types of physical resources. For example, a deep recursive subroutine can apply stress to the register-window mechanism in Nios CPU architecture. There may be some useful methodologies for automatic generation of special-purposed soft-transfers, such as the randomized instruction sequences for the purpose of processor self-testing. However, this level of automation is beyond the scope of this paper.

(iii) **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, *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 work "concurrently" 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, while the read/write 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 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. Because 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 quite flexible in the sense that defining a transfer-type allows for trade-off between the number of transfer-types in a system and the complexity of their $P(\cdot)$s'. To one extreme, we could model only one single transfer-type to represent all possible interaction patterns in a system, but its $P(\cdot)$ needs to deal with a very large but also very 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, while their $P(\cdot)$s' all have trivial complexity. In other words, given an SoC, the more generalized each transfer-type is, the fewer transfer-types are required, but at the cost of more complex $P(\cdot)$s'.

In practice, it is natural to adopt this strategy: generalizing interaction patterns with similar parameterization style (including resource allocation style) as one transfer-type. Taking the example of the Nios SoC, we initially planned

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

TABLE I

Implement different categories of transfers.

to model 12 transfer-types to represent DMA transactions among four source memory modules (ROM, RAM, FLASH, SRAM) and three destination memory modules (RAM, FLASH, SRAM). But later on 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. Whereas, 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-expectancy 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. In our Nios SoC, transfer-type "ROM-to-RAM-DMA" has the rate of 33.3MB/sec; while the transfer-type "UART-Rx-by-interrupt" is operating at 14.4KB/sec (or 115200bps baud-rate). Now 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-byte-long transfer T_1 at the speed of 10MB/sec runs alongside another 1000-byte-long transfer T_2 at 10KB/sec? It appears that this is a poor match, since T_1's life is only one thousandth of T_2's, meaning that the parallelism exists only 0.1% of the simulation, so the competition on the shared resource (the bus) is very little. Therefore, it makes sense to configure all transfers to have comparable life-expectancies, say, 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 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 should 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/notification, is the overhead. So 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 simulation time will be spent on the overhead. Fortunately, the length of control-phase is predictable because all transfer-types' configuration/invocation/notification are made up of instruction sequences of similar length. Hence, we assume that the following quantities are available:

- the average execution time of transfer configuration, C;
- the average execution time of transfer invocation, I;
- 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 $(10 \sim 100) \times (C + I + N)$, 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, so 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 $4500 \sim 45000$ SoC clocks. This estimation guides us on how to model transfer-types and especially on how to bias their $P(\cdot)$s' behaviors.

From the above 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 to understand the features and limitations of 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 Transfer-Resource-Graph

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. Even more intriguing situation is that competitions in various domains can interfere with each other, 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/notification, a test-program 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 resource-*conflicts*. Resource-contentions represent the physical level competitions that are supposed to be resolved by hardware mechanisms (e.g. bus protocol, 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 programmer's discretion to *avoid*. For example, we should allow the DMA engine to compete with the CPU for a physical memory module, but 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, ..., t_n, which are respectively instantiated from transfer-types $t_1.T$, $t_2.T$, ..., $t_n.T$, 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 t_i's configuration, 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 t_i's temporal relations (sequential, overlapping, etc) with all other transfers in $\mathbf{t}$, we say $\mathbf{t}$ 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; but 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, each having a *start* and an *end* event, then we shall allow for $\frac{(2n)!}{2^n}$ possible event sequences, all of which shall yield the same results in the memory/register space.

To avoid 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. A "golden" SoC model (instead of simple checkers) could help the automation of indeterministic result checking, but the problem is often that there is no such golden model of an *entire* SoC.

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 physical level are constructed implicitly.

B. Logical Resouces

Since resource-conflict is a logical concept, we only need to model the local resources in the system. Therefore, there is no need to model hardware'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.

