Integration Verification in System on Chips Using Formal Techniques

Subir K Roy
1: Texas Instruments
Bangalore, India

1. Introduction

System on Chips (SoCs) have become an all pervasive component in many of the equipments - both the common placed and the sophisticated, that are relied upon by human beings in today's modern societies; ranging from mobile phones, personal computers, microwave ovens, high definition televisions, base stations for cellular mobile communication and automobiles. Their penetration into every day aspects of human life, and the range of applications and products in which SoCs are being deployed is increasing at a rapid pace. To keep up with this rapid pace it is imperative to design SoCs with reduced turn-around time and cost. Towards this, SoCs are being increasingly designed by integrating existing in house IPs, or third party IPs provided by external vendors. The integration process in realizing an SoC implementation consists of several different kinds of integration which can be classified as (1) static integration, which is essentially of a non-functional nature consisting of simple electrical connections (or hookup) of the inputs and outputs of different component IPs, (2) dynamic, and (3) functional integration; where, besides the pure electrical connectivity, a temporal and a functional dimension, respectively, needs to be taken into account [1]. Typical sizes of state of art SoCs range from fifty million to a few hundred million logic gates. Designing these SoCs involves an integration process consisting of tens of thousands of pure static connections that needs to be established between the input and output ports of the constituent IPs, and when carried out manually can result in introduction of inadvertent errors [1], involving wrong connections, or even, no conncections. The degree of the effects manifested by these errors, depends on when they are detected in the design verification cycle. The latter these are observed in the design cycle, the more difficult and expensive are these to detect, and consequently, to correct, in the implementation. While several approaches have been adopted to tackle the issue of integration verification of SoCs, in this chapter, we focus on the use of formal verification techniques to solve them.
While formal verification has been used in, rather, niche areas of functional validations of IPs and modules, it has found application in the domain of SoC functional validation only recently[13]. With increasing maturity of commercial offerings of formal verification tools by EDA vendors this area of application is expected to grow at a fair pace. The issue of which category of formal verification approaches needs to deployed, for different aspects of SoC functional validation, is however, largely left unanswered. In this chapter we give a glimpse, in Section 2, of the different formal verification techniques that are available, either

Source: Micro Electronic and Mechanical Systems, Book edited by: Kenichi Takahata,
ISBN 978-953-307-027-8, pp. 572, December 2009, INTECH, Croatia, downloaded from SCIYO.COM

as academic tools, or as commercial offerings, and see their applicability to different aspects of SoC verification. We discuss the underlying concepts, the strengths and weaknesses of each approach, the justification for taking these approaches, so that the interested reader can make a judicious choice in their intended application domains. We also point to important references in each of the approaches, so that the interested reader can refer to them for more details.

In Section 3, we will briefly allude to existing methodologies using the formal verification approaches that have been reported in the literature to set the stage for presenting approaches that are not covered by them. More specifically, we will highlight an important aspect of SoC integration verification, vis-a-vis DFT logic, to show the manner in which re-usability is leveraged through automated generation of re-usable parameterized properties and constraints for DFT logic and the hookup or integration logic. And towards "ends justifying means" we will present data and results from their deployment on a real SoC design and show the benefits that can be derived from these approaches. In Sections 4, we will present one interesting scenario from the domain of DFT IP verification.

In Section 5 we will summarize the main contribution of our approaches, which are (1) effective use of formal techniques based on symbolic model checking in the top level verification of SoC integration, (2) effective use of abstraction and modeling of SoC sub-systems in enabling assertion based formal verification, (3) automated generation of assertions and constraints to detect integration errors, (4) automated generation of scripts to capture the SoC design information and invoke a formal verification tool on which to prove the validity or correctness of these assertions. We will end this section and the chapter by drawing conclusions from the presented approaches, data and results, respectively.

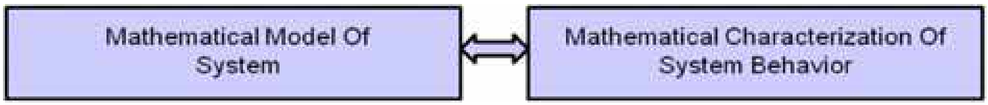

Mathematical Model Of System
Mathematical Characterization Of System Behavior

Fig. 1. Formal Models and System Behaviour

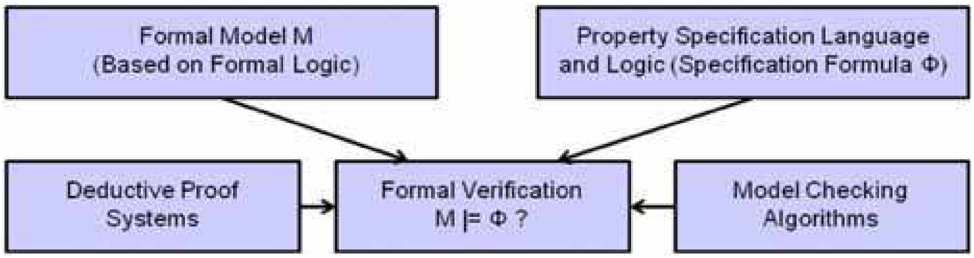

Formal Model M (Based on Formal Logic)
Property Specification Language and Logic (Specification Formula Φ)
Deductive Proof Systems
Formal Verification M |= Φ ?
Model Checking Algorithms

Fig. 2. Generic Structure of the Formal Verification Process

2. Formal approaches

In this section, a brief introduction to formal verification for hardware and a brief review of the different formal verification approach is given. For a detailed presentation and review of hardware formal verification techniques and their application to the problem of verifying IPs the readers are refered to the survey paper given in reference [3, Greenstreet]. The block diagram of the generic structure of the formal verification process, in Figures 1 and 2, succinctly explains the key components involved in formal verification. At the most abstract level [Figure 1] formal verification essentially consists of having (1) a general mathematical model (**M**) , capturing abstractly the system being verified, (2) the system behavior, described abstractly, again, through a set of mathematically well characterized formulae (**Φ**), and finally (3) proving that the set of formulae (**Φ**) holds true on the mathematical model (M), represented symbolically by **M |= Φ**. This is further elaborated in Figure 2, where **M** is either a computational model or a formal logic model, **Φ** is a set of formulae from a formal logic system, and the proof techniques used for establishing the truth value of **M |= Φ** are either deductive or based on model checking. In deductive proof systems, **M** is decribed by a set of axioms (also known as invariants of the system), and the proof method essentially consists of establishing that the truth of **Φ,** in the underlying formal logic, by using only the given set of invariants (or axioms) of the system. The proof is largely driven by inputs provided by the user, and therefore not fully automated, though steps in the proof may lend themselves to full automation. On the contrary, in the model checking approach, the proof is fully automated for some of the underlying formal logic, as it is based on constructing the reachable set of states of the system.

2.1 Symbolic model checking

A hardware module is formally verified by stating a property on the design and then checking that the design satisfies the property. The most commonly specified property is an invariant, which expresses a condition on the hardware module that should never happen in a reachable state (or conversely, a condition that should always be true in a reachable state). Formally, an *invariant* is a boolean formula over the signals of the module. The module M satisfies the invariant I if every reachable state of M satisfies I . Thus, invariant verification on a module is performed by computing the set of its reachable states. However, this computation is difficult because the set of reachable states can be exponential in the number of signals in the module. This exponential growth in the number of states is known as the *state explosion problem.*

Model checking is one of the most popular approaches to formal verification. In model checking, a mathematical representation of a design in the form of a finite state machine (FSM) is first constructed. Any specified behaviour (or a specification) of the design is then formally stated in terms of a property, or a assertion, in unambiguous terms, both syntactically and semantically, in a formal temporal logic. The mathematical model, i.e. the FSM, is then analysed using different state traversal techniques starting from the set of initial states, to check whether it satisfies the formal temporal property, on all, or atleast, one computational path of the state transition graph that is implicitly generated by the above state traversal. This state traversal is known as *reachability analysis*. In case the temporal property is violated or falsified, a trace with respect to the primary inputs and state variables of the FSM, starting from its set of initial states, is generated up to the Kth set of states, where the property fails on one of its states. This is known as an *error trace*. This search is realized because every set of states that is reachable on each clock cycle starting from the set of initial states is stored internally by the model checker. The collection of such sets of reachable states is finite for a finite state machine. When each of the reachable state set is implicitly represented as a *binary decision diagram (BDD)*, the model checking technique is known as symbolic model checking(SMC). BDDs enable a compact representation of the set of states. In many situations, the negation of a desired property needs to be verified, so that the error trace generated automatically by the symbolic model checker when the stated

property is falsified, or the desired property satisfied because of the negation, will result in a sequence of input and state variable data values in the abstract FSM model. Thus, we implicitly use symbolic model checking as a sophisticated search engine. For a number of hardware designs, while it may be possible to construct the the BDD representation of a very large set of reachable states, it may be impossible and infeasible to explicitly enumerate such a set of states. Despite this, in most cases invariant verification based on SMC techniques is limited to a few hundred signals and states.
In symbolic model checking, properties are specified using different temporal logic, e.g. *Linear Temporal Logic (LTL)*, or *Computation Tree Logic (CTL)* [3]. Some of the temporal properties specified in LTL, or CTL, can be equivalently specified in the form of a finite state machine (FSM) using the same set of internal signals that were used to define them in LTL, or CTL. We, next, give a brief overview of CTL and LTL.

