1

Transfer-Resource Graph and Petri-net for System-on-Chip Verification

Xiaoxi Xu and Cheng-Chew Lim
The University of Adelaide[*]
Australia

1. Introduction and Background

Verification of integrated circuits is an inherently difficult problem and the popular system-on-chip (SoC) design paradigm has brought about additional challenges. This section discusses these difficulties and challenges and gives an outline of our concurrency oriented solution to the SoC verification problem.

1.1 The VLSI Verification Problem

Verification of integrated circuit design is a process that checks the implementation of the design against its specification and identifies design bugs. The view that design *verification* is subservient to design *implementation* has soon become invalid when designs become just moderately complex. It has been claimed that the verification complexity is growing at a double-exponential rate (Saleh, 2004), i.e., exponential with respect to Moore's law. Consequently, nowadays, about 50%-80% of the design time and efforts are spent in very-large-scale-integration (VLSI) design verification. It becomes well known that verification is the biggest single bottleneck (Goering, 2008) in VLSI design.

There are two categories of verification methods.

- **Simulation-based methods**. In this category, the verification engineers develop a set of *tests* to stress a given design; the design is therefore often called the *design-under-test* or DUT. A test can be an abstract description about the control and observation applied upon the DUT. To actually apply the control and observation, a structure called test-bench (TB) needs to be constructed and simulated together with the DUT. The TB concretizes the abstract tests into "0/1" signals and directly interact with the DUT in these signals.
- **Formal methods**. In formal methods, verification engineers don't provide tests, but *design properties* – properties that a correct design should or should not have. Meanwhile, the design usually needs to be represented as a finite-state-machine (FSM) model. Then some model-checking tool computes whether the model abides by the properties. Formal methods form useful supplement to simulation in verifying control-intensive and FSM-based designs.

[*] This work was supported by the Australian Research Council under Linkage Project LP0454838.

Despite these approaches, verification of non-trivial design remains a major challenge due to its inherent difficulties.

• **The correctness issue**. Although a correct design specification is regarded as the foundation for verification, a real-world specification cannot be perfectly correct or complete. An SoC specification cannot foresee all corner-cases, especially when concurrency is concerned, and determine what behavior should be regarded as correct or incorrect.
• **The completeness issue**. For simulation, we have no absolute criteria to decide whether sufficient tests have been applied, because *passed tests do not indicate the absence of bugs*. Formal methods may exhaustively prove or disprove a given property, but the completeness problem turns into whether enough properties have been provided (Katz et al., 1999).

1.2 System-on-Chip (SoC) and Challenges for SoC Verification

The system-on-chip (SoC) solution – designing a whole system ready for application on a single chip – has become a popular VLSI design paradigm. Although there is no official definition of SoC, many would agree that a VLSI design that features *multiple components, including at least one processor, connected by on-chip interconnection* falls into the category of SoC. For example, the Nios SoC (Altera, 2004) in Figure 1 used in our research shows SoC features. The Nios CPU is a pipelined processor; the DMA can perform data transfer between any slaves on the bus; the full-duplex capable UART is the communication interface; ROM and SRAM store instructions and data respectively; RAM and FLASH are additional memories. The on-chip Avalon bus is actually a cross-bar interconnection made of a set of arbitrators, which communicate with each other. There are also a number of interrupt sources.

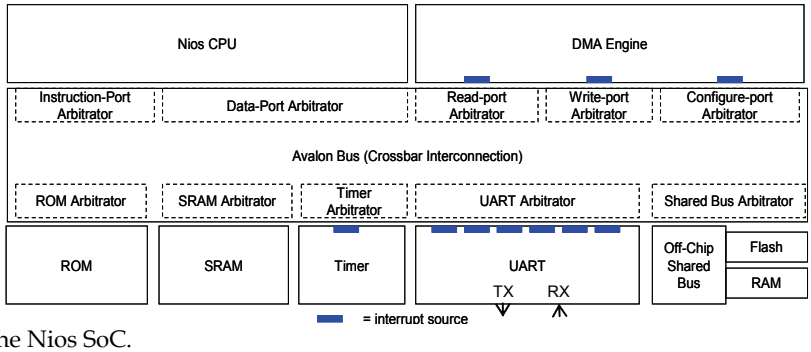

Fig. 1. The Nios SoC.

In addition to many advantages including higher performance, the SoC paradigm has practically reduced the process of designing a complex system into integrating some predesigned and reusable components. However, the verification of an SoC becomes even more difficult due to its features.
First, SoCs feature large scale of hardware (HW) integration. On one hand, the integration introduces *emerging properties* that are not present in any individual component and thus need *additional* control and observation effort; on the other hand, these components are not guaranteed to work together in all circumstances since they have been designed and verified separately under different sets of principles and assumptions.

Second, software (SW) running on the SoC processor(s) also contributes to the full SoC functionality, and makes an SoC behave much more intelligently than an FSM. Therefore checking hardware-software interactions becomes an important part of the system-level verification and introduces another new dimension to the verification problem.
Even if these two issues, i.e., HW-HW interaction and HW-SW interaction, are addressed *separately*, full system-level verification is still not achieved, because a system capable of running heterogeneous forms of interactions *concurrently* could be subject to various unforeseeable implementation bugs, including (Mosensoson, 2002):

- 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.

