IPSJ Transactions on System LSI Design Methodology Vol. 2 131–144 (Aug. 2009)

Invited Paper

High-Level Verification

Sudipta Kundu,[†1] Sorin Lerner[†1] and Rajesh Gupta[†1]

The growth in size and heterogeneity of System-on-Chip (SOC) design makes their design process from initial specification to IC implementation complex. System-level design methods seek to combat this complexity by shifting increasing design burden to high-level languages such as SystemC and SystemVerilog. Such languages not only make a design easier to describe using high-level abstractions, but also provide a path for systematic implementation through refinement and elaboration of such descriptions. In principle, this can enable a greater exploration of design alternatives and thus better design optimization than possible using lower level design methods. To achieve these goals, however, verification capabilities that seek to validate designs at higher levels as well their equivalences with lower level implementations are crucially needed. To the extent possible given the large space of design alternatives, such validation must be *formal* to ensure the design and important properties are provably correct against various implementation choices. In this paper, we present a survey of high-level verification techniques that are used for both verification and validation of high-level designs, that is, designs modeled using high-level programming languages. These techniques include those based on model checking, theorem proving and approaches that integrate a combination of the above methods. The high-level verification approaches address verification of properties as well as equivalence checking with refined implementations. We also focus on techniques that use information from the synthesis process for improved validation. Finally, we conclude with a discussion and future research directions in this area.

1. Introduction

The quantitative changes brought about by Moore's law in design of integrated circuits (ICs) affect not only the scale of the designs, but also the scale of the process to design and validate such chips. While designer productivity has grown at an impressive rate over the past few decades, the rate of improvement has not kept pace with chip capacity growth leading to the well known design-productivity-gap [47]. The problem of reducing the design-productivity-gap is crucial in not only handling the complexity of the design, but also improving against the increased fragility of heterogeneous components that are composed in a design. Unlike software, integrated circuits are not repairable. The development costs are so high that multiple design spins are essentially ruled out, a design must be correct in the one and often the only one design iteration to implementation.

High-Level Synthesis (HLS) [30),39),59),65),85)] is often seen as a solution to bridge the design-productivity-gap. HLS is the process of generating the Register Transfer Level (RTL) design consisting of a data path and a control unit from the behavioral description of a digital system, expressed in languages like C, C++ and Java. The synthesis process consists of several inter dependent sub-tasks such as: specification, compilation, scheduling, allocation, binding and control generation. HLS is an area that has been widely explored and relatively mature implementations of various HLS algorithm have started to emerge [39),59),85)]. This shift in design paradigm enables designers to avoid many low-level design issues early in the design process. It also enables early design space exploration, and faster functional verification time. However, for verification of high-level designs, the focus so far has been on traditional testing techniques such as simulation and emulation. Over the last few decades we have seen many unfortunate examples of hardware bugs (like Pentium FDIV bug, Killer poke, and Cyrix coma bug) that have eluded testing techniques. Recently, many techniques (as discussed in this paper) inspired from *formal methods* have emerged as an alternative to ensure the correctness of these high-level designs, overcoming some of the limitations of traditional testing techniques.

The new techniques and methodology for verification and validation at higher level of abstraction are collectively called *high-level verification* techniques. The high-level verification problem can be further divided into two parts. The first part deals with verifying properties of high-level designs. The methods for verifying high-level designs allow designers to check for certain properties like functional behavior, assertion violation, and deadlock in their designs. Once the properties are checked, the designers refine their design to low-level RTL using a HLS tool. HLS tools are large and complex software systems, and as such they are prone to logical and implementation errors. Errors in these tools may lead to the synthesis of RTL designs with bugs in them. As a result, the second part deals with

†1 University of California, San Diego

verifying that the translation from high-level design to low-level RTL preserves semantics. Taken together, these two parts guarantee that properties satisfied by the high-level design are preserved through the translation to low-level RTL.

Unfortunately, despite significant amount of work in the area of formal verification we are far from being able to prove automatically that a given design always does the right thing or a given synthesis tool always produces target programs that are semantically equivalent to their source versions. However, with recent advances in SAT solvers [35),67)], automated theorem proving [36),75),76)], and model checking [14),15),82)] researchers are at least able to prove that the designs and tools satisfy some properties. Also, in many cases they are able to guarantee the functional equivalence between the initial behavioral description and the RTL output of the HLS process.

In this survey we provide an overview of the formal verification techniques used for proving the correctness of high-level designs and HLS tools. Earlier surveys on formal verification in hardware design [38),49),61)] give more comprehensive details about the theoretical and application aspects of it for RTL designs.

The remainder of the paper is organized as follows: Section 2 presents a brief overview of high-level verification and presents a classification of the techniques in this area. In Sections 3, 4, and 5 we present a survey of the different formal verification techniques used in the context of high-level verification. Finally, Section 6 concludes with a discussion and future research opportunities in this area.

2. Overview of High-Level Verification

The HLS process consists of performing stepwise transformations from a behavioral specification into a structural implementation (RTL). The main benefit of HLS is that it provides faster time to RTL and faster verification time. **Figure 1** shows the various components involved in high-level verification and how they interact. The design flow from high-level specification to RTL is shown along with various verification tasks. These tasks can be broadly classified as follows:

(1) High-level property checking
(2) Translation validation
(3) Synthesis tool verification

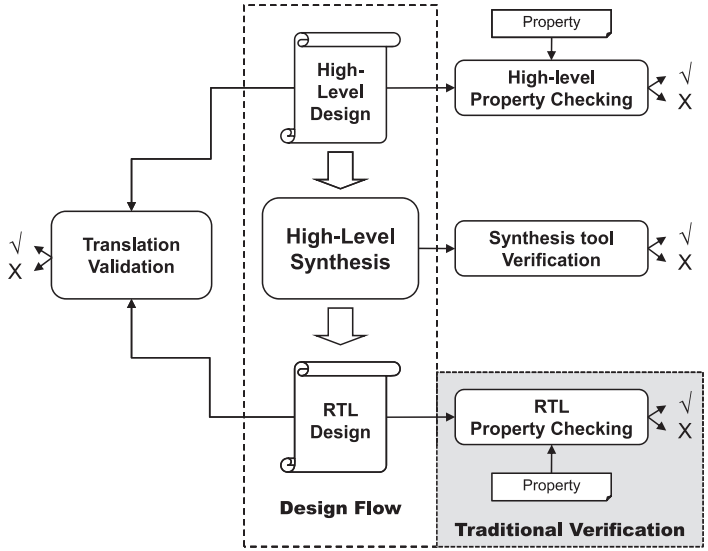

Fig. 1 Overview of high-level verification.

(4) RTL property checking

Traditionally, designers start their verification efforts directly for RTL designs. However, with the popularity of HLS, these efforts are moving more toward their high-level counterparts. This is particularly interesting because it allows faster (sometimes three orders of magnitude [83)]) functional verification time, when compared to a more detailed low-level RTL implementation. Furthermore, it allows more elaborate design space exploration, which in turn leads to better quality of design. Since RTL property checking techniques have been widely explored in earlier surveys [38),49),61)], here we focus only on the first three verification tasks.

