Functional Equivalence Verification Tools in High-Level Synthesis Flows

Anmol Mathur
Calypto Design Systems

Edmund Clarke
Carnegie Mellon University

Masahiro Fujita
University of Tokyo

Pascal Urard
STMicroelectronics

Editor's note:
High-level synthesis facilitates the use of formal verification methodologies that check the equivalence of the generated RTL model against the original source specification. The article provides an overview of sequential equivalence checking techniques, its challenges, and successes in real-world designs.
—Andres Takach, Mentor Graphics

MOST MODERN DESIGNS start with an algorithmic description of the target design that is used to identify high-level system, or hardware and software, performance trade-offs. The designers' goal is to transform these algorithmic descriptions into ones that are hardware-friendly, such that the final transformed descriptions are accepted by high-level synthesis tools to generate high-quality RTL designs.[1] These design refinements consist of many transformation steps including changes of data types (for example, floating point to fixed point, refinements of bit-widths), removal of certain types of pointer manipulations, and partitioning of memories. Design models at levels higher than RTL, such as algorithmic and high-level models, are called system-level models (SLMs). Commonly, C/C++ or SystemC is used to represent these SLMs. Once a high-level synthesizable description is obtained in this manner, it can automatically be transformed into an RTL description through appropriate high-level synthesis tools.

Figure 1 illustrates a high-level design flow that starts from an algorithmic design description. To enforce the correctness of various descriptions in such a design flow, we must resolve two issues:

- *Eliminate bugs insofar as possible from the given description levels.* This is necessary at the highest level of abstraction that cannot be verified via equivalence checking, but it is also necessary when equivalence checking cannot verify all the behaviors of a model by comparing it against a higher-level model.
- *Guarantee the equivalence of the two descriptions.* We must guarantee equivalence of both a validated higher-level model and of a lower-level model obtained automatically or through manual refinements of the higher-level model.

These two issues complement each other to assure the correctness of the descriptions as a whole.

To eliminate bugs as much as possible from a given design description, simulation is not sufficient, especially for large and complicated designs. Therefore, we also use various formal methods to verify the design descriptions.

Model checking for SLM

Validating the SLM's functional correctness can be done via simulation (a dynamic technique). However, simulation can never be exhaustive due to the large input and state space of real models, and can miss functional issues in corner cases. Consequently, we also use formal (static) techniques to validate the SLM. These techniques do not require the user to specify test vectors.

0740-7475/09/$26.00 © 2009 IEEE Copublished by the IEEE CS and the IEEE CASS

Static analysis methods

Static analysis does not execute and follow design behaviors globally; instead, it checks behaviors of the design locally. The most basic static methods are those used in lint-type tools, and there have been significant extensions for more detailed and accurate analysis.[2] Designers generate control and data flow graphs from given design descriptions, and examine dependencies on control and data that result in various dependency graphs.

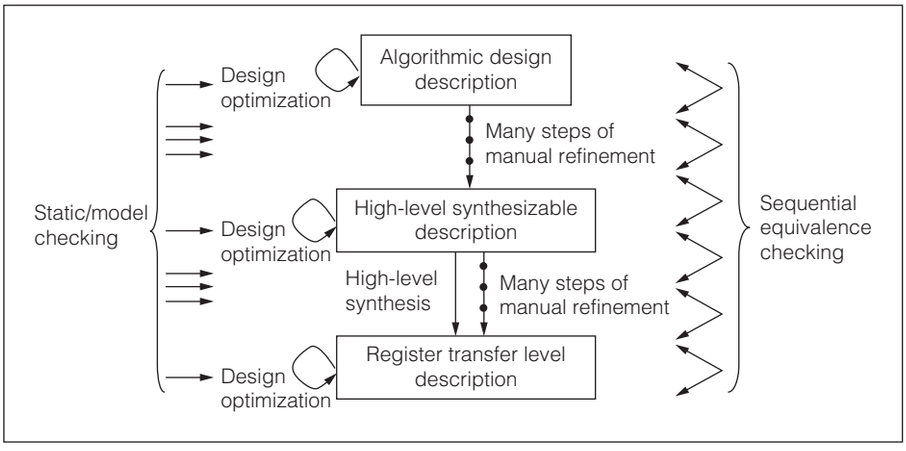

Figure 1. High-level design flow.