2.2 CTL model checking

The main purpose of a model checker is to verify that a model satisfies a user specified set of desired properties. Specifications to be checked can be expressed in two different temporal logics: the Computation Tree Logic (CTL), and the Linear Temporal Logic (LTL).
CTL is a *branching-time* logic. Its formulas allow for specifying properties that take into account the non-deterministic, branching evolution of a FSM. The evolution of a FSM from a given state can be described as an infinite tree, where the nodes are the states of the FSM and the branching is due to the non-determinism in the transition relation. The paths in the tree that start in a given state are the possible alternative evolutions of the FSM from that state. In CTL one can express properties that should hold for *all the computational paths* that start in a state, as well as, those that should hold only for *some of the computational paths*.
As an example, consider the following CTL formula ***- AF p***. It expresses the condition that, for *all* the paths (***A***) starting from a state, *eventually in the future* (***F***) condition ***p*** must hold. Thus, in every possible single path of the computation tree over which the abstract model of the design, or system, evolves temporally, it will eventually reach a state in which the condition ***p*** is logically satisfied; i.e. in the considered temporal logic the formula will be asserted as a TRUE, in this state. Differently from this, the CTL formula ***EF p***, has the semantics, that requires the *existence* (E) of any one, or some path that eventually, in the future, satisfies ***p***. Similarly, formula ***AG p*** semantically implies that condition ***p*** is satisfied always (or *globally)*, i.e. it is true in every state in every path that exists in the computation tree; while formula ***EG p*** requires that there is some path along which condition ***p*** is true in all states in that path. Other CTL operators are as follows,

- ***A[p U q]*** and ***E[p U q]***, requiring condition ***p*** to be true *until* a state is reached that satisfies condition ***q***;
- ***AXp*** and ***EXp***, respectively, require that condition ***p*** is true in all, or in some of the next states reachable from the current state.

2.3 LTL model checking

In this, specifications or properties are expressed in linear temporal logic (LTL). LTL characterizes each linear path induced by the FSM (linear time approach). LTL has a different expressive power as compared to CTL. Typical LTL operators are :

- ***Fp*** ("in the future ***p***"), stating that a certain condition ***p*** holds in one of the future time instants.

- ***G p*** (" globally ***p***"), stating that a certain condition ***p*** holds in all future time instants.
- ***p U q*** ("***p*** until ***q***"), stating that condition ***p*** holds until a state is reached where condition ***q*** holds.
- ***X p*** ("next ***p***"), stating that condition ***p*** is true in the next state.

Compared to CTL, LTL temporal operators do not have CTL path quantifiers **A** or **E**. LTL formulas are evaluated on linear paths, and a formula is considered true in a given state, if it is true for all paths starting in that state. Its performance is similar to CTL model check as described above. It has been shown that the complexity of a LTL symbolic model checking algorithm is higher than that of a CTL symbolic model checking algorithm.

2.4 Bounded model checking

In Bounded Model Checking (BMC) the model checker instead of evaluating CTL or LTL properties on paths over infinite time, does so over a finite time defined by a parameter ***k*** which represents ***k*** units of time. It tries to find a counterexample of increasing length, and immediately stops when it succeeds, declaring that the formula is false. The maximum number of iterations can be controlled by the parameter ***k***. If the maximum number of iterations is reached and no counter-example is found, then the model checker exits, and the truth of the formula is not decided, i.e. it cannot be concluded that the formula is true, but only that any counter-example should be longer than the maximum length. The model checking engine in most implementations of BMC is based on a satisfiability (SAT) solvers instead of BDDs. The complexity of SAT solvers depend on the number of satisfiability constraints that need to be formulated, which in turn is directly dependent on the parameter ***k***. For reasonable values of ***k,*** BMC based on SAT is computationally more efficient than SMC based on BDDs [4].

2.5 Checking invariants

BMC can be used, not only for checking LTL specification, but also for checking invariants. An invariant is a propositional property which must always hold. BMC tries to prove the truth of invariants via a process of inductive reasoning, by checking if (i) the property holds in every initial state, and (ii) if it holds in every state that is reachable from a state where the propositional property holds.

2.6 Newer approaches

Here, we highlight the need to look for other formal verification approaches. We present brief descriptions of some of the promising approaches that are from areas of ongoing research and development in formal verification, in both academic and industrial research circles.

Formal verification has been applied to many classes of designs [13]. We will discuss this aspect in some details in a later sub-section. The key drawback of the automated symbolic model checking based formal verification approaches has been the bane of state explosion faced by even moderately sized modules. Any module, in which the number of state elements or flip-flops exceeds 1000, is liable to face the issue of state explosion during the formal proof of the properties. Microprocessors with modest capabilities, such as the following - non-pipelined instruction stage, single stage instruction pipeline, four stage instruction pipeline, and a four stage instruction pipeline supporting jump and branch instructions - are known to result in state explosion. Typically, for the different SMC approaches the increasing order of performance are as follows,

- CTL, or LTL model checking,
- Invariance checking using CTL,
- Invariance checking with the CTL, or LTL temporal properties represented as FSMs,
- Bounded model checking, and
- Bounded model checking with CTL, or LTL temporal properties represented as FSMs.

One approach to addressing the state explosion problem in such designs is to use compositional formal verification techniques, at the module level of the design heirarchy. Compositional verification is enabled by the assume and guarantee approach [3]. This is shown in Figure 3 below.

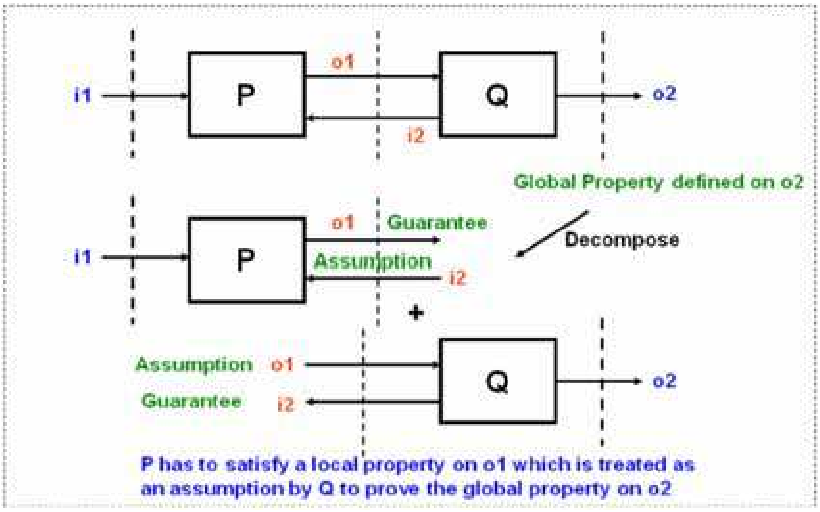

i1
P
o1
i2
Q
o2
Global Property defined on o2
Decompose
i1
P
o1
Guarantee
Assumption
i2
+
Assumption
o1
Guarantee
i2
Q
o2
P has to satisfy a local property on o1 which is treated as
an assumption by Q to prove the global property on o2

Fig. 3. Assume and Gaurantee approach to Compositional Verification

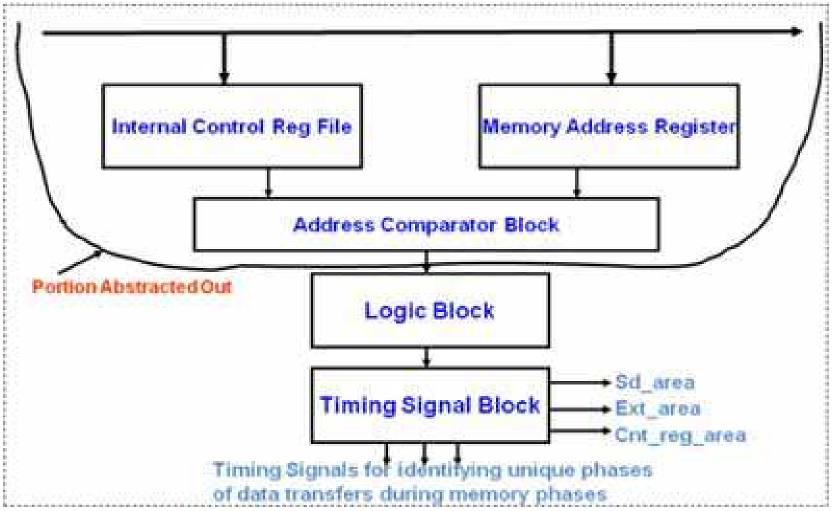


Fig. 4. Design Abstractions to Reduce Complexity of Formal Verification

Another approach is that of abstraction (see Figure 4 and Figure 5), where the design is abstracted (or simplified), to remove portions of design not needed to prove a property. This can result in a substantial reduction in the number of flip-flops, thereby enabling automated proof convergences of the formal properties.
However, for complex industrial RISC and DSP processors, or SoCs based on them, even these approaches are not be feasible. We will need newer formal verification approaches which are not limited by the state explosion problem.
Recent research carried out by different academic and industrial research groups address these capacity issues in formal verification. Though, no stable implementations of formal verification tools exist for such approaches, they serve as good pointers to pursue in the future to address difficult verification problems. We give below, very brief descriptions of some of the approaches.