The first category of methods, *high-level property checking*, allow various properties to be verified on the high-level designs. Once the important properties that the high-level components need to satisfy have been checked, various other techniques are used in order to prove that the translation from high-level design to low-level RTL is correct, thereby also guaranteeing that the important properties of the components are preserved.

The second category *translation validation* include techniques that try to show,

for each translation that the HLS tool performs, that the output program produced by the tool has the same behavior as the original program. Although this approach does not guarantee that the HLS tool is bug free, it does guarantee that any errors in translation will be caught when the tool runs, preventing such errors from propagating any further in the hardware fabrication process.

The third category *synthesis tool verification* consists of techniques whose goal is to prove automatically that a given optimizing HLS tool itself is correct. Although, these techniques have same goal as translation validation i.e., to guarantee that a given HLS tool produces correct result, these techniques are different because it can prove the correctness of parts of the HLS tool *once and for all*, before they are ever run.

Each one of the three areas outlined above, namely high-level property checking, translation validation, and synthesis tool verification, have been explored in a wide variety of research efforts. In the following sections we survey various techniques from each of these areas, outlining the connections and trade-offs between them.

3. High-Level Property Checking

The high-level designs written using languages like C, SystemC, SystemVerilog are mostly software programs with support for specialized hardware data types and other hardware features like synchronous concurrency, synchronization, and timing. Thus, many efforts to use software verification tools to verify these designs have been explored. Model checking is the most prevalent automatic verification technique for software and hardware. It is a technique for verifying that a hardware or software system satisfy a given property (specification). These properties, which are usually expressed in temporal logic, typically encode deadlock and safety properties (e.g., assertion violations). In this section, we survey several software model checking techniques. Model checking techniques can be further classified as *explicit* and *symbolic*.

3.1 Explicit Model Checking

In explicit state enumeration model checking, the reachable states of a design are generated using an exhaustive search algorithm. This technique explicitly stores the entire state space in memory and checks if certain error states are reachable. For finite state system this technique is both sound (i.e., whenever model checking cannot reach a given error state, it is guaranteed to not reach that error state ever in real execution) and complete (i.e., whenever model checking finds an error, it is guaranteed to be an error in real execution). However, as the size of the finite state spaces grow larger and larger, this technique suffers from the well known *state explosion* problem. To address the state explosion problem, researchers use techniques to construct the state space *on-the-fly* [44] during the search, rather than generating all the states and transitions before the search. In addition, they use *bit-state hashing* [44], in which the hash value of the reachable state is stored, instead of the state itself. Due to possible hash collision the bit-state hashing technique is unsound. Other techniques include *partial-order-reduction* [32], *symmetry reduction* [19),27] and *compositional techniques* [18].

Intuitively, the partial-order-reduction technique exploits the independence between parallel threads to compute a subset of the enabled transitions in each visited states. Next, if a selective search is done using only the transitions from these subsets the detection of all the deadlocks and safety property violations are guaranteed. Symmetry reduction on the other hand exploits symmetries in the program, and explores one element from each symmetry class. Compositional techniques decompose the original verification problem to related smaller problems such that the result of the original problem can be obtained by combining the smaller ones.

The most popular finite state explicit model checker for concurrent programs are SPIN [44] and MURPHI [24]. Both tools have been successfully used for verification of sequential circuits and protocols.

Moreover, in order to achieve scalability some systems give up completeness of the search and focus on the bug finding capabilities of model checking. For instance, one can bound the depth of the search and/or bound the number of context switches [70]. This line of thought also leads to the execution-based model checking approach. These methods are typically used for improving the coverage of a test. Traditionally, in testing the user writes a test bench and runs it. Typically, the operating system scheduler executes only one fixed schedule out of the many possible behaviors. However, the scheduler of the execution based model checker systematically explore all possible behaviors of the program for a

given test input and depth. The most striking benefit of this approach is the ease of implementing it, as it sidesteps the need to formally represent the semantics of the programming language as a transition relation. Another key aspect of this method is the idea of *stateless* [33] search i.e., it stores no state representations in memory but only information about which transitions have been executed so far. Although stateless search reduces the storage requirements, a significant challenge for this approach is how to handle the exponential number of paths in the program. Here again various reduction techniques like symmetry, partial-order [29], and abstraction have been explored.

Verisoft [33] is the first tool in this domain, and it explores arbitrary code written in full fledged programming language like C or C++. It does so by modifying the OS scheduler and systematically exploring all possible interleavings. Java PathFinder [84] is a tool for Java programs that uses the virtual machine rather than OS scheduler to explore the different behaviors. CMC [69] is another tool for C programs that improves the efficiency of the search by storing a hash of each visited state. Dynamic validation using execution-style model checking is also well adapted for validating SystemC designs [41),51].

3.2 Symbolic Model Checking

The above reduction techniques like partial-order, address the state explosion problem for asynchronous concurrent systems (by reducing the number of interleavings that need to be explored). However, they are not so effective in the case of synchronous concurrent systems, which do not involve interleaving. Symbolic model checking techniques, on the other hand, are quite effective for both synchronous and asynchronous concurrent systems. Furthermore, the reduction techniques discussed in Section 3.1 including partial-order reduction are orthogonal and can be used in conjunction with symbolic techniques.

Symbolic algorithms manipulate sets of states, instead of individual states. These algorithms avoid ever building the graph for the system; instead, they represent the graph implicitly using a formula in propositional logic. They can also represent infinite states using a single formula. For example the predicate $(x > 1 \wedge y > 1)$ denotes the set of all states in which the value of the variables x and y are both greater than 1. The first major step toward symbolic representation is the use of *Binary Decision Diagrams* (BDD) [13]. BDDs are a canonical form representation for boolean formulas, and are particularly important for finite state programs, as these programs can be represented using boolean variables. BDDs are used in symbolic model checker like SMV [63] and have been instrumental in verifying hardware designs with very large state spaces [14].

As in explicit model checking, one sometimes trades off completeness for bug finding capabilities of symbolic model checking. *Bounded Model Checking* (BMC) [10] is one such algorithm that unroll the control flow graph (loop) for a fixed number of steps (say k), and check whether a property violation can occur in k or fewer steps. This typically involves encoding the restricted model as an instance of Satisfiability (SAT) problem. This problem is then solved using a SAT [67] or SMT (Satisfiability Modulo Theory) [68] solver. BMC tools like CBMC [17] and FSoft-BMC [45] use iterative deepening depth-first search so that the above process can be repeated with larger and larger values of k until all possible violations have been ruled out.