With appropriate sets of rules, we can check various items. For example, we can detect uninitialized variables by traversing control/data dependencies to check if a variable is used before it is assigned a value. Note that this search process is local and does not need to start from the beginning of the description. Whereas model-checking methods basically examine the design descriptions exhaustively starting from initial or reset states, static checking analyzes design descriptions only in small and local areas. Instead of starting from initial states, static checking first picks up target statements and then examines only small portions of design descriptions that are very close to those target statements, as shown in the left part of Figure 2.

Because analysis is local, it can deal with very large design descriptions, although such analysis has less accuracy. This means that static checking methods can produce false errors or warnings. Null pointer references can be checked with backward traversals of control and data dependencies, and relative execution orders among concurrent statements can be checked with backward and forward traversals starting from the target concurrent statements to be checked. Static checking methods, which have been applied to various software verification tasks, have proven highly effective even for very large descriptions having millions of lines of code. For example, Unix kernels and utilities have been automatically analyzed by static methods, which have revealed several design bugs.[2] Also, researchers have been working to combine static and model-checking methods within the same framework for more efficient and accurate analysis.[3]

Model-checking methods

After designers apply static checking methods to the design descriptions, they can use model-checking methods to detect more complicated bugs that can be found only through systematic traversals starting from initial states or user-specified states. Given a property, model-checking methods determine whether all possible execution paths from initial states satisfy that property. *Properties* are conditions on possible values of variables among multiple time frames, such as "if a request comes, a response must be returned within two cycles" and "even if an error happens, the system eventually returns to its reset state."

In one approach, the C-bounded modeling checking tool, CBMC,[4] verifies that a given ANSI-C program satisfies given properties by converting them into bit-vector equations and solving satisfiability (SAT) using

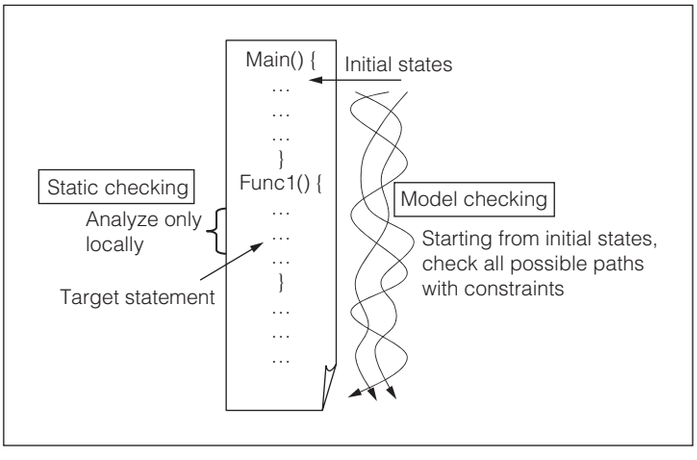

Figure 2. Static checking and model-checking approaches.

a SAT solver, such as Chaff.[5] All statements in C descriptions are automatically translated into Boolean formulas, and they are analyzed up to given specific cycles. Although CBMC and similar approaches can generally verify C descriptions, they cannot deal with large descriptions because the SAT processing time grows exponentially with design size.

Model-checking methods for RTL or gate-level designs are mostly based on Boolean functions. These methods make each variable a single bit and use decision procedures for Boolean formulas, such as SAT procedures. In SLMs, however, there can be many word-level variables that have multiple bit widths, such as integer variables. If we expand all of them into Boolean variables, the number of variables can easily exceed the capacity of SAT solvers. Therefore, when reasoning about high-level descriptions, we commonly use word-level decision procedures. Extended SAT solvers, called Satisfiability Modulo Theory (SMT) solvers, such as CVC,[6] can deal with multibit variables as they are. They are similar to theorem provers and consist of several decision procedures.