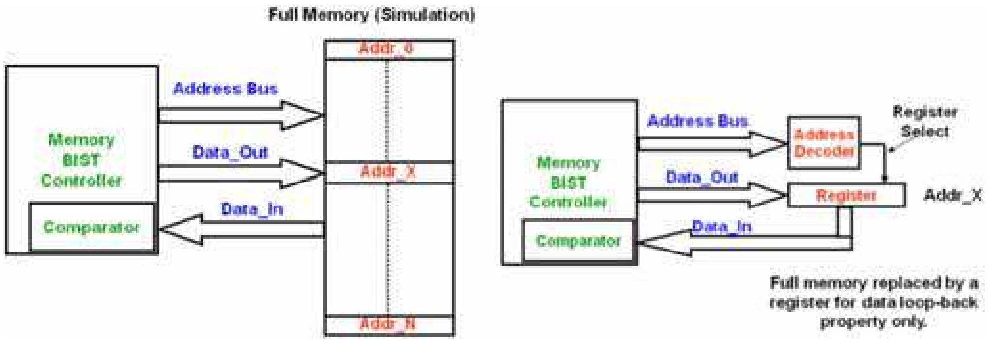


Fig. 5. Memory Abstraction to Reduce Complexity of Formal Verification (Each memory bit adds to a state bit in the verification process)

2.7 Generalized symbolic trajectory evaluations

Symbolic trajectory evaluation (STE) provides a means to formally verify properties of a sequential system by a modified form of symbolic simulation. In this the desired system properties or specifications are expressed in a notation combining Boolean expressions and the temporal *next-time* operator. If the state space of a system is a lattice, the behavior of the sytem can be expressed as a *trajectory*, a sequence of points in the lattice determined by the initial state and the system functionality. Formulas in a simple temporal logic express properties of the system. Given a formula, one can derive bounds that trajectories with the desired property must obey. In its simplest form , each property is expressed as an assertion [A => C], where the antecedent A is a trajectory formula which expresses some assumed conditions on the system state over a bounded time period, and the consequent C another trajectory formula which expresses conditions that should result. That is, it determines whether or not every state sequence satisfying A must also satisfy C. It does this by generating a symbolic simulation sequence corresponding to A, and testing whether the resulting symbolic state sequence satisfies C. A generalization allows simple invariants to be established and proved automatically.
The Boolean expressions provide a convenient means of describing many different operating conditions in a compact form. By allowing only the most elementary of temporal operators, the class of properties that can be expressed is relatively restricted, as compared

to other temporal logics. However, it has been found in [5] that many aspects of synchronous digital systems at various levels of abstraction can be readily expressed. It is adequate for expressing many subtleties of system operations such as instruction pipelining in modern processors.
The verifier operates on system models in which the state space is ordered by "*information content*". By suitable restrictions to the specification notation, it can be guaranteed [5] that for every trajectory formula, there is a unique weakest trajectory for A and testing adherence to C. Also, establishing invariants corresponds to simple fixed point calculations. STE implementation of [5] requires a comparatively small amount of simulation and symbolic manipulation to verify an assertion. In [5] it is shown that the length of the simulation sequence depends only on the depth of nesting of the temporal next time operators in the assertion and the speed of convergence of the fixed point calculations.
Formal verification techniques such as, symbolic model checking and theorem proving have met with limited success because of intrinsic problems related to state explosion and the need for manual intervention, respectively. Even though STE is less sensitive to state explosion problem and proven to be a viable methodology for large scale data path verification, it suffers from the problem of inexpressibility. Properties which are spread over infinite time intervals cannot be expressed in STE, let alone be verified [5,6]. GSTE constitutes a very significant extension to STE [8-10]. It has been used successfully by INTEL on its new generation microprocessor designs. GSTE addresses the drawbacks of STE and has the power to verify complex assertion graphs with which any ω-regular property can be equivalently represented, while at the same time it preserves the benefits of STE, like the insensitivity to state explosion, thereby capturing the expressiveness of classical model checking ([3-4] and [6]).
Verification of a complex pipelined data path designs and memories using GSTE model checking techniques have been reported in the literature. Complex properties which are spread over infinite time intervals are specified and verified. The verification time is improved by carefully reducing the number of precise nodes used to perform reachability analysis, while providing complete state information to the symbolic simulator. These results prove the viability of the GSTE methodology as a formal verification technique for control dominated designs such as large scale pipelined data paths. GSTE, therefore, seems a good candidate formal verification approach to use, as it appears to scale well with the actual implementation model of a processor.

2.8 Theorem provers

These are based on formal systems such as logic. For hardware verification, both the specification and implementation can be described in a formal logic, and the task of verifying the system is to prove that the implementation entails the specification. The core of a theorem prover is a set of axioms and inference rules. Using only these, the user can prove a theorem, with the system mechanically checking each step in a proof. One of the best known theorem provers is the HOL system [3], with which theories of different sorts can be built up in a rigorous way using a small number of primitive axioms and inference rules. One major advantage is that the proof can be checked mechanically. Another advantage is that it can be used to argue at different levels of abstraction. As theorem proving is structural rather than behavioral, one can exploit the structure of the system to manage complexity. A major disadvantage of theorem provers is that it can be extremely tedious to verify certain low level properties of systems.

In [6] the combining of theorem proving and trajectory evaluation is explored, with a motivation to gain the benefits of both the approaches. In their theorem proving approach the mathematical objects manipulated by the theorem prover are the trajectory assertions. VOSS is an implementation of these ideas in which STE is used to perform partial verification based on the decomposition of the original specification. Combinational theory is then used to combine these results through the use of the theorem prover framework.

2.9 Logic of Positive Equality with Uninterpreted Functions (PEUF)

This provides a means of abstracting the manipulation of data by a processor when verifying the correctness of its control logic. By reducing formulas in this logic to propositional formulas, one can apply Boolean methods such as BDDs and Boolean satisfiability checkers to perform the verification. In [7], two approaches have been shown to translate formulas in PEUF into propositional logic. The first interprets the formula over a domain of fixed length bit vectors and uses vectors of propositional variables to encode domain variables. The second generates formulas encoding the conditions under which pairs of terms have equal valuations, introducing propositional variables to encode the equality relations between pair of terms. In [7] techniques are presented to drastically reduce the number of propositional variables that need to be introduced and to reduce the overall formula sizes. This allows verification of microprocessors with load, store and branch instructions at both the RTL or the gate level model. This again makes the approach based on PEUF, a good candidate for solving many formal verification problems.

3. Existing formal based approaches

In this section, we first justify the need to resort to formal verification, then we will briefly allude to existing methodologies based on the formal verification approach that have been reported in the literature to set the stage for presenting newer approaches in latter sections.
As an example case study, we will present the methods and challenges in verifying the integration of Design For Testability (DFT) logic - both BIST and non-BIST, in complex SoCs using formal techniques. We will first present a generic architecture of the DFT logic that is typically present in state of art SoC designs. For this DFT logic, we will, next, list the validation task that needs to be accomplished, to ensure its proper integration into the functional logic portion being implemented in a SoC design. We will then identify the commonality that exists amongst the listed tasks from the perspective of verification. We will then show how such common verification tasks are amenable to automation. As some of these verification flow automations have already been reported in available literature, we will briefly describe them in the context of our adoption of these flows, and refer the interested reader to relevant reference papers for more details. As these flows are being applied in the regression mode, to the various revisions of the currently ongoing implementation design of an in-house SoC, we report recent data from our formal verification efforts, to show-case the value propositions brought in by these approaches.
To simplify the above discussion, we will assume that different DFT IPs present in the generic DFT architecture are pre-verified. However, in reality, this is not always the case. In many situations, it may be necessary to verify even the different DFT IPs, specially, if these are parameterized, configurable and auto-generated, to ensure that the version intended for integration into the SoC, is indeed being generated correctly. Towards this, we will briefly

discuss, how congifurable DFT IPs can be formally verified with a configurable set of generic properties, so that alongwith any desired IP configuration, the corresponding set of properties are auto-generated to verify the IP. This will demonstrate how re-usability is being leveraged through automated generation of re-usable parameterized properties and environmental constraints for DFT logic and the integration logic.

3.1 Justification for using formal approaches to integration verification

An exceedingly important design phase, which gets carried out in the background, and far from the lime-light of the functional features of any SoC, is the integration of DFT logic and the verification of this integration to other sub-systems and IPs in a SoC. While this does not feature as a prominent front end task in the design of any SoC, it does constitute a significant portion of the overall design and verification effort. Any savings in this design integration phase, and its subsequent verification, helps in reducing the overall SoC design cost. Some of the key components in DFT logic that need to be integrated into a SoC are those for 1) testing embedded memories and core logic, 2) control logic to enable different test modes to be set up during post-fabrication Silicon testing, 3) multiplexing and control logic to enable different test modes to selectively run tests, directly from SoC top level ports on different portions of functional logic by bypassing intervening logic blocks, 4) configuring scan chains to rout test vectors to different portions of functional logic.

A key to realizing the above mentioned cost reduction for the above tasks can be through their automation. A prominent factor that can help in facilitating automation is the fact that most DFT IPs possess behaviors and structures that are of an extremely canonical and regular nature, and that these are independent of the functional nature of the SoC. Besides this, the interconnection of the IPs to the rest of the logic in the SoC is also of a very generic nature. This has resulted in the consolidation of SoC level DFT logic architecture towards a highly standardized, and a highly configurable form (known as the DFT sub-system), enabling it to be auto-generated through software tools. Individual components within this sub-system are auto-generated using point commercial tools addressing the highly specialized requirements corresponding to each DFT task. Two such examples are, memory and logic BIST controllers. We briefly discuss below the DFT tasks of testing embedded memories and core logic in a SoC, in the context of these controllers.