(i) **Master:** Master is defined as anything that can conduct a transfer-type. Examples of master in our Nios SoC include the read-master and the write-master 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 interrupt-service-routine (ISR) that cooperates with hardware to perform data-intensive operations, e.g. the UART receiver-ready-ISR is a virtual-master performing transfer-type "UART-Rx-by-Interrupt". (Also see Section III-C.) A virtual master is usually capable of only one transfer-type, but we can model as many virtual-masters as necessary for an SoC, independent from the number of physical CPUs. Other examples in our Nios SoC include UART transmitter-ready-ISR and timer-ISR. Once modeled, the test-generator does not distinguish virtual and real masters. In this way, the resource-contention on CPU-time can be constructed implicitly.

(ii) **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.
Since control/status registers across an 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.

(iii) **Memory-range:** Memory-ranges are flexible resources dynamically maintained by the test-generator. A memory-range is an object with properties of base-address, size, sub-word granularity, R/W mode. From within one free memory-range, 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 even can overlap if they are all read-only. By 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. But this restriction can be lifted if we view the whole memory space as a single free memory-

range and allocate sub-ranges to transfers. In that case, the corner-cases in which a transfer crossing physical boundaries can be naturally built. However, this implementation needs to take account of miscellaneous constraints such as: ROM cannot be written; memory-mapped-registers should be excluded from the address space; different memory modules may accept different granularities, etc.

Just like that transfer-types are the generalization of similar transfer-instances (see Section III-D), the above discussed logical resource types (master/register/memory-range) are the generalization of the *bit-resources*, namely, all *bits* in memory and registers accessible by a programmer. *Bit* (regardless of data-, control- or status-bit) is the finest resource object to a programmer; 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 control/status 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 is a lot of (continuous) data bits.

C. Formal Definition of TRG and Scenario

Once transfers and resources are modeled, their relation becomes explicit: transfers need resources to run. They can be interlinked to form transfer-resource graph (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:

- $\mathbf{t}$ is a set of concrete transfer-instances in a system;
- $\mathbf{r}$ is a set of bits accessible to a programmer;
- function u: $(\mathbf{t} \times \mathbf{r}) \rightarrow \{n, s, e\}$, where n, s, and e respectively represents *no-use*, *shared-use* and *exclusive-use*. Notation "$u(t, r) = n/s/e$" respectively means 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:

- $|\mathbf{s}| = 1$, or
- $|\mathbf{s}| \geq 2$ 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 generate scenarios practically, we use a different TRG definition based on transfer-types and master/register/memory-range resource models.

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

- $\mathbf{T}$ is a set of transfer-types in a system; each transfer-type is a set of transfer-instances;
- $\mathbf{R}$ is a set of logical resources; each resource is a set of bits;
- 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, $U(T, R) = s$.

Figure 4 visualizes 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.

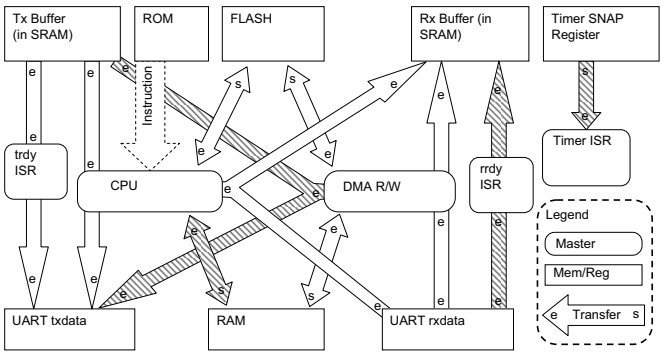


Fig. 4. Simplified TRG of the Nios SoC. The shaded transfers form a scenario.