Another area that has recently received lot of attention is *abstract model checking*, which trades off precision for efficiency. Abstraction [21),22)] attempts to prove properties of a program by first simplifying it. Next, the reachability analysis is performed on the simplified (or abstract) domain, which usually satisfies some, but not all the properties of the original (or concrete) program. Generally, one requires the abstract domain and its semantics to be sound (i.e., the properties proved in the abstract semantics implies properties in the concrete semantics). However, typically, the abstraction is not complete (i.e., not all true properties in the concrete semantics are true in the abstract semantics). An example of abstraction is, to only consider boolean variables and the control flow of a program and ignore the values of non boolean variables. Although, such an abstraction may appear coarse, it is sometimes sufficient to prove properties like mutual exclusion.

The polyhedral abstract domain has been successfully used to check for array bounds violations [20]. Another interesting domain, *predicate abstraction* [6),23),37),54)] is parameterized by a fixed finite set $\mathcal{B} = \{B_1, B_2, \cdots, B_k\}$ of first-order formulas (predicates) over the program variables, and consists of the lattice of Boolean formulas over $\mathcal{B}$ ordered by implication. A cube over $\mathcal{B}$ is a conjunction of possibly negated predicates from $\mathcal{B}$. The domain of predicate

abstraction is the set of all cubes, and one cube is computed at each program point.

The goal of predicate abstraction is to compute a set of predicates from $\mathcal{B}$ at every program point. Thus, given a set of predicates $\mathcal{B}$, a program statement s, and a cube over $\mathcal{B}$ flowing into the statement, it computes the cube over $\mathcal{B}$ that flows out of the statement. For example, consider the set of predicates $\mathcal{B} = \{B_1, B_2\}$, where $B_1 \equiv (a = b)$ and $B_2 \equiv (a = b+1)$. Given the cube $B_1 \wedge \neg B_2$ and the statement $a := b+1$, then predicate abstraction would compute that the cube $\neg B_1 \wedge B_2$ should be propagated after the statement.

A problem with abstract model checking is that although the abstraction simulates the concrete program, when the abstraction does not satisfy a property, it does not mean that this property actually fails in the concrete program. When a property fails, the model checker produces a *counterexample*. A counterexample can be *genuine* i.e., can be reproduced on the concrete program, or *spurious* i.e., does not correspond to a real computation but arises due to imprecisions in the analysis. Counterexamples are checked against the real state space to make sure they are genuine. In the case when it is spurious, methods have been developed to automatically refine the abstract domain and get a more precise analysis which rules out the current counterexample and possibly many others, without losing soundness. This iterative strategy is called *Counter Example Guided Abstraction Refinement* (CEGAR).

SLAM [6] is a popular CEGAR based model checker for C programs. It was used successfully within Microsoft for device driver verification [7] and has been developed into a commercial product (Static Driver Verifier, SDV). BLAST [9] is also a CEGAR based model checker that uses *lazy abstraction* [42]. The main idea of BLAST is the observation that the computationally intensive steps of abstraction and refinement can be optimized by a tighter integration which would allow it to reuse the work performed in one iteration toward subsequent iterations. Lazy abstraction tightly couples abstraction and refinement by constructing the abstract model on-the-fly, and locally refining the model on-demand. MAGIC [16] is another CEGAR based compositional model checking framework for concurrent C programs. Using MAGIC, the problem of verifying a large implementation can be naturally decomposed into the verification of a number of smaller, more manageable fragments. These fragments can be verified separately, enabling MAGIC to scale up to industrial size programs.

Advances in model checking and related techniques in the past several decades have allowed researchers to verify increasingly ambitious properties of critical software programs including device drivers, operating systems code, and large commercial applications. They have also enabled the verification of large hardware components like microprocessors. Although this is a significant step forward toward reducing the design-productivity-gap, state-of-the-art verification techniques are still far away from proving full correctness of programs.

4. Translation Validation

Once the design has been checked to satisfy certain properties using techniques discussed in Section 3, the next step is to make sure that those properties are preserved through the synthesis process. In this section we discuss a category of methods called translation validation which guarantee the preservation of safety properties through the synthesis process. Translation validation techniques are employed during synthesis to check that each transformation performed by the HLS tool preserves the semantics of the initial design. The initial design is called specification and the transformed design is called implementation. The validation step check for either *refinement* or *equivalence*. Typically, the implementation is said to be a refinement of the specification if the set of execution traces of the implementation is a subset of the set of execution traces of the specification. They are equivalent when the two sets are equal. In this section, we discuss different techniques for translation validation. Depending upon the core approach these techniques are primarily based on, they are divided into three categories: relational approach, model checking, and theorem proving.

4.1 Relational Approach

Relational approaches [12),25),48),50] are used to check the correctness of the synthesis process by establishing a functional equivalence between the Control-Data Flow Graphs (CDFG) of the program, before and after each step of HLS. The equivalence is defined on some predefined *observable events* that are preserved across the transformations. Intuitively, the idea is to show that there exists a *simulation relation* R that matches a given program state in the implementa-

tion with the corresponding state in the specification. This simulation relation guarantees that for each execution sequence of observable events in the implementation, a related and equivalent execution sequence exists in the specification. The relation $R \subseteq State_1 \times State_2$ operates over the program states $State_1$ of the specification and the program states $State_2$ of the implementation. If $Start_1$ is the set of start states of the specification, $Start_2$ is the set of start states of the implementation, and $\sigma \rightarrow^e \sigma'$ denotes state σ stepping to state σ' with observable event e, then the following conditions summarize the requirements for a correct refinement:

$$\forall \sigma_2 \in Start_2 \;.\; \exists \sigma_1 \in Start_1 \;.\; R(\sigma_1, \sigma_2)$$
$$\forall \sigma_1 \in State_1, \sigma_2 \in State_2, \sigma_2' \in State_2 \;.$$
$$\sigma_2 \rightarrow^e \sigma_2' \wedge R(\sigma_1, \sigma_2) \Rightarrow$$
$$\exists \sigma_1' \in State_1 \;.\; \sigma_1 \rightarrow^e \sigma_1' \wedge R(\sigma_1', \sigma_2')$$

These conditions respectively state that:

(1) For each starting state in the implementation, there must be a related state in the specification.

(2) If the specification and the implementation are in a pair of related states, and the implementation can proceed to produce observable events e, then the specification must also be able to proceed, producing the same events e, and the two resulting states must be related.

The above conditions are the base case and the inductive case of a proof by induction showing that the implementation is a refinement of the specification.

One example of using the relational approach is Karfa, et al.'s technique [48] for establishing the equivalence between the initial Finite State Machine with Datapath (FSMD) and the scheduled FSMD. The technique introduces cut-points in the original and transformed FSMD automatons, which allows computations through the original and transformed FSMD to be seen as the concatenation of paths from cut-points to cut-points. The technique then establishes the equivalence by exploiting the structural similarities between related cut-points using weakest pre-condition.

Another example of the relational approach can be found in Dushina, et al.'s proposed method [25] for checking the functional equivalence between a scheduled abstract FSM and the corresponding RTL after binding. The method establishes the equivalence transition by transition. In particular, for each transition in the RTL controller, it performs a symbolic execution of the associated RTL data path. The symbolic execution results are then syntactically compared with the data operations specified in the equivalent transition of the abstract FSM.