Present generation SoC designs are built hierarchically with a large number of embedded memories of different kinds and sizes and different embedded cores (e.g. processors, peripherals, etc.). The embedded cores may themselves have different types of internal memories, functional logic blocks and different types of I/O ports. Built-in self-test (BIST) techniques are employed to reduce expensive ATE time for the post manufacturing silicon testing of these blocks; besides, they enable low pin count testing, and testing of embedded core of SoCs fabricated on low cost packages with fewer pin count. Application of test vectors to memories and core logic can span several million clock cycles depending on the size of the embedded memories, the core logic, and the testing algorithm employed to generate these vectors, resulting in exceedingly long verification times for the BIST controllers through simulation. A memory BIST tool needs a description of the embedded memories and memory test pattern generation algorithms to generate and integrate the different memory BIST controller logic needed for different types of memories. In a similar manner, the logic BIST tool requires a description of the gate level net-list representation of a design, to analyze and to extract the core logic portion, before generating the logic BIST controller needed to configure them for testing and for generation of the test vectors specific to this configuration. The configuration hook-up logic and the logic BIST controller are then integrated into the netlist automatically by the tool. To meet performance, timing and power constraints specific to a SoC, and to support scan, self test and clocking, it is often the case that, such, auto-generated BIST logic have to be modified, thereby, necessitating their verification, to ensure that the modifications do not break the intended behavior or functionality.

Verification of proper integration of memory BIST logic into a design, and any modifications to it, to meet performance and timing constraints in the original design, has been traditionally done based on simulation techniques. This is often incomplete and time consuming, as the correctness of the integration is verified indirectly by running the entire test suite developed for memory BIST logic. Even a single change in the control or hookup logic and its integration into the rest of the design may necessitate re-running the entire set of simulation vectors. Besides being time and compute intensive, the time needed to analyze errors detected in these simulation runs and to correlate them to design integration problems can be correspondingly large. These simulation test benches are often created manually, and may be design specific, making them un-useable across different controller-memory, or controller-embedded logic configurations. Verification of memory BIST logic using formal techniques is appealing, as the behavior of the controller block is sequential, while the behavior of the hookup block is combinational. Writing re-usable formal properties for such blocks are easy, precise and less time consuming. It is possible to obtain comprehensive verification coverage across different environmental constraints, resulting in high quality and confidence in the verification process using formal techniques. Formal verification of BIST controllers, however, can be difficult, if we include models of embedded memories, as in simulation. This is due to the large number of register elements used to model memory, which leads to the problem of state explosion and can be overcome by effective modeling and abstraction techniques.

We next justify the need to verify even the auto-generated DFT logic sub-system and its integration into the SoC. DFT logic sub-systems have to be verified as different modular configurations arising out of generic customizable, configurable and parametrisable components may be needed for different SoCs. This implicitly enforces verification requirements on the integration of such configurable DFT logic modules into an SoC whose RTL itself could be auto-generated with a tool (for example, 1-Team-Genesis [11]) and with its own set of configurable functional IPs. While there is variability in the configurations, each configuration nevertheless, retains the above characteristics, thereby, rendering the verification of DFT logic and its integration into a SoC a very good candidate for formal approaches. To leverage the capabilities of FV in the context of auto-generated configurable modules, it is essential that the formal properties themselves be configurable and auto-generated, along with the formal verification environment. This enables high re-usability of properties developed during the tactical formal verification of each module present in the DFT logic subsystem in different SoCs. While the generation of DFT logic and its integration in a SoC is automated, our approach results in the automation of the verification task as well. This enables the complete automation of DFT logic in terms of verification and integration in a SoC at the RTL implementation level, resulting in a considerable reduction in the overall SoC design cost and design turnaround time. We briefly describe below the process by which we systematically achieved this automation.

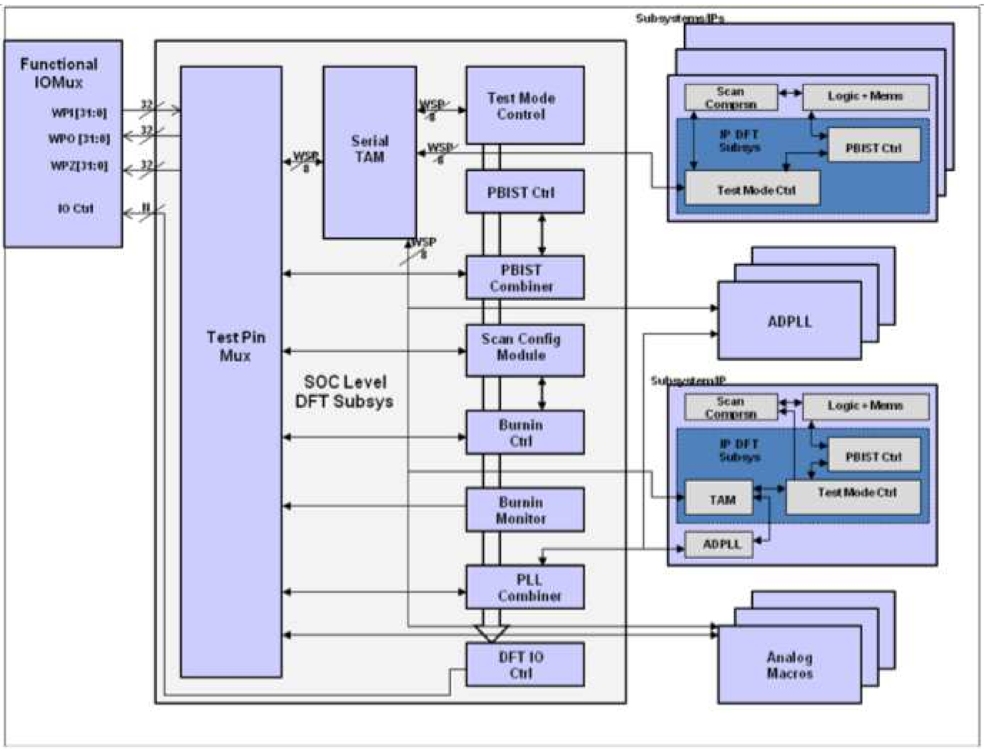


Fig. 6. Generic SoC DFT Logic Architecture

Verification IPs (VIPs) in the form of formal properties, verification environment and verification tool setup were developed tactically for the maximal configuration possible in each block within the standardized configurable DFT sub-system described above, using techniques given in [1,2]. These VIPs were then validated on several in-house driver SoC designs. Once these VIPs reached a level of maturity by way of functional specification coverage, their parameterization in the context of individual blocks were taken up, to enable different sets of VIPs to be auto-generated for a given set of parameters specific to a particular desired configuration of the DFT sub-system. VIPs necessary to check the correct generation of the sub-system includes the VIPs to check the correct integration of individual blocks within the sub-system. Different verification sub-tasks related to the validation of behavior of DFT logic and its interaction with functional logic under different test modes were identified, and corresponding VIPs along with their auto-generation scripts were developed tactically. These were then validated on several driver designs. The tactical development of these VIPs on driver SoC designs were then moved into a common infrastructure through which desired configurations of the DFT sub-system logic and VIPs specific to different SoC designs are generated, enabling high re-use and faster turn-around times.

We next give details of how some of the formal verification flows related to the different DFT verification tasks have been achieved. Towards we first briefly describe a generic DFT logic architecture typically found in any state of art SoC.

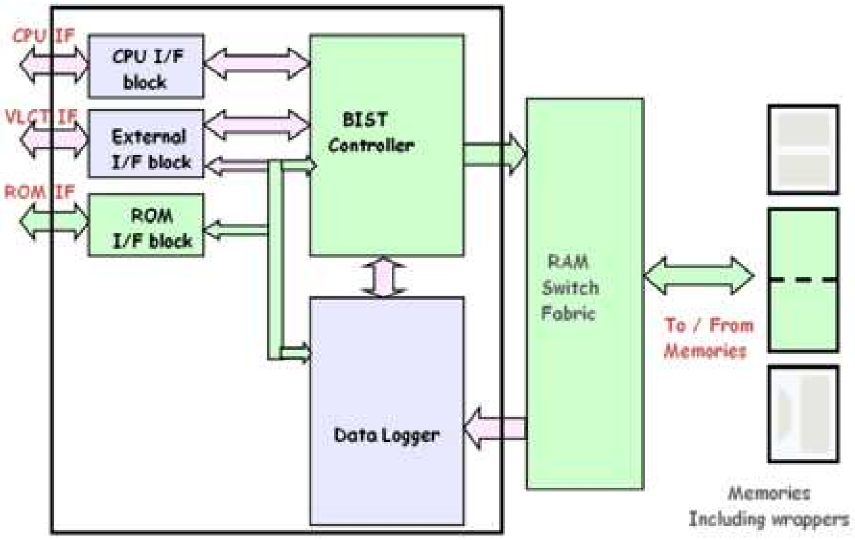


Fig. 7. Microcoded Programmable Memory BIST Controller Architecture.