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 aware of resource usage. A transfer-type T has a resource-allocation operation. This allocation is an important part of the T's parameterization operation $T.P(\cdot)$, and it is denoted as $T.P.A(\cdot)$. The allocated *exclusive* and *total* resource-usages are respectively denoted as $T.\mathbf{U_e}$ and $T.\mathbf{U_t}$, where $T.\mathbf{U_e} \subseteq T.\mathbf{U_t} \subseteq \mathbf{R}$. To construct a *parameterized* scenario, we need to search for any non-empty subset $\mathbf{S}$ of $\mathbf{T}$ and perform $T_i.P(\cdot)$ and $T_i.P.A(\cdot)$ of each T_i in $\mathbf{S}$, so that for any distinct T_j and T_k in $\mathbf{S}$, $T_j.\mathbf{U_e} \cap T_k.\mathbf{U_t} = \varnothing$ and $T_j.\mathbf{U_t} \cap T_k.\mathbf{U_e} = \varnothing$.

Before we give a scenario generation algorithm, we need to introduce another internal operation of transfer-type T. Once $T.P(\cdot)$ has decided 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 $T.P(\cdot)$'s output; and its output is SoC instructions that implement the configuration/invocation.

Given a TRG $G = (\mathbf{T}, \mathbf{R})$, let $\mathbf{R_S}$ and $\mathbf{R_E}$ respectively represent the current resources available for *shared* and *exclusive* access. 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:
   - $T_x.\mathbf{U_e} \subseteq \mathbf{R_E}$, and
   - $(T_x.\mathbf{U_t} \setminus T_x.\mathbf{U_e}) \subseteq \mathbf{R_S}$
(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 Figure 4 form a legal test scenario. Although they appear loosely distributed in the TRG, the test quality is high because all hardware components are supposed to behave concurrently in simulation: 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. Instruction flow is not as manageable as data-flows, we treat it as "noise".) Therefore, 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 the test-generation by specifying a bias file, which biases most randomization operations in the test-generator, including:

- the behavior of transfer-type selection, i.e., step (2) in the above algorithm;
- the behavior of transfer-type parameterization, i.e., $T.P(\cdot)$;
- 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 test-generation with feedback information from post-simulation analysis. Section VII-D provides further information.

E. Features and Limitations

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

- 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 while the latter is relatively independent from an actual SoC, which makes TRG applicable to a wide range of designs.
- Most effort is required in modeling each transfer-type (e.g. manually composing $T.P(\cdot)$); the task of generating legal scenario is left to the test-generator.
- TRG is a method independent from the simulation/emulation platform. Hardware components are allowed to be modeled at different abstraction levels. It is even possible to apply TRG to generating manufacturing tests.
- The target bugs are *not* the bugs inside each hardware component, but hard-to-detect bugs caused by close resource competitions. Therefore, hardware components are preferably free of obvious internal bugs.
- 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. Also, test-generation with TRG is decoupled from test-bench development, and does not require extensive hardware implementation knowledge.

We use TRG to identify concurrency in *test-generation stage*. But we have yet to answer two further questions:

- How to let a test-program explore the temporal relations between concurrent transfers *at simulation time*?
- How to quantify the completeness of temporal relations *after simulation*?

The next two sections will respectively explore these issues.

V. Test-program Structure

A. Overview

We identify three roles that software plays at system level verification:

- **Role 1**: some software components, i.e., interrupt service routines (ISR), should *cooperate* with raw hardware devices to fulfill their expected functionalities. This role extends a system from a collection of raw hardware to a collection of usable functionalities;
- **Role 2**: some software components 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 software should *manage* system-level concurrency by efficiently scheduling hardware and software behaviors.

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

Our differentiation between these roles is not *ad hoc*. Role 1, 2 and 3 components respectively resemble the software 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. These two sets of components (ISR/Testcase/TP and driver/user-process/OS) have different purposes and work on different levels. (For example, OS

manages inter-process parallelism, while TP should manage hardware-hardware and hardware-software concurrency.) 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 inter-process parallelism. Similarly, from the hardware-verification point of view, TP (Role 3) is only the control *overhead* to manage the user defined tests; SoC processor should distribute most time executing the *payload* code – the code of Role 1 and 2. Therefore, the structure of TP becomes extremely critical since 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 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 Test-program