These bugs are all related to interactions especially to concurrent ones with resource competitions. *Concurrency*, together with the associated *resource competition*, is the central characteristic of a system; therefore constructing concurrency is the key to system-level verification. However, current verification practices are not dealing with this concurrency problem effectively.
Formal methods are simply not at the position to do system-level verification. They work best with moderately complex, single-component and FSM-based designs; while a typical SoC shares none of these features. Moreover, formal methods, which require the user to provide properties, face a fundamental dilemma in dealing with SoC verification. An SoC, which is capable of heterogeneous HW-HW and HW-SW interactions, has *unforeseeable* failure modes; therefore, the user cannot postulate those properties they are yet to know.
System-level verification has to substantially rely on simulation. However, existing simulation-based methodologies such as VMM (Bergeron et al., 2005) put substantial focus on test-bench (TB) construction. A TB tends to stimulates and observes a DUT from its *exterior*; so it has inherent issues in controlling and observing the concurrency *internal* to the DUT. Moreover, in the mainstream TB-centric methodologies, software's position and roles are not given sufficient consideration to be well integrated in the verification framework.

1.3 Outline of Concurrency-Focused Solution to SoC Verification

To deal with the SoC verification challenges, particularly concurrency, we present two ideas: **Software-Centric Verification** (Xu et al., 2008)**.** This means that the software native to the SoC, instead of the test-bench (TB), should take more proactive roles in verification. In the sequel, we call software also as "test-program" (TP). TP takes high level *control* roles, especially in *concurrency management*; while TB only takes relatively passive *observation* roles. Software-centric verification reshapes the traditional verification framework, which heavily relies on complex TB for all control and observation roles. Figure 2 shows the new framework. The rationale, implementation and advantage of software-centric verification are detailed in (Xu et al., 2008)**.**

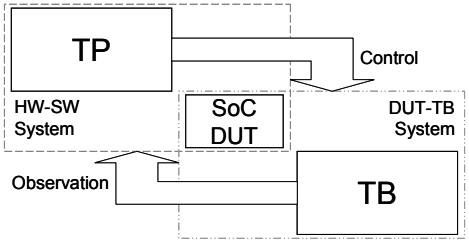

Fig. 2. Software-centric verification. The SoC and the test-bench (TB) form a DUT-TB system, which is under the active control of the test-program (TP); meanwhile, the SoC and the TP form a HW-SW system, which is under the passive observation of the TB.

Interaction-Oriented Verification (Xu & Lim, 2007). We also emphasize that objects-under-test are actually *interactions among components*, rather than the *components* themselves. Focusing on interactions (or communications) is already a common practice, particularly in transaction-level-modeling (TLM) (Cai & Gajski, 2003). TLM emphasizes the separation of components' *communication capabilities* with their *communication capabilities*. However, the verification community has not yet proceeded to migrate from the view in which interactions are treated as the properties, or "capabilities", *attached* to hardware components, to the view in which interactions themselves become a set of objects *independent* from hardware components.

Treating interactions as objects-under-test is philosophically attractive: the terms "tests" and "objects-under-test" are now referring to *one* category of entities – interaction-objects, which can naturally model "concurrency".

- As "tests", interaction-objects stress the communication mechanisms. Moreover, they can be combined to run *concurrently* to exercise the system more vigorously.
- As "objects-under-test", interactions bring their own properties to be tested, for instance, their temporal relations when they run *concurrently*.

The rest of this chapter handles some critical issues in our approach and they share the same theme – concurrency. Particularly, since *test-generation* and *coverage* are two fundamental aspects of simulation-based verification, we need to provide i) a method to generate test-cases of concurrency, and ii) a method to quantify the concurrency completeness. The first issue is addressed by a model called transfer-resource graph (TRG) and the second by a Petri-net derived from the TRG. We start our treatment by introducing the interaction model – *transfer*. A transfer is a *software-controllable interaction-object*, linking the ideas of software-centric and interaction-oriented verification.

2. The Transfer Model

This section discusses how to identify and characterize interactions, which appear to be abstract and shapeless, as valid objects that can be further combined to form concurrency.

2.1 Overview: Proper Abstraction Level

For system-level verification, interactions must be modeled at a proper abstraction level. Interactions in a system come in different levels, such as signal-level handshakes, logical

level frames/packets/tokens, and also application-level threads/processes. The abstraction level should not be too low, for we are to implement tests in software known as test-program (TP), which hardly have direct control and observation on signal-level events (e.g., Bus-Request and Bus-Acknowledge). However, the abstraction level should not be too high either, since after all a TP is supposed to vigorously stress hardware devices.
To trade-off the above considerations, the interaction model should be readily comprehended by a device level programmer, who understands hardware functionalities and performances, but may have little knowledge about hardware implementation. "Transfers" represent interaction-objects at this specific abstraction level.

2.2 Transfer Definition

While interactions need to be treated as objects-under-test, 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.

(i) A Flash-to-RAM DMA transfers. 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; this kind of interaction is driven by the execution of SW.
(iii) An incoming bit 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. While checking each of them is common sense, checking their concurrent execution will greatly improve the test quality – we are able to observe not only interactions but also *interferences 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 the UART interrupt would be nested in the DMA interrupt.
The key to effectively constructing such parallelism is to generalize heterogeneous interaction forms into a common model, which we call *transfer-type*.
Definition 1: A transfer-type is a set of software-controlled and data-intensive interaction patterns among SoC components. Its *software-controlled* feature means that a transfer-type has the following properties.