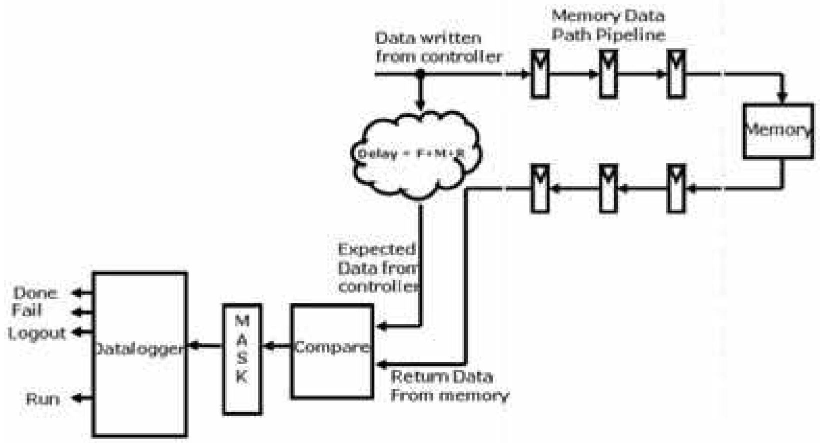

Memory Data Path Pipeline
Data written from controller
Memory
Delay = F+M+R
Expected Data from controller
Done
Fail
Logout
Run
Datalogger
MASK
Compare
Return Data From memory

Fig. 8. Flow of Memory Data between PBIST Controller and Embedded Memories

3.2 Generic SoC DFT logic architecture

Figure 6 shows the DFT logic architecture typically found in any state of art SoC design. This SoC has a heirarchical DFT logic architecture characterised by a complex top level DFT sub-system, and depending on the complexity of the constituent IPs, several simpler IP level DFT sub-systems could be present. The *Functional_IO_Mux* block at the top level of the SoC routes external inputs to the SoC to either the functional core logic or to the DFT sub-sytems depending on the SoC operational modes, viz., functional or test modes. Under the test

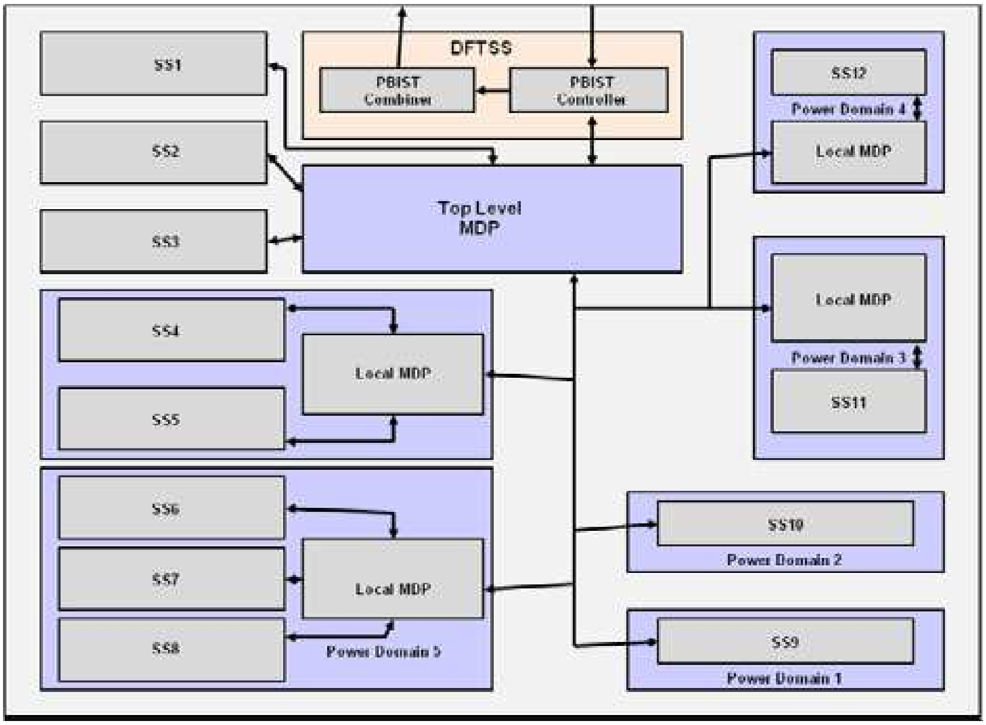


Fig. 9. Generic Top Level Heirarchical Memory Data Path Architecture in a SoC

mode, the external inputs are routed by the *Test_Pin_Mux* block, under different test modes to different modules within the top level DFT sub-system or the IP level DFT sub-system. The *Test_Pin_Mux* block achieves the routing to IP level DFT sub-system blocks through a IEEE1500 module in the *Test_Mode_Ctrl* block, which not only enables setting up of the various SoC level test modes, but also IP level test modes. The latter is achieved through a serial programmation of the *Serial_TAM* block under the control of the IEEE1500 module. The programmation of the IEEE1500 module is carried out by the ATE through top level SoC JTAG ports. The *Test_Mode_Ctrl* block exercises control over choice of, either serial test data, or parallel test data through the *Functional_IO_Mux* block, through the *DFT_IO_Ctrl* block based on the requirements imposed by the different SoC and IP test modes. The different IP level test modes are set by the programmation of the *Serial_TAM* block in the top level DFT sub-system under the control of the IEEE1500 module. Based on the value written into its control register the ports of the different IP level TAM blocks get connected to its corresponding ports. Serial data from the top level SoC ports can then be routed directly to the individual IP level *Test_Mode_Ctrl* blocks to set the desired test modes within the IPs.

The *PBIST_Ctrl* block in the top level DFT sub-system is the programmable BIST controller. Several PBIST controller blocks can also be present in the IP level DFT sub-systems as shown in Figure 6. Memory test data from these controllers and the top level PBIST controller, are routed sequentially to top level SoC test ports by the *PBIST_Combiner* block.

The ***PLL_Combiner*** block controls the generation of the various functional and test clocks with different frequencies needed by different IPs and test controllers under the different functional and test modes of operation. These clocks are generated from the top level system clock of the SoC. For the sake of brevity we will not discuss the remaining blocks present in the top level DFT sub-system.
The correctness of the SoC DFT logic architecture described above is established through different sets of verification checks carried out at different levels in its module hierarchy. We list below some of them.

- SoC Level Checks
  - Hook-up checks
  - IO Related Checks
  - Memory Data Path
  - P1500 Slave Verification
  - Test Mode Entry
- Module Level Checks
  - Burn in monitor module
  - Clock observation module
  - Test secure controller
  - Test clock management module

There are different categories of connectivity checks (***Hook-up Checks***) that need to be carried out at the top level. As listed below there are a large number of connectivity checks that need to be performed at the SoC top level under various categories.

- Hook-up checks
  - Test pin mux verification
  - Clock propagation checks
  - ATPG reset propagation checks
  - ATPG control signal checks for soft macros
  - Memory power management ports hook up
  - Power switch ports hook up
  - WPI/WPO connectivity from DFT-SS to complex IO's, analog macros, digital hard IPs
  - DFT-SS DFT read/write signals to control modules
  - Compression wrapper connectivity
  - Connectivity checks between DFTSS to IPs
  - Burnin monitor input/output connectivity
  - Clock observation/lock observation signal connectivity
  - PLLCM/ADPLL connectivity
  - Connectivity checks for PBIST, DPLL, SCM interface
  - IForce/VSense connectivity checks
  - Memory port connectivity checks
  - Margin mode pin checks
  - Memory power management ports hook up
  - Power switch ports hook up
- Direct Connectivity
  - TPM, DPLL, SCM, PBIST, ATPG Reset, PRCM Clock,
- Muxed Connectivity
  - Burnin monitor muxing logic

- Safe Value
    - IE, PU/PD, GZ checks
- Connectivity with inverted value
    - Slew Override checks
- Test Mode Entry
    - THBMode, TestMode
- Clock division based on a division factor
    - Clock Observation Module
- TAM Connectivity
- Memory Data Path connectivity
- Register loading through JTAG

Most of the above checks are simple point to point, static connectivity checks which have been discussed in details in references [1,2]. We will discuss below, briefly, one check which is more complex as compared to the other checks. More details on this check can be found in the references [1, 2]. This is the memory data path (MDP) check, in which, the correctness of the pipelined datapath connectivity between a PBIST controller and its corresponding set of embedded memories is established. This correctness has to be established individually between every possible pair of controller-memory combination. The setting up of each unique pair is achieved by a hierarchical mux logic structure known as the Memory Data Path (MDP). Each pair can have different numbers of pipelined registers along both the forward memory data path (from controller to the embedded memory) and the return path (from embedded memory to the controller), to account for the different path delays due to different geographical seperations of the embedded memories vis-à-vis the controller (Figure 7 and Figure 8).

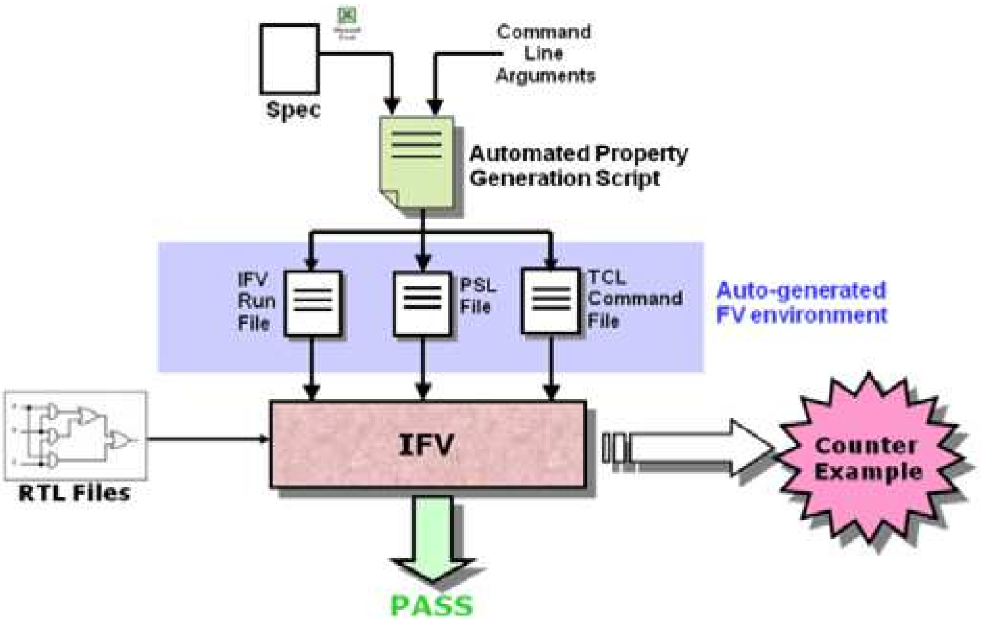


Fig. 10. Automation Flow For Memory Data Verification Using Formal Properties

The MDP check consists of the following – correct establishment of a pair, correct temporal to and fro transportation of the address, memory and control data between the corresponding memory ports and controller ports. Establishing the correctness of a pair under a unique control value issued by the PBIST controller is done by ensuring that only the desired pair is chosen, and that for this pair no other pair is chosen. As will be seen below typical SoCs can have tens of embedded memories of differing types. Besides, each IP can have its own local PBIST controller. Figure 9 shows the hierarchical MDP structure in a typical SoC, with the grouping of the local MDPs being based on the different power domains that each IP belongs to in the SoC. Based on the argument presented earlier in the section, formal verification of the MDP structure for an ongoing SoC implementation is being carried out using an automated formal verification flow, shown in Figure 10. Details of this flow can be found in [1, 2]. We discuss the data obtained from our formal verification efforts for this check.