In our own work [52),53)], we have developed an automatic algorithm based on the relational approach, and implemented it within the SPARK [39] HLS tool, which is a parallelizing HLS framework that does code motion across basic blocks. Our algorithm validates all the phases (except for parsing, binding and code generation) of SPARK against the initial behavioral description. Unlike previous methods, which assume that the scheduler does not move code across basic blocks and variable names do not change, our technique can handle these features fully automatically.

In general, relational approaches work well when the transformations preserve most of the program's control flow structure. Such transformations are called *structure-preserving* [86] transformations. Unfortunately, relational approaches tend to be ineffective in the face of non structure-preserving transformations like loop unrolling, loop tiling and loop reordering. Despite these limitations, relational approaches are very useful in practice: with only a fraction of the development cost of an HLS tool, they can uncover bugs that elude testing.

4.2 Model Checking

Techniques involving model checking [5),11)] are used for verifying register-transfer logic against its scheduled behavior. The key idea is to partition the equivalence checking task into two simpler subtasks, verifying the validity of register sharing/binding, and verifying correct synthesis of the RTL interconnect and control. The success of these methods can be attributed to the observation that the state space explosion in most designs is caused by the data path registers rather than the number of control states.

The following algorithm outlines the method of verifying the validity of register sharing.

- The first step involves identifying paths in the scheduled graph along which potential conflicts can occur. During this step no interpretation of the data path is done. If no conflict is identified then verification successfully termi-

nates.

- Otherwise for each violation the set of all conflict paths is summarized in a reduced Conflict Sub-Graph (CSG).
- The reduced CSG is then checked to find out if the conflict was benign. If a conflict is detected during the checking, then a logically possible path with incorrect register binding has been detected. In this case the appropriate path is shown to the user as a counterexample.
- Else, if it ends without any conflict detected, all possible conflict paths are logically impossible and the verification algorithm successfully terminates.

Ashar, et al. [5] analyzed potential conflicts by means of structural methods, and then the reduced CSG is checked for satisfiability by the VIS model checker [82]. Whereas Blank [11] identifies possible conflicts using a symbolic model checker [14]. The result of the analysis is summarized in a reduced internal representation called Language of Labeled Segments (LLS) [43], which is then checked by symbolic simulation [66]. However, symbolic simulation allows reasoning for a defined finite number of steps. Thus, loops in the program cannot be verified for an arbitrary number of iterations.

Ashar, et al. also presented an algorithm to verify the correct synthesis of the RTL interconnect and control [5]. This part of equivalence checking is done state-by-state, i.e., for each state in the schedule, the computations performed in that state are shown to be equivalent to those performed in the RTL implementation for the same state. The equivalence is shown using symbolic simulation.

4.3 Theorem Proving

Although most of the translation validation approaches discussed so far use theorem provers in some way, the theorem prover is not at the center of the approach. The Correctness Condition Generator [60], on the other hand, is primarily based on a theorem proving technique. This approach assumes that the synthesis tool can identify the binding relation between specification variables and registers in the RTL design, and between the states in the behavior and the corresponding states in the RTL design. A correctness condition generator is tightly integrated with the high-level synthesis tool to automatically generate (1) formal specifications of the behavior and the RTL design including the data path and the controller, (2) the correctness lemmas establishing equivalence between the synthesized RTL design and its behavioral specification, and (3) proof scripts for these lemmas that can be submitted to a higher-order logic proof checker without further human interaction. The tight integration of the synthesis process with the theorem prover allows the theorem prover to gather information about what kinds of transformations were performed, and therefore better reason about them.

5. Synthesis Tool Verification

Another attractive way of proving that an HLS tool produces correct RTL is to verify the correctness of HLS tool once and for all, before it is ever run once.

One can categorize such techniques into three broad classes: (1) *formal assertions*, which can be used to guarantee the correctness of the synthesis tool, (2) *transformational synthesis tools*, which are correct by construction, and (3) *witness generators*, which recreate the steps that an existing HLS tool has performed using formally verified transformations.

5.1 Formal Assertions

Narasimhan, et al. proposed a Formal Assertions approach [71)–73)] to building a verified high-level synthesis system, called *Asserta*. The approach works under the following premise: If each stage in the system, like scheduling, register optimization, interconnect optimization etc. can be verified to perform correct transformations on the input specification, then by compositionality, we can assert that the resulting RTL design is equivalent to its input specification. This technique has the following four main steps.

(1) *Characterization*: A base specification model is identified for each synthesis task. The base specification model is usually a tight set of correctness properties that completely characterizes the synthesis task.

(2) *Formalization*: The base specification model is then formalized as a collection of theorems in a higher order logic theorem proving environment, which form the base formal assertions. An algorithm is also chosen to realize the corresponding synthesis task and is described in the same formal environment.

(3) *Verification*: The formal description of the algorithm is verified against the base theorems. Inconsistencies in the base model are identified during the

verification exercise. Furthermore, the model is enhanced with several additional formal assertions derived during verification. The formal assertions now represent the invariants in the algorithm.

(4) *Formal Assertions Embedding*: In the next step a software implementation of the algorithm that was formally verified in the previous stage is developed. The much enhanced and formally verified set of formal assertions is then embedded within this software implementation as program assertions.

During synthesis, the implementation of each task is continually evaluated against its specification model specified by these assertions and any design error during synthesis can be detected.

Asserta [71] is a high-level synthesis system developed to show the effectiveness of assertion-based verification techniques to generating first-time correct RTL designs. The synthesis engine has three main stages, namely scheduling, register optimization and interconnect generation. The proof effort was conducted using Prototype Verification System (PVS) [75], a higher order logic theorem prover.

Since the main tasks of Asserta have been verified, it can be used with an increased degree of confidence. This approach is also not affected by the state space or complexities of any synthesized RTL design. However, the correctness of the system depends on the completeness and correctness of the base assertions. Another concern is that during the formal assertions embedding step, due to difference in the expressive power of logic and software program, the translation process often could get quite complicated and finally, the correctness of the method hinges on this translation process. It is also hard to generate a tight base specification for all the steps of the synthesis process. Thus, although Asserta is a first step toward achieving correct synthesis, verifying large synthesis programs is quite tedious and complex.

5.2 Transformational Synthesis Tools

The basic idea of this method is to determine a set of transformations, which when applied to an initial specification, transform the source into the required implementation. This transformations are then embedded in a theorem prover to prove their correctness. The correctness of the HLS system thus follows from a 'correct by construction' argument.