(i) **Configuration:** a transfer-type has its own parameters, which can be configured by some instructions. An important part of the configuration is the resources to be used.
(ii) **Invocation:** a configured transfer-type can be activated by some instructions. Invocation instructions are allowed to have side effects of configuration.
(iii) **Notification:** the completion of invoked transfer-type can be notified to software in some way (e.g., via interrupt), so that some software flag can indicate the event.

A closely related concept is the *instance* of a transfer-type called transfer-instance.
Definition 2: A transfer-instance is a transfer-type bounded with a specific configuration.
We may view a transfer-type as a set of transfer-instances. When the discrimination between these two concepts is insignificant or can be inferred, we use the term *transfer*.
Figure 3 shows the life-cycle of a transfer, which includes a data-phase and a control-phase. Note that the configuration, the invocation and the notification are the overhead of a transfer and that the main body of a transfer is the data-flow.

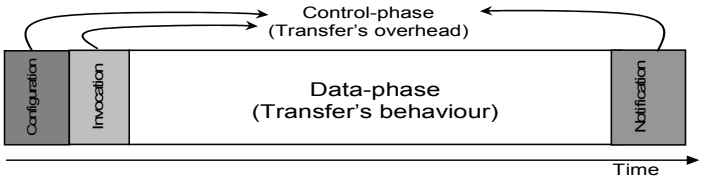

Fig. 3. The life cycle of the transfer model.

Notice that the differentiation between "control" and "data" is relative to the abstraction level. One control-operation at the transfer level, typically a register-write, is also a data-operation at instruction level; similarly, the data-flow of a transfer has already implicitly included physical-level control.

2.3 Transfer's Expression Power

In the early stage of SoC design/verification, the abstraction level should be high enough to hide the differences between hardware behaviors and software behaviors (Keutzer et al., 2000). Our transfer-type model meets this requirement. All three examples of interaction in Section 2.2 can be expressed as transfer-types.

(i) Transfer-type "Flash-to-RAM-DMA":

- **Configuration**: source-address, destination-address, transaction width/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, reverse;
- **Invocation**: call subroutine sort(address, type, length, 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 Rx-finish conditions.

More generally, the *transfer* model can model three categories of data-intensive interactions.

(i) **Hardware Behaviors (Hard-transfers).** The read/write operations on the bus are driven by master devices, whose behaviors are mostly hardwired. So these operations 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 important 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. 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 also 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 the Nios CPU architecture.

(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 in the ISR, from a higher level of abstraction, it is functionally equivalent to perceive that a virtual master (also see Section 3.2) is conducting the stream between the UART receiver and a memory buffer, *independently* 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 parallelism between the 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 may be secondary.

Table 1 lists the three transfer categories and summarizes how to implement their configuration, invocation and notification.

Category	Transfer Examples	Configuration	Invocation	Notification
Hard-tansfer	Any DMA transfer	Setting control-registers	Setting control-registers	Hardware interrupt
Soft-transfer	UART polling; sorting; recursive subroutines; processor self-testing	Setting control-registers; Passing arguments to subroutines	Calling subroutine	Subroutine's return
Virtual-transfer	UART trdy/rrdy ISRs; Timer time-out ISR	Setting control-registers; Setting global variables	Enabling interrupt sources	The virtual master (ISR) itself

Table 1. Implementation of hard-, soft- and virtual-transfers.

2.4 Transfer Complexity

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

One transfer-type's complexity is caused by its configuration. We use **T** to denote the set of transfer-types in a system, and denote T_i as each transfer-type member. For T_i, each of its parameter has a set of values to select from. T_i's parameter-space is very application-oriented because parameters could be either dependent of or coupled with one another. Therefore, T_i requires an operation $T_i.P(\cdot)$ – denoted from the object-oriented programming point of view – to perform its parameterization. The complexity of $T_i.P(\cdot)$ can represent the complexity of T_i. To let $T_i.P(\cdot)$ *deterministically* traverse its parameter-space seems neither necessary nor practical; therefore, we implement $T_i.P(\cdot)$ using weighted and constrained randomization.

Our transfer model is quite flexible in the sense that defining transfer-types allows for the 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 every possible interaction pattern of concrete parameters; then, 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(·)s.
In practice, it is natural to adopt this strategy: to generalize interaction patterns with *similar* parameterization style as one transfer-type. Taking the example of the Nios SoC, we initially modeled 12 transfer-types to represent DMA transactions among four source memory modules (ROM, RAM, FLASH, SRAM) and three destination memory modules (RAM, FLASH, SRAM). However, we have later decided to merge them into one transfer-type called "memory-to-memory DMA", with a single but stronger P(·) 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.