Our first structure is the *polling-based* TP [11]. 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, and then the TP can proceed to the next scenario. The execution of the polling-based TP is illustrated in Figure 7(a). Figure 5 is a TP fragment, which submits one scenario made up of transfers T_1 and T_2.

```
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. The pseudo code of a scenario in a polling-based TP.

C. Event Driven Test-program

The second structure is the *event-driven* TP also called scheduler, in which polling statements are canceled [12]. Figure 6 conceptually visualizes 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 re-activate the scheduler at their completion events; again, the scheduler may submit new transfers since some resources must have been released by the completed transfer.

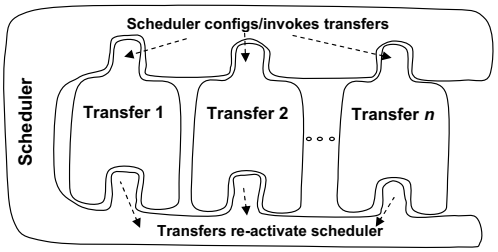


Fig. 6. Scheduler and transfers.

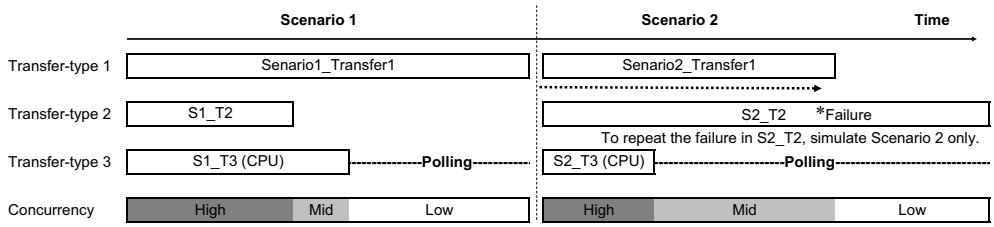
In this scheme, the test-generator does not *pre-determine* scenarios in the TP. Instead the generator generates a collection of transfers, detects their resource conflicts according to the TRG, and encodes these conflicts as *submission conditions* of each transfer. Transfers' configuration and invocation instructions and their submission conditions are stored in an "action table". In 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 Figure 7(b).

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

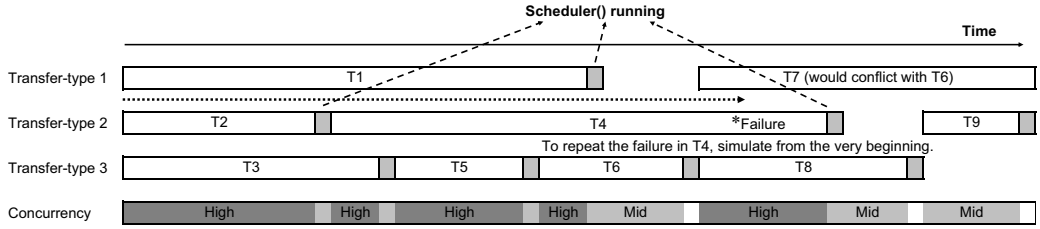
The event-driven scheme does have a shortcoming in some cases. The simulation-time scenario is not pre-determined at generation-time, so whenever we want to repeat a failure due to resource-contentions among some specific transfers, we have to re-run the whole simulation from the very beginning, even if the failure occurs near the end of the simulation. The polling-based TP does not always have such a problem (compare Figure 7(a) and 7(b)), since 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 Test-program

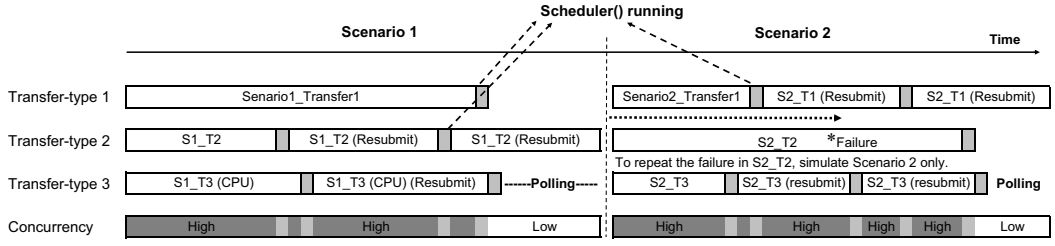
To overcome the above shortcoming, we mix the two structures into a hybrid scheme. The test generator still predetermines scenarios, and the TP runs each scenario in an event-driven manner. That is, whenever a transfer in one scenario has finished, the scheduler is invoked and simply re-submits the finished 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



(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.

Fig. 7. Execution of TPs. In each sub-figure, the first three rows represent three transfer-types that can potentially run concurrently. The shading in the fourth row indicates the degree of concurrency.

in Figure 7(c). The polling mechanism is reserved but only works at the end of a scenario.

Re-submitting 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 but they are excellent circumstances to verify whether the hardware and software can behave correctly.

Another advantage of the hybrid TP is that the scheduler has less overhead than its counterpart in the event-driven scheme. The scheduler now does not have to check resource-conflicts when it re-submits 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 like SoC requires *multiple* coverage models. Each model measures simulation effectiveness from a specific perspective. At system level, since 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 [13], [14] as the model because its semantics describe concurrency which is constrained by resources.