Transformational synthesis is an area that has been widely explored [26),40),46),55),79),81)]. Various tools have been developed in the recent past, which mainly differ in the expressiveness of their input language, the theorem prover used and the type of transformations allowed. Sharp, et al. [81] developed the T-Ruby design system, where the Ruby language is used for specifying VLSI circuits and the theorem prover Isabelle [76] is used to formalize the correctness-preserving transformations. DDD [46] is another system, which is a based on functional algebra. Both systems uses hardware specific calculus to describe a design. The following are few systems based on behavioral transformations. Veritas [40] is a theorem prover based on an extension of typed higher order logic, which provides an interactive environment for converting the specification into an implementation. Larsson [55] presented a transformational approach to digital system design based on the HOL proof system [36]. Hash [26] is another system based on the theorem prover HOL [36]. McFarland [62] investigated the correctness of behavioral transformations using behavior expressions. Rajan [79] on the other hand, used the PVS [75] theorem proving system to specify and verify behavioral transformations.

However, unlike the formal assertion technique presented in Section 5.1, techniques based on transformational synthesis reason only about the specification of the transformations, not their software implementations, which is where many of the bugs arise.

5.3 Witness Generator

The main idea behind witness generator techniques [28),64),78)] is to use a set of behavior-preserving elementary transformations for validating an existing non transformational synthesis system by discovering and to some extent isolating software errors.

Figure 2 shows an overview of the witness generator approach. The source program is first converted into a CDFG, after which point the CDFG passes through a regular unverified HLS process. Following the regular HLS process, the CDFG is also passed to a transformational system that consists of a set of elementary structural transformations, all of which have been formally verified given a set of preconditions. These transformations are sequenced together by the witness generator, whose goal is to find a sequence of elementary transformations which when applied to the initial design, achieve the same RTL outcome. For

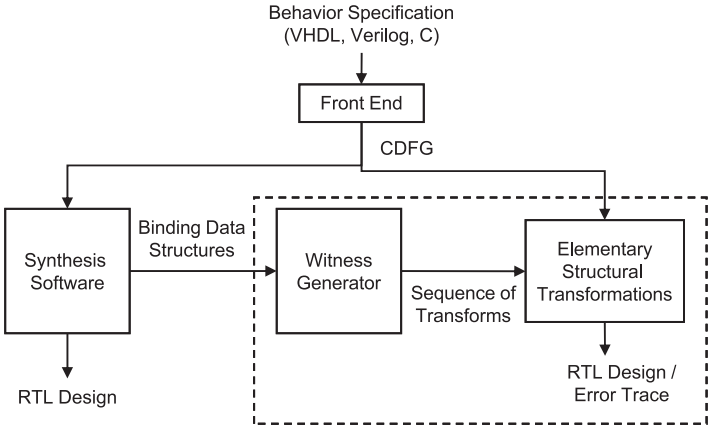

Fig. 2 Using a witness generator system to validate synthesis tools.

this technique to be broadly applicable, the set of elementary transformations must collectively capture a wide variety of synthesis algorithms.

The task of the witness generator is facilitated by the following information, which is provided by the synthesis tool:

- The outcome of a synthesis task can be captured by a simple data structure (binding data structure) such that any algorithm for this task can record its outcome in this data structure. For example, the outcome of any scheduling algorithm can be recorded as a schedule table which records the mapping between operations to control steps and the outcome of any register allocation algorithm can be recorded as mapping from variables to registers.
- It is possible to generate a sequence of elementary transformations to perform the same task by examining this data structure, without any knowledge of the synthesis algorithm used to perform the task.

When a precondition fails during the execution of the sequence of transformations identified by the witness generator, the sequence applied so far forms a counter-example that can be presented to the user.

Radhakrishnan, et al.[78] identified a set of six elementary transformations which were sufficient to emulate the effect of many existing high-level synthesis algorithms. Each of these transformations is mechanically proved in PVS[75] to preserve the computational behavior.

Eveking, et al.[28] uses a similar approach to verify the correctness of various scheduling algorithms. They represented the initial CDFG using the LLS[43] internal language. After that, the process of equivalence verification consists of a number of computationally equivalent LLS transformation steps which *assimilate* the original design to the scheduled design.

Recently, Mendías, et al.[64] used equational specification to describe behaviors and/or structures, in a elaborate formal framework called *Fresh*. In Fresh, seven formal derivation rules, classified into structural and behavioral were used to transform the initial design to the RTL design.

The above systems essentially recreate, within a formal framework, each of the design decisions taken by an external (and potentially incorrect) HLS algorithm. The latest HLS tools are complex and use a variety of transformations to optimize the synthesis result for metrics like area, performance and power. As a result, it is becoming increasingly difficult to find a small set of correct transformations that can recreate all the design decisions taken by external HLS tools.

6. Discussion and Future Directions

The last decade witnessed great improvements in formal methods and HLS. Recently, many commercial formal verification tools for system-level designs, like Static Driver Verifier (SDV)[3], the Sequential Equivalence Checking tool (SLEC)[1], SCADE Design Verifier[2], and Statemate[4] have become available. However, their adoption is in the early stages and the tools are often limited in the quality of the results and the kinds of correctness guarantees that are provided. By performing verification on the high-level design, where verification is easier to perform, and then checking that all refinement steps are correct, high-level verification can provide strong and expressive guarantees that would have been difficult to achieve by directly analyzing the low-level RTL code. In this section we discuss promising future research areas in verification of high-level designs and the tools associated with them.

Hardware-Software modeling: High-level hardware languages support many features that are useful for both software and hardware designs. For example, SystemC allows both asynchronous and synchronous semantics of concurrency, and

also both software and hardware data types. However, existing tools often either target software or hardware. For example, most software model checkers only support software data types and asynchronous semantics of concurrency, and most hardware model checker only support hardware data types and synchronous semantics of concurrency. As a result, researchers often use abstraction or complicated techniques while modeling the non-supported features of a given model checker. This gap points to a possible research direction that would unify techniques for hardware models and techniques for software models into combined methodology for reasoning about hardware-software models.

Compositional techniques: Although many techniques presented in this paper use compositional methods to make the verification problem tractable, these techniques are far too limited in their application. Even after decomposition using the current techniques the problem is still quite large and complex. Advanced and more efficient methods are needed for decomposing a computationally demanding global property into local properties whose verification is simpler.

Modular techniques: A major issue is that most of these verification methodologies are hard and inconvenient to use, and even more harder to modify or extend. Hence, every time a new methodology is proposed a new tool has to be written, often from scratch. None of the formal methods discussed in this paper are designed from a software engineering perspective. Thus, there is a need for a modular and reusable framework, which can be quickly used to prototype new ideas and test them.

Debugging: Most verification tools surveyed in this paper provide only limited feedback to the user. When a bug is found, these tools cannot typically pin-point the error in the code. Some methods are able to output an error trace, but figuring out the cause of the error from it, is not straightforward and requires expertise in formal methods. Although there has been work in this area [8], adapting such techniques to the high-level verification domain is still a challenge. Another limitation of current methods is that they often stop searching when a bug is found, rather than providing a list of all bugs. More broadly, the goal should be to fit formal verification into the regular develop-edit-debug flow, which would require the development of verification tools for speed and ease of use.