2.5 Transfer Temporal Granularity

To further characterize transfers, we give an estimation of their life-expectancy.
First, 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 a rate of 33.3MB/sec; while the transfer-type "UART-Rx-by-interrupt" is operating at 14.4KB/sec. 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-bytelong transfer T_2 at 10KB/sec? It appears to be a poor match, since T_1's life is only 1/1000 of T_2's, meaning that the parallelism and resource contention exist only 0.1% of the simulation. Therefore, it makes sense to configure all transfers to have *comparable* life-expectancies, say, within one order of magnitude of difference.
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 (i.e., its configuration, invocation and 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 appear to be 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 ~ 100)×(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 ~ 100) × (C + I + N), or 4500 ~ 45000 SoC clocks. This is also the temporal granularity of our proposed "system-level" tests. It guides us on how to model transfer-types, especially on how to bias their P(·)s' behaviors.
Now we have a quantitatively feel of the "abstraction level" of transfers to understand its relationship with other identifiable interaction-objects. *Transfers*, whose granularity is around 10^4 cycles, is formed by the aggregation of signal-level *transactions* and *instructions*, whose granularities typically range around 10^1 cycles; in turn, *transfers* themselves can be aggregated to become *processes*, whose granularity can range well beyond 10^6 cycles. Transfer's unique granularity helps us to understand its features and limitations.

- It appears impractical and also unnecessary to consider concurrent transfers' temporal relations at clock-level accuracy.
- It is reasonable to assume that concurrent transfers do *not* need to communicate each other (as processes do), and that transfers do *not* use resources dynamically.

3. Resource and Transfer-Resource Graph

3.1 Overview: Resource-Contentions and Resource-Conflicts

The focus of system-level verification is concurrency. The main purpose of constructing concurrency 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 2.2.
The strength of the transfer model is that it allows these heterogeneous competitions to be built naturally – we simply arrange multiple transfers to run concurrently. However, there should exist some principles to prevent unchecked or meaningless randomness. Our principle is to distinguish between resource-contentions and resource-conflicts. *Resource-contentions* represent 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. 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 transfers will not be predictable from their configurations.
Definition 3: Given a set **t** of transfer-instances 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

$$t_i.T.Check(t_i.configuration, MemRegSpace_{start}, MemRegSpace_{end}),$$

which, according to t_i's configuration, checks if the running of t_i has caused the expected changes 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 **t**, we say **t** is free of resource-conflicts (with respect to those Check() functions).

This definition requires the result of each t_i be *deterministically* predicted; but as a trade-off the temporal relations between conflict-free transfers are allowed to happen in an *indeterministic* manner; that is, if there are n conflict-free transfers, 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.

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(), 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 user. Once resource-conflicts are avoided, no other restrictions are preventing a test-generator from constructing parallelism. In this way, resource-contentions at physical level are constructed implicitly.

3.2 Logical Resources

Since resource-conflict is a logical concept, we only need to model the local resources in the system. 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.** A master is defined as any device 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's behaviors. A virtual-master (also see Section 2.3) 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". 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 and R/W mode. From within one free memory-range, test-generator can allocate sub-ranges of suitable size/location to some transfers; meanwhile, the unused fragments become free memory-ranges. The entire real memory space can be treated as the sole initial memory-range (if some minor constraints are resolved such as ROM cannot be written), so that an allocated sub-range can naturally cross boundaries between physical memory modules. Allocated memory-ranges can overlap if they are used as read-only. With these arrangements, the test-generator naturally constructs resource-contentions on physical memories.

Just like that transfer-types are the generalization of similar transfer-instances (see Section 2.4), 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. A bit, regardless of data-, control- or status-bit, is the finest logical resource object to a programmer; masters, registers and memory-ranges are simply different aggregations of bits. For instance, a physical master device's behavior is controlled and 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.

3.3 Definition of Transfer-Resource Graph

Transfers and resources interlink to form transfer-resource graph (TRG). TRG can be formally defined in terms of *transfer-instances* and *bit-resources*.

Definition 4: A flat TRG is a triple G = (**t**, **r**, *u*), where:

t is a set of concrete transfer-instances in a system;

r is a set of bits accessible to a programmer; and

function *u*: $\mathbf{t} \times \mathbf{r} \rightarrow \{\mathsf{n}, \mathsf{s}, \mathsf{e}\}$, where n, s, and e respectively represent *no-use*, *shared-use* and *exclusive-use*. Notation "$u(t, r) = \mathsf{n/s/e}$" respectively means that transfer t will not use, share or exclusively use bit r.

Then the term "concurrency" can be defined in terms of *scenarios*.

Definition 5: Given a flat TRG G = (**t**, **r**, *u*),

a) if $t, \tau \in \mathbf{t}$, we say that t and τ *conflict with each other* if
$$\exists\ r \in \mathbf{r},\ (u(t, r), u(\tau, r)) \in \{(\mathsf{s}, \mathsf{e}), (\mathsf{e}, \mathsf{s}), (\mathsf{e}, \mathsf{e})\}.$$
b) A *scenario* **s** is a subset of **t** satisfying
$$t \in \mathbf{s}, \tau \in \mathbf{s} \Rightarrow t \text{ and } \tau \text{ do not conflict with each other.}$$
c) A *maximal scenario* is a scenario $\mathbf{s_M}$ satisfying
$$\forall t \in \mathbf{t} \setminus \mathbf{s_M},\ \exists \tau \in \mathbf{s_M},\ t \text{ and } \tau \text{ conflict with each other.}$$

However, to implement 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 logical resource models (masters/registers/memory-ranges).