Of the 14 functional sub-subsystems being integrated into the SoC, 9 are soft IPs, whereas, 5 are hard, pre-verified, third party IPs, requiring only simple connectivity checks. Full set of MDP connectivity checks are carried out on the soft-IPs. The number of memories and their corresponding ports on which connectivity checks are performed are listed in Table 1 below.

IPs	Number of Memories	Ports checked (IP/Mem End)
Sub-System1	**3**	**td, ta, taw, tar, q, twen, tm, twrenz**
Sub-System2	**1**	**td, taw, tar, q, twen, tm, twrenz**
Sub-System3	**28**	**td, ta, taw, tar, q, twen, tm, twtz, twz, twrenz**
Sub-System4	**3**	**td, taw, tar, q, twen, tm, twrenz**
Sub-System5	**23**	**td, taw, tar, q, twen, tm, twrenz**
Sub-System6	**2**	**td, taw, tar, q, twen, tm, twrenz**
Sub-System7	**10**	**td, taw, tar, q, twen, tm, twrenz**
Sub-System8	**3**	**td, ta, taw, tar, q, twen, tm, twrenz, twtz, twz, tez0**
Sub-System9	**1**	**a, ta, q, ez tez, tm**
Sub-System10	**Hard IP – connectivity checks**	**csr, rgs, rds, rdata*, wdata*, addr*, wtz*, ms*, tm, wz*, twrenz**
Sub-System11	**Hard IP – connectivity checks**	**csr, rgs, rds, rdata*, wdata*, addr*, wtz*, ms*, tm, wz*, twrenz**
Sub-System12	**Hard IP – connectivity checks**	**wpi_memory_bist***
Sub-System13	**Hard IP – connectivity checks**	**wpi_memory_bist***
Sub-System14	**Hard IP – connectivity checks**	**wpi_memory_bist***
Total	**74 + connectivity checks**	**NA**

Table 1. Sub-systems, Their Memories and Signals for Formal Verification in Example SoC

The total number of formal properties for each IP is given in Table 2. This table also shows the progression of the checks on different RTL versions released by the design team at different points in the temporal evolution of the SoC implementation. Several useful bugs were caught by the formal verification runs in each release. As can be clearly seen, over each iteration there is a reduction in the number of bugs caught by formal verification.

Towards the formal verification runs, the set up time needed for the first RTL release using our automated flow was approximately 36 hours for all the 14 sub-systems. Most of this time was devoted towards establishing the correct environmetal constraints to be applied at the SoC top level for formal verification runs, and the right heirarchical paths of each functional IP in the SoC, and each module in the heirarchical DFT logic architecture, to enable black-boxing of un-necessary modules. This results in efficient and faster convergence of the properties during formal runs. This is a one time effort. Set up times in subsequent regression runs are drastically reduced to around an hour. A PERL based script is under development to completely automate the above.

	1st iteration			2nd iteration			3rd iteration			4th iteration		
IPs	**Prps**	**Pass**	**Fail**	**Prps**	**Pass**	**Fail**	**Prps**	**Pass**	**Fail**	**Prps**	**Pass**	**Fail**
SubSys1	148	144	4	148	148	0	148	148	0	149	148	1
SubSys2	68	67	1	68	68	0	68	68	0	69	68	1
SubSys3	1344	1312	32	1344	1344	0	1344	1344	0	1372	1372	0
SubSys4	158	155	3	158	158	0	158	158	0	158	158	0
SubSys5	1363	1324	37	1363	1363	0	1363	1363	0	1386	1386	0
SubSys6	48	46	2	48	48	0	48	48	0	54	54	0
SubSys7	670	660	10	670	670	0	670	670	0	690	690	0
SubSys8	172	168	4	172	168	0	172	172	0	175	175	0
SubSys9	38	5	33	38	38	0	38	38	0	38	38	0
SubSys10	29	21	8	29	29	0	33	33	0	35	35	0
SubSys11	29	4	25	29	29	0	33	33	0	35	35	0
SubSys12	NA	NA	NA	3	3	0	3	3	0	3	3	0
SubSys13	NA	NA	NA	3	3	0	3	3	0	3	3	0
SubSys14	NA	NA	NA	3	3	0	3	3	0	3	3	0
Total	4067	3906	161	4076	4076	0	4076	4076	0	4150	4148	2

Table 2. Data For FV Regression Runs on SoC Sub-system Memories For Different RTL releases

The MDP checks on the different sus-sytems/IPs varies from 5 minutes to 10 minutes, with an overall verification time of 90 minutes over different regression runs for different RTL releases. Thus, for each RTL release a regression run of the MDP checks can be completed within 150 minutes (2.5 hours). Simulation based regressions runs need atleast a day to report similar results. This has been consistently observed with respect to other formal verification flows developed to carry out the different SoC level integration checks listed earlier. In Table 3 we report some data based on these checks performed on the latest RTL implementation release of the SoC discussed above.

In the next section we take up the task of verifying formally some interesting aspects of one of the DFT IP discussed above.

4. Formal verification of protocols for transfer of programs and data in programmable DFT controllers

An oft repeated claim in the context of formal verification in the verification community, both academic, as well as, industry has been that model checking based formal approaches do not work well for designs that have behaviors involving aperiodic events with long latencies, such as found in Ethernet MAC interfaces and Elastic Buffers. In this section we discuss a strategy devised to formally verify one such design which involves huge sequential depths.

SL. No.		Properties	Passes	Fails
1	Test Pin Muxing Connectivity + Safe Value Checks	984 272	984 272	0 0
2	SCM Interface Connectivity	518	518	0
3	Burn in Monitor	134	134	0
4	Clock Observation Module Hookup	48	48	
5	Clock divider	1536	1536	0
6	IO Checks –THBMode	475	475	0
7	IO Checks – HiZ instruction	475	475	0
8	IO checks – IDDQ	777	777	0
9	DPLL Interface Connectivity	90	90	0
10	Compression Wrapper Connectivity	153	143	10
11	Boundary Scan Register Connectivity + Override Checks	1610	1592	18
12	EFuse Connectivity + LDO/BG DFT Checks	7 45	5 45	2 0
13	Test Secure Controller Hookup	15	15	0
14	Clock Connectivity Checks	112	112	0
15	Burn-In Monitor Connectivity + Module	95 161	95 161	0 0
16	IEEE1500 TAM Connectivity Checks	550	550	0
17	Memory Margin Mode Checks	21	21	0
18	ATPG Reset Checks	126	73	53
19	Test Mode ATPG Checks	96	96	0
20	DFT Mux Mode + DFT Read/Write Checks	40 16	40 16	0 0

Table 3. Formal Verification Run Statistics on Different SoC DFT Logic Integration Checks

In many critical SoCs (with stringent and low DPPM values) post silicon fabrication verification of embedded memories using programmable built in self test (BIST) controllers involves downloading of memory testing algorithms (for different memory types) in the form of microcoded instructions from an external ROM into the internal memory of the BIST controller. Besides the algorithms, critical information related to them, such as, the embedded memory types and their grouping are also downloaded to enable the controller to execute the memory-testing algorithm on each memory in a group. There is a predetermined grouping of the algorithms and their memory related information, both, within the external ROM and within the internal memory of the controller which enforces a strict protocol with branching semantics to be followed during downloads. Due to limited capacity of the internal memory, the downloading is interleaved with the execution of the memory-testing algorithm by the controller, until each algorithm is downloaded and executed on each memory of their target memory groups. It is, therefore, imperative that the interface implementing the downloading protocol with branching semantics be verified comprehensively for the correct execution of the memory testing algorithms on memories in the targeted groups. In this section, we show how one can effectively use symbolic model checking based formal approach to verify a complex protocol involving long sequence of events until completion of testing of each embedded memory in the SoC.

4.1 The microcoded programmable memory BIST controller architecture

The design under verification here is a ROM Interface which is a block in a programmable memory BIST controller IP (Programmable BIST, or PBIST), as shown earlier in Figure 7. This figure shows the architecture of the PBIST controller. The path through which the data from the external ROM flows into the controller and the embedded memories are highlighted in green. The different memory testing algorithms (ALGO), the information on the memory type (RAM data – RAMD) and the background patterns (BGPs) specific to the algorithm, are all downloaded from the external ROM. The external ROM communicates with the microcoded PBIST memory controller through the ROM Interface, whereby, the relevant data to be downloaded is transferred sequentially to data type specific registers in a program register file within the controller. The memories to be tested are grouped into RAM Groups (RAMG) (Figure 10). PBIST can be instructed to selectively test specific RAMG's *(the targeted RAMG)* using either a single ALGO, or a set of ALGOs applicable to the different memories in the RAMG. The maximum number of RAM groups and the maximum number of different memory testing algorithms supported in the latest version of the PBIST controller is 64 (with a maximum of upto 51 memories in each memory group) and 32 (with a maximum of upto 14 back ground patterns for each algorithm), respectively.