Synthesis-For-Verification: HLS process focuses mainly on three design constraints: area, timing and power. These methodologies tend to ignore verification, which takes about 70% of the design cycle, as a constraint. Recently, Ganai, et al. [31] proposed a new paradigm 'Synthesis-For-Verification' which involves synthesizing "verification-aware" designs that are more suitable for functional verification. Therefore, another research direction may be to use existing infrastructure of HLS to generate "verification friendly" models that are relatively easier to verify using state-of-the-art techniques.

Compiler techniques: Many techniques used in HLS are similar to those used for compilers. As a result, advances in fields like compiler correctness can provide inspiration for developing techniques for high-level verification. For example, translation validation [34),74),77),80),86)] has been successfully used to prove the correctness of many compiler transformations. Necula's translation validation technique [74)] automatically proved many structure preserving transformation used in GCC, whereas Zuck, et al. [34)] proved various non-structure preserving transformation, along with structure preserving transformation. Leroy reported [58)], his efforts on implementing a provably-correct compiler for a subset of C. Lerner, et al. [56),57)] presented ways to automatically prove the correctness of compiler optimizations once and for all. All these techniques have been successfully applied to the compiler domain, and can provide new directions for verification of the HLS process.

References

1) Calypto's SLEC System. www.calypto.com/slecsystem.php
2) Esterel's SCADE Design Verifier. www.esterel-technologies.com/products/scade-suite/design-verifier
3) Microsoft's Static Driver Verifier (SDV). www.microsoft.com/whdc/devtools/tools/SDV.mspx
4) Telelogic's Statemate. modeling.telelogic.com/products/statemate/index.cfm
5) Ashar, P., Bhattacharya, S., Raghunathan, A. and Mukaiyama, A.: Verification of RTL generated from scheduled behavior in a high-level synthesis flow, *ICCAD*, pp.517–524 (1998).
6) Ball, T., Majumdar, R., Millstein, T. and Rajamani, S.: Automatic Predicate Abstraction of C Programs, *PLDI '01: Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2001).
7) Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C.,

Ondrusek, B., Rajamani, S.K. and Ustuner, A.: Thorough static analysis of device drivers, *EuroSys '06: Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pp.73–85, ACM Press, New York, NY (2006).
8) Ball, T., Naik, M. and Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces, *Principles of Programming Languages*, pp.97–105 (2003).
9) Beyer, D., Henzinger, T.A., Jhala, R. and Majumdar, R.: The software model checker Blast: Applications to software engineering, *Int. J. Softw. Tools Technol. Transf.*, Vol.9, No.5, pp.505–525 (2007).
10) Biere, A., Cimatti, A., Clarke, E.M., Fujita, M. and Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs, *DAC '99: Proc. 36th ACM/IEEE conference on Design automation*, New York, NY, USA, pp.317–320, ACM (1999).
11) Blank, C.: Formal Verification of Register Binding, *WAVE '00: Proc. Workshop on Advances in Verification* (2000).
12) Borrione, D., Dushina, J. and Pierre, L.: A compositional model for the functional verification of high-level synthesis results, *IEEE Trans. Very Large Scale Integr. Syst.*, Vol.8, No.5, pp.526–530 (2000).
13) Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams, *ACM Computing Surveys*, Vol.24, No.3, pp.293–318 (1992).
14) Burch, J., Clarke, E., McMillan, K., Dill, D. and Hwang, L.: Symbolic Model Checking: 10^{20} States and Beyond, *Proc. 5th Annual IEEE Symposium on Logic in Computer Science*, Washington, D.C., pp.1–33, IEEE Computer Society Press (1990).
15) Chaki, S., Ouaknine, J., Yorav, K. and Clarke, E.: Automated compositional abstraction refinement for concurrent C programs: A two-level approach, *Proc. Workshop on Software Model Checking* (*SoftMC*), ENTCS, Vol.89 (2003).
16) Chaki, S., Clarke, E. and Groce, A.: Modular verification of software components in C, *IEEE Trans. Softw. Eng.*, pp.388–402 (2004).
17) Clarke, E., Kroening, D. and Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking, *DAC '03: Proc. 40th Conference on Design Automation*, pp.368–371, ACM Press New York, NY (2003).
18) Clarke, E., Long, D.E. and McMillan, K.L.: Compositional Model Checking, Technical Report CMS-CS-89-145, School of Computer Science, Carnegie Mellon University (1989).
19) Ip, C.N. and Dill, D.L.: Better Verification Through Symmetry, *Computer Hardware Description Languages and their Applications*, Agnew, D., Claesen, L. and Camposano, R. (eds.), Ottawa, Canada, pp.87–100, Elsevier Science Publishers B.V., Amsterdam, Netherland (1993).
20) Cousot, P. and Halbwachs, N.: Automatic discovery of linear restraints among variables of a program, *Conference Record of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Tucson, Arizona, pp.84–97, ACM Press, New York, NY (1978).
21) Cousot, P. and Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *Proc. 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, pp.238–252 (1977).
22) Cousot, P. and Cousot, R.: Systematic Design of Program Transformation Frameworks by Abstract Interpretation, *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, OR (2002).
23) Das, S., Dill, D.L. and Park, S.: Experience with Predicate Abstraction, *CAV '99: Proc. 11th International Conference on Computer Aided Verification*, London, UK, pp.160–171, Springer-Verlag (1999).
24) Dill, D.L.: The Murphi Verification System, *CAV '96: Proc. 8th International Conference on Computer Aided Verification*, London, UK, pp.390–393, Springer-Verlag (1996).
25) Dushina, J., Borrione, D. and Jerraya, A.A.: Formal Verification of the Allocation Step in High Level Synthesis, *Forum on Design Languages* (*FDL'98*), Lausanne, Switzerland (1998).
26) Eisenbiegler, D., Blumenröhr, C. and Kumar, R.: Implementation Issues about the Embedding of Existing High Level Synthesis Algorithms in HOL, *TPHOLs '96: Proc. 9th International Conference on Theorem Proving in Higher Order Logics*, London, UK, pp.157–172, Springer-Verlag (1996).
27) Emerson, F.A. and Sistla, A.P.: Symmetry and Model Checking, *Form. Methods Syst. Des.*, Vol.9, No.1/2, pp.105–131 (1996).
28) Eveking, H., Hinrichsen, H. and Ritter, G.: Automatic verification of scheduling results in high-level synthesis, *DATE '99: Proc. Conference on Design, Automation and Test in Europe*, New York, NY, USA, p.12, ACM Press (1999).
29) Flanagan, C. and Godefroid, P.: Dynamic partial-order reduction for model checking software, *Proc. 32nd ACM Symposium on Principles of Programming Languages* (2005).
30) Gajski, D.D. and Ramachandran, L.: Introduction to High-Level Synthesis, *IEEE Des. Test*, Vol.11, No.4, pp.44–54 (1994).
31) Ganai, M.K., Mukaiyama, A., Gupta, A. and Wakabayshi, K.: Synthesizing "Verification Aware" Models: Why and How?, *VLSI Design '07: 20th International Conference on VLSI Design*, pp.50–56 (2007).
32) Godefroid, P.: Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem, Ph.D. Thesis, Univerite De Liege (1995).
33) Godefroid, P.: Model Checking for Programming Languages Using VeriSoft, *Proc. POPL* (1997).
34) Goldberg, B., Zuck, L. and Barrett, C.: Into the Loops: Practical Issues in Translation Validation for Optimizing Compilers, *Electronic Notes in Theoretical Computer Science*, Vol.132, No.1, pp.53–71 (2005).
35) Goldberg, E. and Novikov, Y.: BerkMin: A fast and robust Sat-solver, *Discrete*