Definition 6: A TRG is G = (**T**, **R**, *U*), where

T is a set of transfer-types in a system; each transfer-type is a set of transfer-instances;

R is a set of logical resources; each resource is a set of bits; and

function *U*: $\mathbf{T} \times \mathbf{R} \rightarrow \{\mathsf{n}, \mathsf{s}, \mathsf{e}\}$. For each pair $(T, R) \in \mathbf{T} \times \mathbf{R}$, if all instances of T exclusively use all bits in R, then $U(T, R) = \mathsf{e}$; if no instances in T use any bit in R, then $U(T, R) = \mathsf{n}$; otherwise, $U(T, R) = \mathsf{s}$.

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

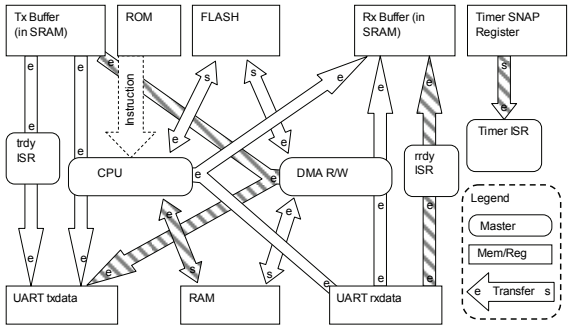

Fig. 4. The abridged transfer-resource graph for the Nios SoC. The shaded transfers form a scenario.

3.4 Implementing TRG for Scenario Generation

We implement TRG as a couple (**T**, **R**), where members in **T** and **R** are all intelligent objects aware of resource usage. A transfer-type T in **T** has a resource-allocation operation within its parameterization operation T.P(·); therefore it is denoted as T.P.A(·), which will set the allocated exclusive and total resource-usages respectively 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 subset **S** of **T** and perform $T_i.P(\cdot)$ and $T_i.P.A(\cdot)$ of each T_i in **S**, so that for any distinct T_j and T_k in **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(·) has decided the concrete parameter-values, the test-generator needs an interpretation operation, denoted as T.I(·), to interpret the parameter-values into actual configuration/invocation instructions.

Given a TRG G = (**T**, **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 maximal scenario.

(1) $\mathbf{R}_S = \mathbf{R}$; $\mathbf{R}_E = \mathbf{R}$;
(2) Randomly select a transfer-type T from **T**;
(3) Issue T.P(·), which in turn issues T.P.A(·), to parameterize/allocate resources to T (now T is actually a transfer-instance) so that $T.\mathbf{U}_e \subseteq \mathbf{R}_E$, and $(T.\mathbf{U}_t \setminus T.\mathbf{U}_e) \subseteq \mathbf{R}_S$;
(4) Issue T.I(·) and output the configuration/invocation instructions in a test-program;
(5) $\mathbf{R}_S = \mathbf{R}_S \setminus T.\mathbf{U}_e$; $\mathbf{R}_E = \mathbf{R}_E \setminus T.\mathbf{U}_t$;
(6) In **T**, drop any transfer-type that cannot obtain sufficient resources from the reduced $\mathbf{R}_E$ or $\mathbf{R}_S$.
(7) If **T** = ∅, one maximal scenario has been generated; otherwise repeat from step (2).

The four shaded transfers in Figure 4 form a maximal 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 busy sorting data in the RAM, and will be interrupted by all peripherals; the DMA is transferring the tx data from a buffer to the UART; the Timer is counting down; and the UART is working in full duplex mode. Meanwhile, the instruction flow also contributes to the concurrency. The instruction flow is not as manageable as data-flows, but can be viewed as the "noise" to transfers. (However, we can extract rich information from it about the quality of TP execution (Xu et al., 2008).)

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 intervene with 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(·);
- some control variables called environment parameters, which globally affect concurrent transfers, e.g. UART baud-rate and CPU data/instruction cache mode.

The bias file will also be used in test-generation with feedback information from Petri-net based post-simulation analysis. Section 5.3 provides further information.

3.5 Generation of Event-Driven Test-Program

Here we briefly introduce a scheme for step (4) in the above algorithm that allows the TP to be executed in an *event-driven* manner. This is illustrated in Figure 5.

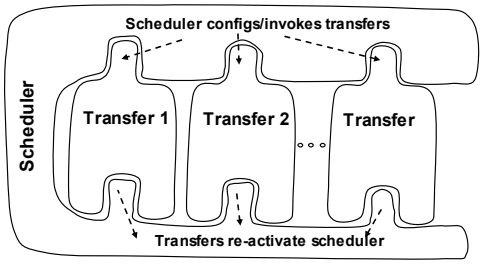

Fig.5. Event-driven test-program.

The parallelism-management part of the TP, shown as Scheduler in Figure 5, configures and invokes some concurrent transfers. Any of these transfers, when finished, re-activates the Scheduler (e.g., for hard- and virtual-transfers, call-back Scheduler; for soft-transfer, return to Scheduler). In turn, the Scheduler can submit or resubmit transfer(s) because some resource must have been released by the finished one. In this way, the execution of a TP is actually driven by transfers' notification events. The majority of the Scheduler can be generated automatically by step (4) in the scenario generation algorithm.
Under this scheme, transfers – software-controllable interaction-objects – are no longer *passive* building blocks subject to arrangement, but become *active* ones which can stress the system rigorously, reflecting the rationale of our software-centric and interaction-oriented verification methodology. The comparison between two event-driven implementations and a polling-based scheme can be found in (Xu & Lim, 2007).