Definition 7: A *Petri-net* is a directed graph represented by a 5-tuple $(\mathbf{P}, \mathbf{T}, \mathbf{F}, W, M_0)$, where,

- $\mathbf{P}$ is a set of nodes known as *places*; each place can hold *tokens*. Tokens are all identical;
- $\mathbf{T}$ is a set of nodes known as *transitions*;
- $\mathbf{F}$ is a set of directed arcs (known as *flows*) connecting places and transitions, i.e. $\mathbf{F} \subseteq (\mathbf{P} \times \mathbf{T}) \bigcup (\mathbf{T} \times \mathbf{P})$;
- Function W: $\mathbf{F} \rightarrow \mathcal{N}^+$; $W(f)$ is called the *weight* of flow f; ($\mathcal{N}^+$ denotes positive integers.)
- Function M_0: $\mathbf{P} \rightarrow \mathcal{N}$, known as *initial marking*. $M_0(p)$ is the number of initial tokens in place p. ($\mathcal{N}$ denotes non-negative integers.)

A transition t is said to be *enabled* when each of its input

places has equal or more tokens than the weight of the input flow. When enabled, t can (but does not have to) *fire*, i.e., t consumes $W(f_i)$ tokens from its input place connected via flow f_i, and puts $W(f_o)$ tokens into its output place connected via flow f_o. The duration of a firing is considered zero. The firing sequence is called the *execution* of the net. The *state* of a Petri-net can be described in terms of its marking, i.e., the distribution of the tokens. A Petri-net has its *reachability graph*, whose nodes are the states (i.e. the markings) and whose directed arcs represent the transitions between states. The reachability graph can be used to define the coverage space.

B. TRG and Petri-net

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

- In TRG, concurrency is defined as a set of n ($n \geq 2$) transfers. In Petri-net, concurrency means a transition-node has *multiple* incoming or outgoing flows;
- In TRG, conflict means that some of concurrent transfers exclusively use the same logical resources. In Petri-net, conflict means a place-node has multiple incoming or outgoing flows.

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 transfer T_1's life, another running transfer T_2 will decease. The timing that T_2 deceases 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 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 post-simulation 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, *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,
   - create two transitions T_{start} and T_{end};
   - create a state-place $T_{running}$ (cf. resource-place P_R.);
   - create flows of weight 1 from T_{start} to $T_{running}$ and from $T_{running}$ to T_{end};
(3) **Connecting Transfers and Resources:**
   - First, for each transfer-resource pair (T, R) that satisfies $U(T, R) = s$:
     - add one token into P_R;
     - create one flow of weight 1 from P_R to T_{start};
     - create one flow of weight 1 from T_{end} to P_R.
   - Then, for each transfer-resource pair (T, R) that satisfies $U(T, R) = e$:
     - if P_R has no token, put one token in it;
     - create one flow of weight $n(R)$ from P_R to T_{start}, where $n(R)$ is the number of tokens in P_R);