Applied Mathematics, Vol.155, No.12, pp.1549–1561 (2007).
36) Gordon, M.: HOL: A proof generating system for higher-order logic, *VLSI Specification Verification and Synthesis*, Birtwistle, G. and Subrahmanyam, P. (eds.), pp.73–128, Kluwer Academic Publishers (1988).
37) Graf, S. and Saidi, H.: Construction of Abstract State Graphs of Infinite Systems with PVS, *CAV '97: Proc. International Conference on Computer Aided Verification* (1997).
38) Gupta, A.: Formal hardware verification methods: A survey, *Form. Methods Syst. Des.*, Vol.1, No.2-3, pp.151–238 (1992).
39) Gupta, S., Dutt, N., Gupta, R. and Nicolau, A.: SPARK: A high-level synthesis framework for applying parallelizing compiler transformations, *International Conference on VLSI Design* (2003).
40) Hanna, F.K., Daeche, N. and Longley, M.: Formal Synthesis of Digital Systems, *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Claesen, L. (ed.), pp.532–548, Elsevier, Leuven, Belgium (1989).
41) Helmstetter, C., Maraninchi, F., Maillet-Contoz, L. and Moy, M.: Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip, *FMCAD '06: Proc. Formal Methods in Computer Aided Design*, Washington, D.C., USA, pp.171–178, IEEE Computer Society (2006).
42) Henzinger, T.A., Jhala, R., Majumdar, R. and Sutre, G.: Lazy Abstraction, *Proc. 29th ACM Symposium on Principles of Programming Languages* (2002).
43) Hinrichsen, H.: Language of Labelled Segments documentation, Technical report, Darmstadt University of Technology (1998).
44) Holzmann, G.J.: The Model Checker SPIN, *Software Engineering*, Vol.23, No.5, pp.279–295 (1997).
45) Ivanicic, F., Yang, Z., Ganai, M.K., Gupta, A. and Ashar, P.: Efficient SAT-based bounded model checking for software verification, *Theor. Comput. Sci.*, Vol.404, No.3, pp.256–274 (2008).
46) Johnson, S. and Bose, B.: DDD — A System for Mechanized Digital Design Derivation, *ACM/SIGDA Workshop on Formal Methods in VLSI Design*, Miami, Florida (1991).
47) Kahng, A.B.: Design technology productivity in the DSM era (invited talk), *ASP-DAC '01: Proc. 2001 Conference on Asia South Pacific Design Automation*, New York, NY, USA, pp.443–448, ACM (2001).
48) Karfa, C., Mandal, C., Sarkar, D., Pentakota, S.R. and Reade, C.: A Formal Verification Method of Scheduling in High-level Synthesis, *IEEE International Symposium on Quality Electronic Design*, pp.71–78 (2006).
49) Kern, C. and Greenstreet, M.R.: Formal verification in hardware design: A survey, *ACM Trans. Des. Autom. Electron. Syst.*, Vol.4, No.2, pp.123–193 (1999).
50) Kim, Y., Kopuri, S. and Mansouri, N.: Automated Formal Verification of Scheduling Process Using Finite State Machines with Datapath (FSMD), *ISQED '04: Proc. 5th International Symposium on Quality Electronic Design*, Washington, D.C., USA, pp.110–115, IEEE Computer Society (2004).
51) Kundu, S., Ganai, M.K. and Gupta, R.: Partial Order Reduction for Scalable Testing of SystemC TLM Designs, *DAC '08: Proc. 45th Annual Conference on Design Automation*, New York, NY, USA, pp.936–941, ACM (2008).
52) Kundu, S., Lerner, S. and Gupta, R.: Automated Refinement Checking of Concurrent Systems, *ICCAD '07: Proc. 2007 IEEE/ACM International Conference on Computer-Aided Design*, Piscataway, NJ, USA, pp.318–325, IEEE Press (2007).
53) Kundu, S., Lerner, S. and Gupta, R.: Validating High-Level Synthesis, *CAV '08: Proc. 20th International Conference on Computer Aided Verification*, Princeton, NJ, USA, pp.459–472, Springer (2008).
54) Lahiri, S.K., Ball, T. and Cook, B.: Predicate Abstraction via Symbolic Decision Procedures, *Computer Aided Verification*, pp.24–38 (2005).
55) Larsson, M.: Improving the Result of High-Level Synthesis Using Interactive Transformational Design, *TPHOLs '96: Proc. 9th International Conference on Theorem Proving in Higher Order Logics*, London, UK, pp.299–314, Springer-Verlag (1996).
56) Lerner, S., Millstein, T. and Chambers, C.: Automatically Proving the Correctness of Compiler Optimizations, *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, CA (2003).
57) Lerner, S., Millstein, T., Rice, E. and Chambers, C.: Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules, *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Long Beach, CA (2005).
58) Leroy, X.: Formal Certification of a compiler back-end or: Programming a compiler with a proof assistant, *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, pp.42–54, ACM Press (2006).
59) Lin, Y.-L.: Recent developments in high-level synthesis, *ACM Trans. Design Automation of Electronic Systems.*, Vol.2, No.1, pp.2–21 (1997).
60) Mansouri, N. and Vemuri, R.: Automated Correctness Condition Generation for Formal Verification of Synthesized RTL Designs, *Form. Methods Syst. Des.*, Vol.16, No.1, pp.59–91 (2000).
61) McFarland, M.C.: Formal verification of sequential hardware: A tutorial, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.12, No.5, pp.654–663 (1993).
62) McFarland, M.C.: Formal analysis of correctness of behavioral transformations, *Form. Methods Syst. Des.*, Vol.2, No.3, pp.231–257 (1993).
63) McMillan, K.L.: A methodology for hardware verification using compositional model checking, *Sci. Comput. Program.*, Vol.37, No.1-3, pp.279–309 (2000).
64) Mendías, J.M., Hermida, R., Molina, M.C. and Penalba, O.: Efficient Verification