3.6 Features and Limitations

As a model at high abstraction level, TRG has the following features and limitations:
• TRG decouples two levels of complexity for test-generation – the complexity of each transfer-type and the complexity of their parallelism. The former is system-specific while the latter is relatively independent from a specific system, making TRG applicable to a wide range of designs.

• Modeling TRG does not require extensive knowledge about Hardware implementation. Most effort is required in modeling each transfer-type, e.g., composing T.P (·).
• TRG is a method independent from simulation platforms or Hardware abstraction levels. Tests generated from a TRG can be used as performance tests as well as functionality tests. SoC designers can use TRG to plan test-cases of parallelism to evaluate performance penalties due to resource-contentions long before a system is actually integrated; TRG may also find its applicability in generating manufacturing tests.
• The target bugs are not those bugs obviously local to each Hardware component, but hard-to-detect bugs buried in 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 easily implemented as functional checkers in test-benches. However, a failed result gives limited indication of the nature and location of the bug. Therefore, other error-detection mechanisms (e.g., assertions) should also be implemented in test-benches.

4. TRG and Petri-net for Coverage

4.1 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 a 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 (line, toggle, conditional, etc) and local state machine based coverage measures 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 (Peterson, 1981; Zhu & He, 2002) as the model because its semantics can describe is resource-constrained concurrency.

Definition: A Petri-net is a directed graph represented by a quintuple (**P**, **T**, **F**, W, M_0), where,

• **P** is a set of nodes known as *places*; each place can hold *tokens*. Tokens are all identical;
• **T** is a set of nodes known as *transitions*;
• **F** is a set of directed arcs (known as *flows*) connecting places and transitions, i.e.,

$$\mathbf{F} \subseteq (\mathbf{P} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{P});$$

• Function W: $\mathbf{F} \rightarrow \mathbf{N}^+$ ($\mathbf{N}^+$: positive integers.); W(f) is called the *weight* of flow f;
• Function M_0: $\mathbf{P} \rightarrow \mathbf{N}$, (**N**: non-negative integers,) known as *initial marking*. M_0(p) is the number of initial tokens in place p. ■

A transition t is said *enabled* when the number of tokens in each input places is no less than the weight of the corresponding input flow. When enabled, t can (but does not have to) *fire*, consuming $W(f_i)$ tokens from the input place connected via f_i, and producing $W(f_o)$ tokens in the output place connected via f_o. Firing does not consume time. 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 and whose directed arcs represent the transitions between states.

4.2 TRG and Petri-net

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

• In TRG, concurrency can be defined as a scenario with more than one transfer. In Petri-net, concurrency means a transition has multiple incoming or outgoing flows.

• In TRG, conflict means that the same resource is accessed by multiple transfers, and at least one access is exclusive. In Petri-net, conflict means a place has multiple incoming or outgoing flows.

Although capable to specify system-level concurrency and conflict, 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 temporal relations can only be, and *are meant to be*, realized in simulation. For example, TRG does not and cannot specify that, at which moment in transfer T_1's life, 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 resource-contentions between T_1 and T_2.

While a scenario in a TRG only represents a snapshot of data-flows, an execution of a Petri-net captures the temporal aspect of the system's behavior. Its reachability graph (readily obtained from a Petri-net tool) can enumerate the possible executions. Therefore, a Petri-net 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 the TRG, create a place P_R in the Petri-net;

(2) **Converting Transfers:** for each transfer T in the TRG,
- create two transitions T_{start} and T_{end};
- create a state-place $T_{running}$ (cf. resource-place P_R.);
- create two flows of weight 1, one from T_{start} to $T_{running}$ and the other from $T_{running}$ to T_{end};

(3) **Connecting Transfers and Resources:**
- First, for each transfer-resource pair (T, R) that satisfies $U(T, R) = \mathsf{s}$:
  - 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 $U(T, R) = \mathsf{e}$:
  - 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.

Figure 6 shows a Petri-net constructed from the TRG of the Nios SoC.

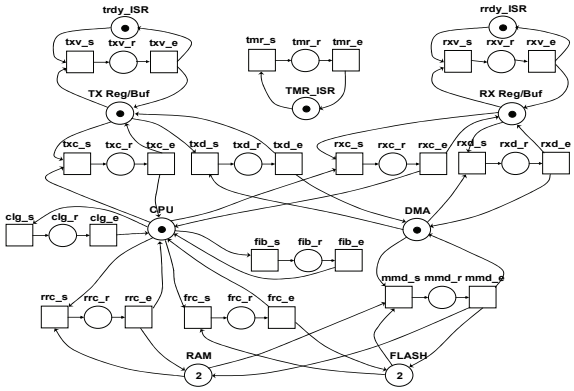

Fig. 6. Petri-net converted from a TRG.

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
- $|\mathbf{F}| = 2|\mathbf{T}_{trg}| + 2|\{(T, R) : (T, R) \in (\mathbf{T}_{trg} \times \mathbf{R}), U(T, R) \in \{e, s\}\}|$.

4.3 Use of Petri-net

Once a Petri-net is obtained from a TRG, we can use the net in a number of ways. For instance, we could prove the liveness and boundedness of the Petri-net; 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 rather theoretical topics are beyond the scope of this chapter.