     - create one flow of weight $n(R)$ from T_{end} to P_R.

The complexity of the above algorithm is linear to the size of TRG. Given a TRG $(\mathbf{T_{trg}}, \mathbf{R}, U)$, the sizes of the resulting Petri-net $(\mathbf{P}, \mathbf{T_{pn}}, \mathbf{F}, W, M_0)$ are:

- $|\mathbf{P}| = |\mathbf{T_{trg}}| + |\mathbf{R}|$,
- $|\mathbf{T_{pn}}| = 2|\mathbf{T_{trg}}|$, and
- $|\mathbf{F}| = 2|\mathbf{T_{trg}}| + 2|\{(T, R) : U(T, R) \in \{e, s\}\}|$.

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

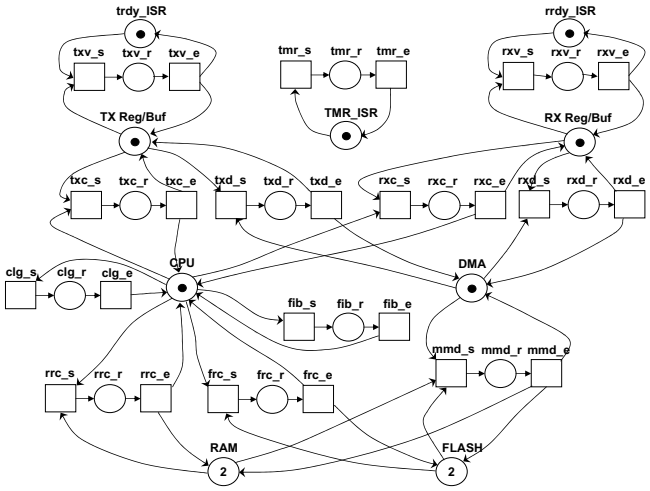


Fig. 8. Petri-net derived from the TRG of the Nios SoC. Square nodes are transitions, round nodes are places, and dots and numbers are tokens.

It should be noted that both TRG and the Petri-net converted from TRG are high level abstraction of an SoC (with its application). Most resource-contentions at physical level are *invisible*, simply because the physical resources are not present in the models. Nevertheless, the Petri-net can provide useful temporal information regarding hardware-hardware and hardware-software interactions at the granularity level 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

Once a Petri-net is obtained from TRG, we can use the net in a number of ways. For instance, we could prove the

liveness and boundedness of the Petri-net and then infer the similar characteristics of the TRG; we can simplify the Petri-net (but keep the reachability graph isomorphic), then we are able to map the simplification back onto the TRG. However, these topics are beyond the scope of this paper.

The derived Petri-net can indicate the total number of of (unparameterized) scenarios (see Section IV-C), because each state in the reachability graph represents a scenario. This size 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:

- All states in the graph (i.e. markings);
- All state-state transitions in the graph (these transitions are different from the transitions in the Petri-net);
- All paths in the graph;
- All cycles in the graph.

These options represent the different levels of temporal details. In [14], a number of coverage-space definitions based on the reachability graph are proposed. These definitions roughly fall into: (1) state-based category, (2) transition-based category, and (3) flow-based category. For example, the path coverage space could be just too enormous due to the graph size and connectivity, but some modifications can be made such as limiting the length of the path.

To check the coverage, we need to collect transfers' start/finish event history from the simulation trace. This history can be easily collected, because each transfer has a software flag indicating whether it is running. The Petri-net reads the event history to re-play 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.

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. Experiment Results

A. Statement Coverages

We have observed that reasonable statement coverages can be achieved by test-programs generated from TRG. Since another software-based test generation methodology called SALVEM [5] is demonstrated on the same Nios SoC, we compare the results of SALVEM tests with the test results of TRG. Figure 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 TRG