of Scheduling, Allocation and Binding in High-Level Synthesis, *DSD '02: Proc. Euromicro Symposium on Digital Systems Design*, Washington, D.C., USA, p.308, IEEE Computer Society (2002).
65) Micheli, G.D.: Guest Editorial: High-Level Synthesis of Digital Circuits, *IEEE Des. Test*, Vol.7, No.5, pp.6–7 (1990).
66) Moore, J.S.: Symbolic Simulation: An ACL2 Approach, *Formal Methods in Computer-Aided Design*, pp.334–350 (1998).
67) Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L. and Malik, S.: Chaff: Engineering an Efficient SAT Solver, *Proc. 38th Design Automation Conference* (*DAC'01*) (2001).
68) Moura, L.D. and Bjørner, N.: Z3: An Efficient SMT Solver, *Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*) (2008).
69) Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R. and Dill, D.L.: Cmc: A Pragmatic Approach to Model Checking Real Code, *Proc. 5th Symposium on Operating Systems Design and Implementation* (2002).
70) Musuvathi, M. and Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs, *SIGPLAN Not.*, Vol.42, No.6, pp.446–455 (2007).
71) Narasimhan, N.: Theorem Proving Guided Development of Formal Assertions and their embedding in a High-Level VLSI Synthesis System, Ph.D. Thesis, University of Cincinnati (1998).
72) Narasimhan, N., Teica, E., Radhakrishnan, R., Govindarajan, S. and Vemuri, R.: Theorem Proving Guided Development of Formal Assertions in a Resource-Constrained Scheduler for High-Level Synthesis, *Form. Methods Syst. Des.*, Vol.19, No.3, pp.237–273 (2001).
73) Narasimhan, N. and Vemuri, R.: On the Effectiveness of Theorem Proving Guided Discovery of Formal Assertions for a Register Allocator in a High-Level Synthesis System, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, Grundy, J. and Newey, M. (eds.), Canberra, Australia, pp.367–386, Springer-Verlag (1998).
74) Necula, G.C.: Translation Validation for an Optimizing Compiler, *PLDI '00: Proc. 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2000).
75) Owre, S., Rushby, J.M. and Shankar, N.: PVS: A Prototype Verification System, *11th International Conference on Automated Deduction* (*CADE*), Kapur, D. (ed.), Lecture Notes in Artificial Intelligence, Vol.607, Saratoga, NY, pp.748–752, Springer-Verlag (1992).
76) Paulson, L.C.: *Isabelle: A generic theorem prover*, Lecure Notes in Computer Science, Vol.828, Springer-Verlag (1994).
77) Pnueli, A., Siegel, M. and Singerman, E.: Translation Validation, *TACAS '98: Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Vol.1384, pp.151–166 (1998).
78) Radhakrishnan, R., Teica, E. and Vermuri, R.: An approach to high-level synthesis system validation using formally verified transformations, *HLDVT '00: Proc. IEEE International High-Level Validation and Test Workshop* (*HLDVT'00*), Washington, D.C., USA, p.80, IEEE Computer Society (2000).
79) Rajan, S.P.: Correctness of Transformations in High Level Synthesis, *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, Johnson, S.D. (ed.), Chiba, Japan, pp.597–603 (1995).
80) Rinard, M. and Marinov, D.: Credible Compilation, *Proc. FLoC Workshop Run-Time Result Verification* (1999).
81) Sharp, R. and Rasmussen, O.: The T-Ruby Design System, *Form. Methods Syst. Des.*, Vol.11, No.3, pp.239–264 (1997).
82) The VIS Group, Brayton, R.K. and Sangiovanni-Vincentelli, A.L.: VIS: A System for Verification and Synthesis, Technical Report UCB/ERL M95/104, EECS Department, University of California, Berkeley (1995).
83) Vardi, M.Y.: Formal Techniques for SystemC Verification, *DAC '07: Proc. 44th Annual Conference on Design Automation* (2007).
84) Visser, W., Havelund, K., Brat, G. and Park, S.: Model checking programs, *Automated Software Engineering Journal*, pp.3–12 (2000).
85) Walker, R. and Camposano, R.: *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, Boston, MA, USA (1991).
86) Zuck, L., Pnueli, A., Fang, Y. and Goldberg, B.: VOC: A Methodology for the Translation Validation of Optimizing Compilers, *Journal of Universal Computer Science*, Vol.9, No.3, pp.223–247 (2003).

(Received March 3, 2009)
(Released August 14, 2009)

(Invited by Editor-in-Chief: *Hidetoshi Onodera*)

Sudipta Kundu received the B.S. and M.S. degree in Mathematics and Computing from the IIT Kharagpur, India in 2002 and 2004 respectively. There he received the prestigious silver medal for being ranked first in the department. He is currently working towards his Ph.D. degree in Computer Science and Engineering at UC San Diego. His advisor at UCSD is Prof. Rajesh Gupta and Prof. Sorin Lerner. His current research interests include high-level verification, equivalence checking, automatic verification of system-level designs, and compiler verification. Earlier Kundu has worked on high-level synthesis, embedded operating systems, and heterogeneous home networks.

Sorin Lerner received the B.Eng. in Computer Engineering from McGill University (Montreal, Canada), and the Ph.D. in Computer Science from the University of Washington (Seattle, USA). He is currently an assistant professor in the Computer Science and Engineering department at the University of California, San Diego. Sorin Lerner's research interests lie in programming languages and program analysis techniques for making software systems easier to write, maintain and understand, including static program analysis, domain specific languages, compilation, formal methods and automated theorem proving.

Rajesh Gupta is QUALCOMM chair professor in Computer Science and Engineering at UC San Diego, California. He received his B.Tech. in Electrical Engineering from IIT Kanpur in 1984, the M.S. in EECS from UC Berkeley in 1986 and the Ph.D. in Electrical Engineering from Stanford in 1994. Earlier Gupta was on the Computer Science Faculty at University of Illinois, Urbana-Champaign and UC Irvine. Prior to Illinois, Gupta was at the Intel Corporation in Santa Clara, California where he worked as a member of design teams for three generations of microprocessor devices with design experience from Bi/CMOS to high-speed GaAs devices. His current research is focused on energy and thermally efficient large-scale systems and distributed processing in sensor networks. His recent contributions include SystemC modeling and SPARK parallelizing high-level synthesis, both of which are publicly available and have been incorporated into industrial practice. Earlier Gupta lead the DARPA-supported Adaptive Memory Reconfiguration Management (AMRM) project which demonstrated methods to optimize movement and placement of application data across the memory hierarchy. His ongoing efforts include energy-efficient data-centers (NSF supported project GreenLight) and large scale computing using memory-coherent algorithmic accelerators and non-volatile storage systems (DOD supported project NV-DISC). In recent years, Gupta and his students have received a best paper award at IEEE/ACM DCOSS'08 and a best demonstration award at IEEE/ACM IPSN/SPOTS'05. Gupta is a recipient of the NSF CAREER Award and achievement awards at Intel. He has served as EIC of IEEE Design and Test of Computers and chair of the steering committee of IEEE Transactions on Mobile Computing. He is EIC of IEEE Embedded Systems Letters. Gupta is a Fellow of the IEEE.