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. For instance, the reachability graph can indicate the total number of (unparameterized) scenarios in the TRG, because each state in the reachability graph simply represents a scenario in the TRG. This size contributes to the total complexity of scenario generation algorithm in Section 3.4. Generally, there are several options to define the space:

- All states in the graph (i.e., markings);
- All state-state transitions in the graph (not 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 (Zhu & He 2002), 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. Some coverage spaces could be just too enormous to be practically covered due to the graph size and connectivity, in such cases, some restrictions can be applied to bound the coverage space, such as limiting the length of the path and the size of the cycle.

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, and 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 5.3 for details.

4.4 Discussion

It should be noted that both TRG and the Petri-net converted from a TRG are high level abstraction of an SoC (with its application). Most resource-contentions at physical level are invisible in the TRG and the derived Petri-net, simply because physical resources are not present in these models. This is a feature as well as a limitation. If we feel the resulting Petri-net is too "coarse" for coverage purpose, we can consider the following approaches.

- Adjust the size of the TRG by dividing a transfer-type into some sub-types (see Section 2.4), so that the resulting Petri-net has more states and state-transitions. This is in effect to take transfer parameterization into coverage consideration.
- Include more aggressive coverage spaces, such as path coverage and cycle coverage. This is effectively to take more detailed temporal relations into coverage consideration.
- Take system-specific knowledge into coverage consideration. We could model a transfer as an FSM with more internal states and transitions in addition to the simple "start", "running" and "finish", and convert it to a state-machine typed Petri-net (i.e., a net with only one token) and embed it in the backbone net derived from the TRG. It can be seen that TRG-derived Petri-net provides a reasonable start-point and backbone for a more accurate coverage model.

5. Experiments

5.1 Statement Based Coverage Measures

We have observed that reasonable statement coverage measures can be achieved by test-programs generated from a TRG. Since another software-based test generation methodology called SALVEM (Cheng et al., 2005) is demonstrated on the same Nios SoC, we compare the results of SALVEM tests with the test results of TRG. Figure 7 shows the comparisons of the statement (toggle and conditional) coverage. The TRG method has higher coverage 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 coverage on CPU using 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 article. We believe that the TRG method imposes no restrictions on achieving reasonably high statement-based coverage.

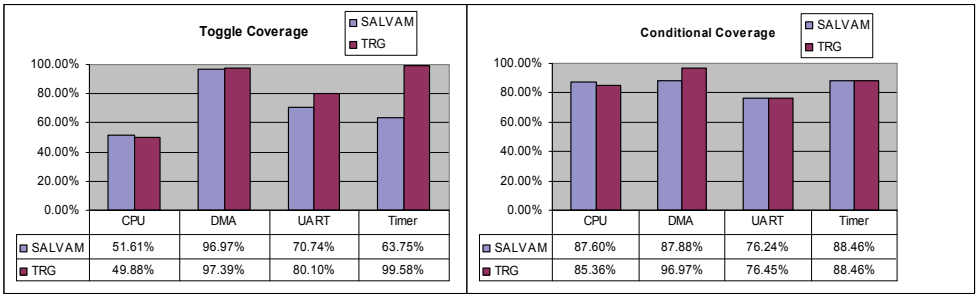

Fig. 7. Toggle and conditional coverage comparisons.

5.2 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; the theoretical space of size 2^{64} is so large that even its reachable sub-space is impractical to be traversed exhaustively by any set of real-world tests. 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 filled with scenarios holding one or two transfers, and the other contains TPs filled with maximal scenarios. Figure 8 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.

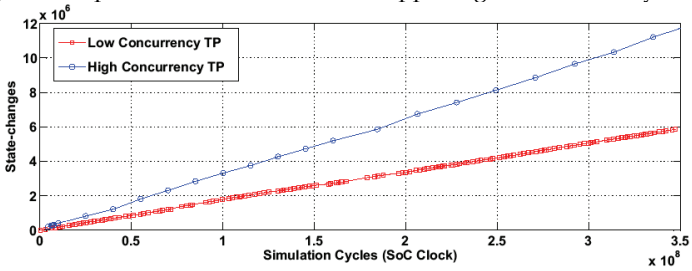

Fig. 8. State-changes against simulation cycles.

However, faster state-change rate does not necessarily mean efficient state-space traversing, for any state can 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 9, 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 less sensitive to the number of known-states. This encouraging comparison implies that concurrency is the key to efficiently exploring the state space.

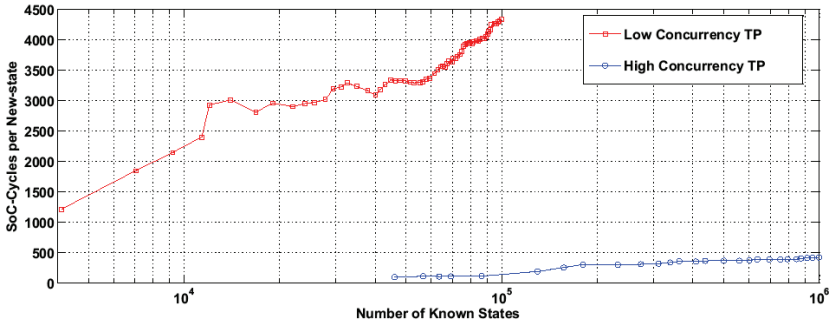

Fig. 9. New-state emergence rate against the number of known-states.use

5.3 Test Generation with Feedback

We modeled 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 3.4), 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).