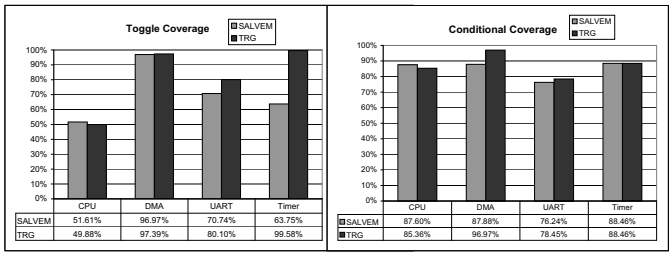


Fig. 9. Toggle and conditional coverage comparison.

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. (See explanation in Section III-C.) 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-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-bit long, so 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 since system states can give information regarding concurrency. For example, from the traversed states, we can tell if *all* peripherals have requested interrupt *simultaneously*.

We compare the capabilities to traverse 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 maximum number of (i.e., three, four or five) transfers. Figure 10 shows the rate of *state-change*. The high-concurrency TPs has a state-change rate roughly two times the rate of the low-concurrency TPs. Faster state-change rate implies that more events are happening simultaneously.

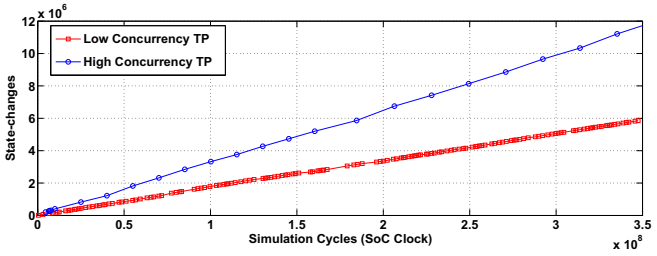


Fig. 10. State-changes against simulation cycles.

However, faster state-change rate does not necessarily mean efficient state-space traversing. This is because that

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+GHz workstation); in comparison, high-concurrency TPs can traverse 10^6 distinct states in the same simulation duration. In Figure 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 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.

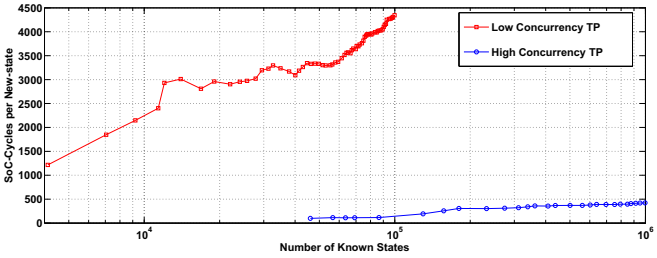


Fig. 11. New-state emergence rate against the number of known-states.

C. Test-program Efficiency

We further compare different TP structures' performance in terms of execution time. Regarding the execution of the TPs as the *overhead* to build scenarios and the execution of other hand-written software 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:

- 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.
- A post-simulation analyzer is developed to extract various information from the record, including average execution time of each function (with/without callings and interrupts), interrupt distribution among functions, nested interrupts.

Figure 12 plots 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 time executing the polling statements, but 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 dependency 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 the percentage is comparable to the pure event-driven case but is marginally higher. Once again, the percentage is still independent from

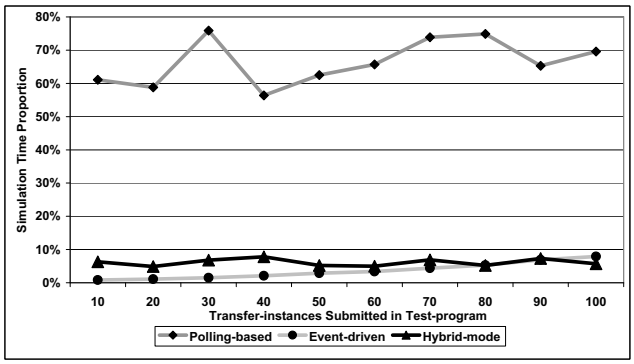


Fig. 12. TP profiling comparison.

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. Also because of its advantages for debugging, we use the hybrid mode TP extensively in our research.

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 test-generation with feedback at state-level and transition-level.

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

- Count states and transitions in the trace log file;
- Compare the counts with the total states and transitions in the reachability graph;
- Identify the target (i.e. uncovered or less frequent) states/transitions;