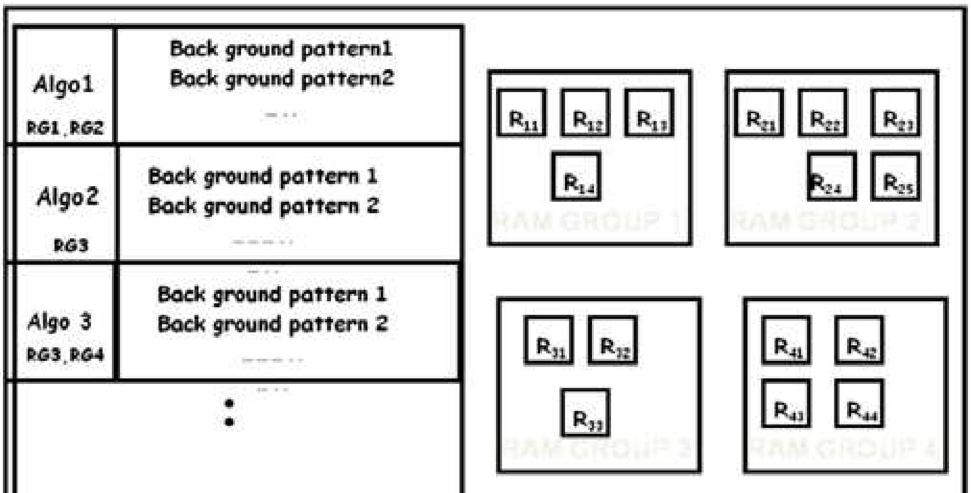


Fig. 10. Memory Test Algorithms and Their Mapping To Memory Groups.

4.2 Source of enormous sequential depth in the PBIST controller's ROM interface behaviour

For the maximum number of algorithms, the maximum number of RAM groups and the maximum number of background patterns that can be supported by a single PBIST controller, we can easily calculate the maximum number of clock cycles it takes for the controller to assert its MDONE (or PASS) signal in case no memory errors are detected by any of the algorithms executed in each memory in each RAM group. To simplify this calculation we will assume the following relevant set of data values :

- There are 32 ALGO, 14 BGP in each ALGO, 64 RAMG and 51 RAMs in each RAMG.
- Each ALGO targets all the RAMG.
- The ROM has a data read latency of 1 clock (we ignore all clocks during which the controller does not attempt to fetch any data from the ROM; for example, during a switch over from the ALGO section to RAMD section, during the execution of a memory testing algorithm on a specific memory in a specific RAM group).

For the above set of assumed values, the number of clock cycles required to just fetch all the relevant data from the ROM into the PBIST controller based on the transaction protocol shown in Figure 11 alone can be easily seen to be,

$$[34+\{2+(64*51*10)\}*14]*32 = 14.2 \text{ million cycles!}$$

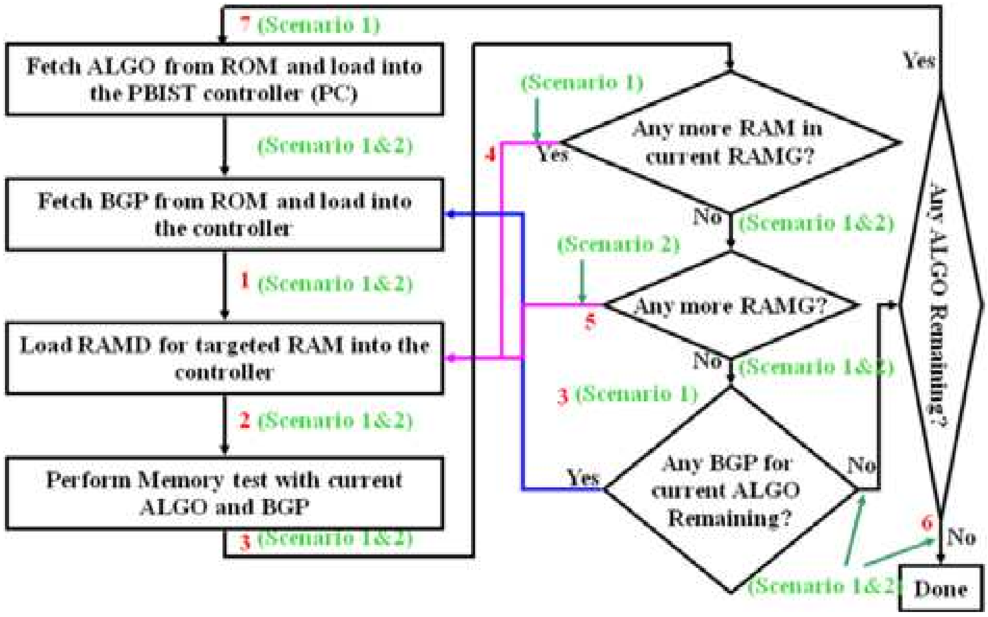

7 (Scenario 1)
Fetch ALGO from ROM and load into the PBIST controller (PC)
(Scenario 1&2)
Fetch BGP from ROM and load into the controller
1 (Scenario 1&2)
Load RAMD for targeted RAM into the controller
2 (Scenario 1&2)
Perform Memory test with current ALGO and BGP
3 (Scenario 1&2)
(Scenario 1)
4
Yes
Any more RAM in current RAMG?
No
(Scenario 1&2)
(Scenario 2)
5
Any more RAMG?
No
(Scenario 1&2)
3 (Scenario 1)
Yes
Any BGP for current ALGO Remaining?
No
(Scenario 1&2)
Yes
Any ALGO Remaining?
6
No
Done

Fig. 11. Transaction Protocol followed by the ROM Interface logic and its functional classification. (Each functional category is numbered in red, while the test case covering it is numbered in green.)

As can be easily noted, this is a rather conservative figure, as we ignore all clock cycles consumed during suspension of data downloads. Besides, if the latency is higher, than the optimistic value of 1, the sequential depth would be even larger. Symbolic Model Checking tools, such as IFV, are incapable of handling functional behaviors with such enormous sequential depths. This was borne out by the fact that even simple properties written to validate the ROM interface behavior exhibited state explosion.

4.3 Verification strategy

To verify design behavior involving exteremly large sequential depths we cannot take recourse to structural abstraction techniques based on module heirarchies to reduce the complexity of the verification effort. A close look at the root cause of the issue reveals the following - during the download process of a data element from the external ROM, one part of a switching logic block is repeatedly exercised every time the control jumps from downloading data from the memory testing algorithm section to the memory data section. Therefore, for a maximum of N memory testing algorithms that are supported by the protocol, this logic will be exercised N times. This also implies that a property written to verify the sequence of events associated with this switching, would be triggered N times in the antecent of the property and, therefore, the final pass status, depending on the satisfaction of the consequent of the property will be declared, after an extremely large sequential depth with respect to the set of initial states is traversed. A simple startegy of reducing N, to say 5, not only exercises the switching logic to check for any corner case arising from the switch in data transcation from the algorithm portion to the memory data portion, and vice-versa; but also results in a smaller sequential depth. This simple idea is similarly used to reduce the number of background test patterns assigned to each ALGO, the number of memory groups, and, finally, the number of memories in each group. Towards this, we chose values of 5 for the number of algorithms, 5 for the background patterns, 5 for memory groups and 5 memories in each group, respectively.

We simplified the verification task, further, by splitting the environment to enable verification of two different cases:

i. 1 algorithm, 1 BGP, 1 RAMG and 5 RAMs in each RAMG.
ii. 5 algorithms, 5 BGP, 5 RAMG and 1 RAM in each RAMG.

The two cases have been carefully chosen to further reduce the sequential depths traversed by IFV to prove the corresponding properties, as well as, exercise complementary portions of the corresponding logic in the RTL. For example, in test case 1, logic enabling transition to a new memory testing algorithm will not be exercised, as only one algortihm is assumed to be present; while in test case 2, logic enabling transition to a new memory in a memory group will not be exercised, as only one memory is assumed to be present in each memory group. The gaurantee on the exhaustiveness of the verification process with respect to the entire functional behavior of the ROM interface logic is based on the following considerations. A complete analysis of the RTL functionality results in its being classifiable into the following seven categories:

1. Branching into the RAMD section once an algorithm and the first BGP have been fetched from the external ROM and transferred to the PBIST controller.
2. Branching from the RAMD section into a wait mode and remaining in that mode until an external signal flags the completion of a memory testing algorithm test on the corresponding RAM.
3. Once a RAM has been tested, the information for the next RAM within the same RAMG needs to be fetched provided the currently chosen RAM is not the last RAM within the present RAMG.
4. Once a chosen testing algorithm has been executed completedly on all the RAMs in a RAM group, control should revert back to the BGP section to enable fetching the next BGP, corresponding to the memory test algorithm to be run on the next memory group.
5. After a RAM has been tested, the information for the first RAM of the next RAMG needs to be fetched, provided the current RAM is the last RAM within the current RAMG

6. After all the memory testing algorithms with all their respective BGPs have run on all their targeted RAMs, the control should revert back to the idle state of the underlying control FSM of the ROM interface logic.
7. Once a RAM has been tested the control should revert back to the ALGO section to enable fetching the next ALGO, in case all the RAMs targeted by the current algorithm have been tested.