First, 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.

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: 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 10 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.

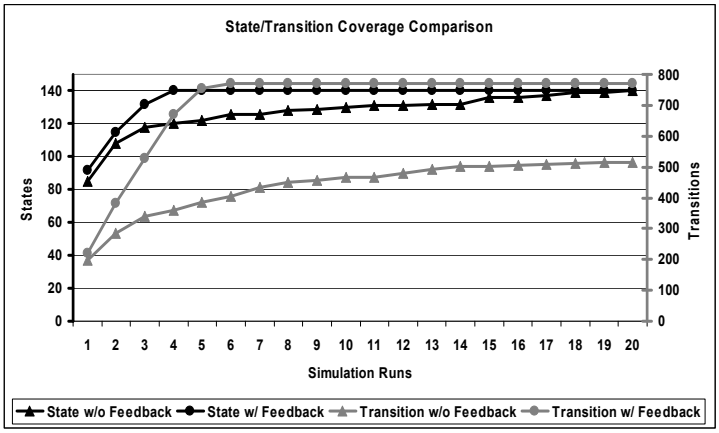

Fig. 10. State coverage and transition coverage with and without feedback.

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.

6. Conclusion

Software-centric and interaction-oriented verification presents a natural interpretation of a system: a system is made up of programmer-controlled data-flows, i.e., transfers, which are constrained by programmer-controlled resources. As the result, the central part of the system-level verification problem is now modeled as a concurrency problem and can be dealt with concurrency models. Particularly, the test-generation problem is handled by the TRG model and the coverage measures can be treated by the Petri-net model, whose backbone can be converted from the TRG model.

Our method has been successfully demonstrated on the single processor Nios SoC. But 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 a TRG can represent certain level of resource-constrained concurrency, we may need to incorporate the domain knowledge about a real world system to capture enough information regarding fine-grained resource competitions. More research in this area is necessary on problems such as to include how much domain knowledge to trade-off between accuracy and scalability.

7. References

Altera, Inc. (2004). *Nios Hardware Development Tutorial ver 1.2*, 2004, http://www.altera.com/literature/tt/tt nios hw.pdf

Bergeron, J.; Cerny, E.; Hunter, A. & Nightingale, A. (2005). *Verification Methodology Manual for SystemVerilog*. Springer Science Business Media, Inc., ISBN: 978-0387255385, NY

Cai, L. & Gajski, D. (2003). Transaction Level Modeling: An Overview, *Proceedings of International Conference on Hardware-Software Codesign and System Synthesis*, pp. 19-24, ISBN: 1-58113-742-7, Newport Beach, CA, Oct 2003

Cheng, A.; Parashkevov, A. & Lim, C.-C. (2005). Verifying System-on-Chips at the Software Application Level, *Proceedings of IFIP-WG Conference on Very Large Scale Integration System-on-Chip*, pp. 586-591, ISBN: 0729806103, Perth, Australia, Oct 2005

Goering, R. (2008). Ten 2008 Trends in System and Chip Design. *SCD Source Online Article,http://www.scdsource.com/article.php?id=68, Feb 2008.*

Katz, S.; Grumberg, O. & Geist, D. (1999). "Have I written enough Properties?" – A Method of Comparison between Specification and Implementation. *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pp. 280–297, ISBN:3-540-66559-5, London, UK, 1999

Keutzer, K.; Malik, S.; Newton, R.; Rabaey, J. & Sangiovanni-Vincentelli A. (2000). System Level Design: Orthogonalization of Concerns and Platform-Based Design, *IEEE Transactions on Computer-Aided Design*, Vol 19, No. 12, (Dec 2000), pp. 1523-1543, ISSN: 0278-0070

Mosensoson, G. (2002). Practical Approaches to SoC Verification. *Proceedings of DATE User Forum*, 2002.

Peterson, L. (1981). *Petri-net Theory and the Modeling of Systems*, Prentice Hall, ISBN: 978-0136619833, Upper Saddle River, NJ, USA

Saleh, R.; Wilton, S.; Mirabbasi, S.; Hu, A.; Greenstreet, M.; Grecu, C.; & Ivanov, A. (2006). System-on-Chip: Reuse and Integration. *Proceedings of the IEEE*, Vol 94, No.6, pp. 1050–1069, ISSN: 0018-9219

Xu, X.; Lim, C-C. & Liebelt, M. (2008). Positioning Test-Benches and Test-Programs in Interaction-Oriented System-on-Chip Verification, *Proceedings of IEEE International Workshop on High Level Design Validation and Test*; pp. 3-10, ISBN: 978-1-4244-2922-6, Nevada, Nov 2008

Xu, X. & Lim, C.-C. (2007). Using Transfer-Resource Graph for Software-Based Verification of System-on-Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 27, No. 7, (July 2008), pp. 1315-1328, ISSN: 0278-0070

Zhu, H. & He, X. (2002). A Methodology of Testing High Level Petri-nets, *Information and Software Technology*, Vol 44, (June 2002), pp. 473-489,