- In a bias file (see Section IV-D), adjust the randomization arguments about transfer-selection and transfer 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 event T_s or a transfer-end event T_e, 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 target scenario (S_1 or S_2) and target event (T_s or T_e).

Firstly, we identify the target scenario:

- In case of $S_1 \xrightarrow{T_s} S_2$, the target scenario is S_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 the target scenario more likely to happen.

Secondly, 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: 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 to enter or leave the target scenario.

Figure 13 includes the accumulative state-coverage and transition-coverage comparisons between two sets of 20 simulation-runs, one with feedback and the other only with scenarios generated randomly. (Each set needs approximately 15 computing hours on a 3G+Hz 1G 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, state-coverage space is traversed 5 times slower, and the transition coverage space cannot be traversed in 20 runs.

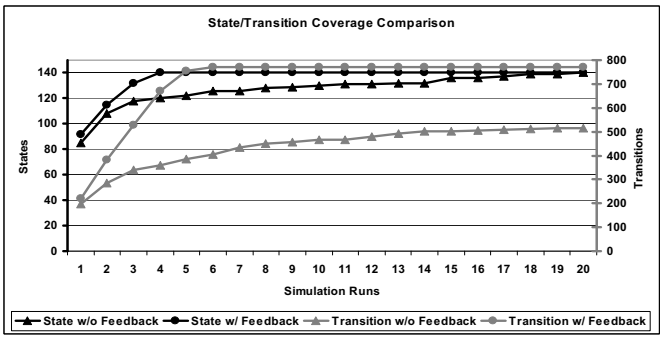


Fig. 13. State coverage and transition coverage with and without feedback.

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. Like 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: 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 software domain and hardware domain 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. While the current Petri-net model derived from TRG can represent 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 as well.

References

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

[2] F.Hunsinger, S.Francois, A.A.Jerraya, *Definition of a systematic method for the generation of software test-programs allowing the functional verification of System On Chip(SoC)*, 4th International Workshop on Microprocessor Test and Verification (MTV'03)

[3] 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`

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

[5] 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, Oct, 2005

[6] 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*, IEEE International High Level Design Validation and Test Workshop, 2002, (HLDVT02)

[7] R.Emek, Y.Naveh, *Scheduling of Transactions for System Level Test-Case Generation*, HLDVT03.

[8] Y. Kwon, Y. Kim, C.Kyung, *Systematic functional coverage metric synthesis from hierarchical temporal event relation graph*, in Proceedings of the 41st Annual Conference on Design Automation, San Diego, CA, USA, 2004 (DAC 2004)

[9] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y. Wolfsthal, *Coverage-Directed Test Generation Using Symbolic Techniques*, in Proceedings of the First International Conference on Formal Methods in Computer-Aided Design, 1996 (FMCAD96)

[10] K. Keutzer, S. Malik, R. Newton, J. Rabaey, A. Sangiovanni-Vincentelli, *System Level Design: Orthogonalization of Concerns and Platform-Based Design*, IEEE Trans on Computer-Aided Design, Dec 2000

[11] 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, 2006 (VLSI-SoC 2006)

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

[13] J.L.Peterson, *Petri-net theory and the modeling of systems*, Prentice Hall, 1981,ISBN-13: 978-0136619833

[14] H. Zhu, X. He, *A methodology of testing hight-level Petri-nets*, Information and Software Technology, 44(2002), p473-489