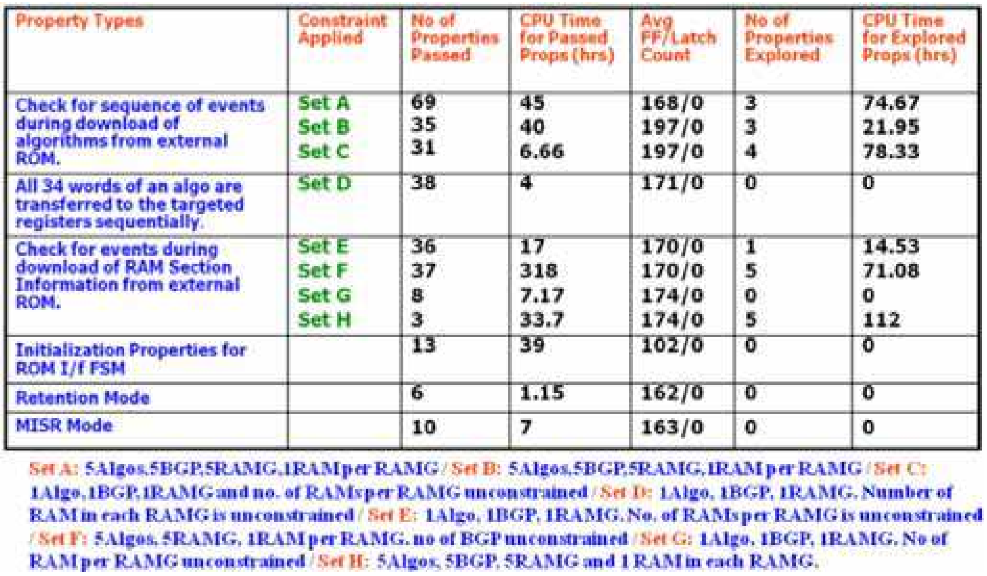
The above categories of logic are marked on a process flow diagram to ensure that none of the interface functionality is missed by the above classification. This flow diagram is shown below in Figure 11. In this figure, the verification test case which covers one of the above sub-functionality is marked in green and red, respectively. The overall coverage for each test case is captured in Table 4 below.

Verification Test Cases	Targeted Functionalities
Test Case 1	1,2,4, 6 & 7
Test Case 2	1,2,3, 5 & 6

Table 4. Functional category coverage by the different test cases.

4.4 Results from formal verification runs

The results from different formal verification runs based on the approach discussed above are shown in Table 5. A significant improvement is observed in the run-times of the different properties - many of the properties, which suffered state-space explosions earlier, converged; while, many converged properties from earlier runs report significant reduction in their run-times. We report results from IFV runs on two properties in Table 6.



Property Types	Constraint Applied	No of Properties Passed	CPU Time for Passed Props (hrs)	Avg FF/Latch Count	No of Properties Explored	CPU Time for Explored Props (hrs)
Check for sequence of events during download of algorithms from external ROM.	Set A	69	45	168/0	3	74.67
	Set B	35	40	197/0	3	21.95
	Set C	31	6.66	197/0	4	78.33
All 34 words of an algo are transferred to the targeted registers sequentially.	Set D	38	4	171/0	0	0
Check for events during download of RAM Section Information from external ROM.	Set E	36	17	170/0	1	14.53
	Set F	37	318	170/0	5	71.08
	Set G	8	7.17	174/0	0	0
	Set H	3	33.7	174/0	5	112
Initialization Properties for ROM I/f FSM		13	39	102/0	0	0
Retention Mode		6	1.15	162/0	0	0
MISR Mode		10	7	163/0	0	0

Set A: 5Algos,5BGP,5RAMG,1RAM per RAMG / Set B: 5Algos,5BGP,5RAMG,1RAM per RAMG / Set C: 1Algo,1BGP,1RAMG and no. of RAMs per RAMG unconstrained / Set D: 1Algo, 1BGP, 1RAMG, Number of RAM in each RAMG is unconstrained / Set E: 1Algo, 1BGP, 1RAMG, No. of RAMs per RAMG is unconstrained / Set F: 5Algos, 5RAMG, 1RAM per RAMG, no of BGP unconstrained / Set G: 1Algo, 1BGP, 1RAMG, No of RAM per RAMG unconstrained / Set H: 5Algos, 5BGP, 5RAMG and 1 RAM in each RAMG.

Table 5. Formal verification results from proposed approach.

Property Name	Targeted Functionality	Result before	Result After
ramgroup_start	**To check the FSM state transition and other events that are expected during the control transfer to a new RAMG**	**Passed in 12.7 hrs**	**Passed in 3.95 hrs**
ram_addr_write_str	**The last word in the RAMD for each RAM is STR. It is a mnemonic for the start instruction issued to the controller to start the memory testing. This property checks the transfer of this instruction to the corresponding register of the controller and the associated events.**	**Exploded in 12.7 hrs**	**Passed in 8.96 hrs**

Table 6. Comaprison of Formal Verification Results from different approaches.

4.5 Another useful methodology based on functional compositional verification

While the above proposed approach significantly improved the convergence of the property set needed to verify the ROM interface functional behavior with reduced runtimes, a few properties continued to suffer state-space explosions, as seen from the results presented in Table 5. Fortunately, the convergence issue related to such properties was much simpler to analyse and resolve. The simple startegy of splitting the original property into several smaller sub-properties resolved convergence issues. As an example, consider the property which verifies the sequential transfer of the first 36 words in a ALGO section, to their respective registers in the program register file of the controller. This property took 17 hours in IFV to converge. It was then split into 36 different properties, with each one dedicated to verifying just one word in the sequence of 36 words. This entire set of 36 properties took less than 8 hours to converge.

Functional behaviors involving extremely large sequential depths can pose a formidable challenge to existing automated formal verification approaches. However, analysis of such behavior usually lend themselves to prudent partitions; while these, in most cases suffer from specificity, usually result in convergence of formal verification runs on the partitioned behaviours.

5. Summary and conclusion

To summarize, the key motivation of our approach has been to automate integration verifications of IPs and DFT logic towards, 1) cycle time reduction by a factor of two in the DFT logic verification task by minimizing usage of simulation based chip level verification requirements, 2) improvement in Silicon quality by elimination of all DFT logic and its SoC integration related bugs and 3) deployment of DFT logic generation, its integration in SoC and its verification through a common infrastructure to facilitate re-use of these tasks across different SoC designs. One of the key contributions in the automation of the DFT logic verification task has been the deployment of formal verification techniques, as justified above.

Based on our experience in deploying the proposed approach, good insight has been developed into the DFT verification problem for comparison of simulation based and formal approaches. Experimental data using a commercial formal verification tool IFV [12] show that the proposed approach is an order of magnitude faster than approaches based on simulation. Though we report our results based on IFV, our approach is independent of any FV tool and can work with any FV tool which supports the Property Specification Language (PSL), or the System Verilog Assertion (SVA) language.

6. Acknowledgement

The contributions of Bijitendra Mittra, Amit Roy, Supriya Bhattacharya, Lopamudra Sen, Deepanjan Roy (all from Interra India Private Limited, Bangalore) and Abhishek Kothari (who was earlier with Interra India Private Limited, Bangalore), is gratefully acknowledged.

7. References

[1] Subir K. Roy, “Top Level SoC Interconnectivity Verification using Formal Techniques”, International Workshop on Microprocessor Test and Verification, Austin, Texas, USA, 2007.

[2] Subir K. Roy and R. A. Parekhji, “Modeling Techniques for Formal Verification of BIST Controllers and Their Integration into SoC Designs”, International Conference on VLSI Design, Bangalore, India, 2007.

[3] C. Kern and M. R. Greenstreet, "Formal Verification in Hardware Design: A Survey", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 4, April 1999, pp. 123 - 193.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, In TACAS'99*, March 1999.

[5] C.J.H. Seger and R.E. Bryant, "Formal Verification by Symbolic Evaluations of Partially – Ordered Trajectories", *Formal Methods in System Design*, 6, 147-189, 1995.

[6] S. Hazelhurst and C. J. H. Seger, "A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDDs", *IEEE Transaction on Computer Aided Design of Integrated Circuits*, 14, 4 (April 1994), 413-422.

[7] R.E. Bryant, S. German and M. N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic", *Technical Report CMU-CS-99-115*, May 1999, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[8] Yang, J. and Seger, C.: “Introduction to Generalized Symbolic Trajectory Evaluation,” *IEEE Trans. on VLSI Systems, 11(3)*, pp. 345-353, 2003.

[9] Jin Yang and C.-J. Seger, “Generalized symbolic trajectory evaluation - Abstraction in action,” *LNCS: Proc. Of FMCAD2002*, November 2002.

[10] Jin Yang, “GSTE: An illustrative and comparative introduction,” *5th International Conference on ASIC*, Volume 1, pp. 41-44, October 2003.

[11] 1-Team-Genesis – Tool for Architecture Generation, Atrenta Inc., 2008.

[12] IFV – Incisive Formal Verification Tool, Cadence Design Systems Inc., 2009.

[13] Bill Murray, “Mixing Formal and Dynamic Verification – Part 1 and 2“, Special Technology Report, SCDsource, http://www.scdsource.com/article.php?id=333 and 341, 2009.

Micro Electronic and
Mechanical Systems
INTECH