Although reasoning about the SLM can be much more efficient with SMT solvers, realistic design sizes are still simply too large to be processed. Moreover, the types of formulas that word-level decision procedures can deal with are limited, and sometimes the formulas must be expanded into Boolean ones if they are outside the scope of word-level decision procedures. Therefore, researchers have proposed several automatic abstraction techniques, such as that proposed by Clarke et al.,[7] by which large C design descriptions can be sufficiently reduced, and SAT/SMT solvers can quickly reason about the reduced design description. This automatic reduction is generally target property dependent. That is, given a property to be verified, portions of design descriptions that do not influence the correctness of the property are automatically eliminated.

Depending on characteristics of target properties, the amount of reduction varies. For example, if the target is to check whether array accesses never exceed array bounds, most of the design descriptions are irrelevant and can be omitted for model checking. Abstraction-based approaches, however, have the problem of possibly generating false counterexamples—that is, the generated counterexamples are true only for abstracted descriptions, but not true (executable) for the original design descriptions. Abstraction must be refined to avoid such false negative cases. A considerable amount of ongoing research concerns automatic abstraction refinements for model checking.[7] State-of-the-art model checkers for SLMs can process tens of thousands of lines of codes within several hours.

Sequential equivalence checking

Sequential equivalence checking (SEC) is a formal technique that checks two designs for equivalence even when there is no one-to-one correspondence between the two designs' state elements. In contrast, traditional combinational equivalence checkers need a one-to-one correspondence between the flip-flops and latches in the two designs. SLMs can be untimed C/C++ functions and have very little internal state. RTL models, on the other hand, implement the full microarchitecture with the computation scheduled over multiple cycles. Accordingly, significant state differences exist between the SLM and RTL model, and SLM-to-RTL equivalence checking clearly needs SEC. Researchers have investigated SEC techniques,[8-12] and also commercial SEC tools that are now available, such as the one from Calypto Design. SEC can check models for functional equivalence even when they have differences along the following axes:

- *Temporal differences at interfaces:* The SLM typically has a parallel interface where all inputs arrive at the same time, whereas the RTL model accepts inputs serially.
- *Differences in internal state and microarchitecture:* Various sequential transformations in converting an SLM to an RTL model (such as scheduling of a computation over multiple cycles, pipelining, resource sharing to minimize area or power, retiming of registers across logic) create significant divergence between the state machine in the SLM and RTL.

The output of the equivalence checking process is either a counterexample illustrating a scenario in which the behaviors of the SLM and the RTL differ or a proof that the two are equivalent.

The use of equivalence checking to verify RTL functional correctness has two key advantages. The first advantage is complete verification of the RTL model with respect to the SLM. Unlike simulation-based or assertion-based approaches, in which

functional coverage of the RTL model is an issue, SEC checks that all the RTL behaviors are consistent with those in the SLM. This results in very high coverage of the RTL behaviors. It should be noted that SLM-to-RTL equivalence checking verifies only the RTL behaviors that are also present in the SLM. Thus, if the SLM has a memory implemented as a simple array while the RTL model implemented a hierarchical memory with a cache, equivalence checking will not verify whether the cache is working as intended as long as the overall memory system works as expected.

The second advantage of equivalence checking is simplified debugging. In case of a functional difference between the SLM and RTL, SEC produces the shortest possible counterexample that shows the difference. This contrasts with traditional simulation-based approaches, which may find the difference but only after millions of cycles of simulation. The conciseness of the counterexample makes the process of debugging and localizing the error much more efficient.

Key technology in SEC

Four key technology pieces are required to make SEC feasible:

- Model extraction from SLM
- Sequential analysis
- Word-level solvers
- Mechanism for specifying temporal mappings at I/Os and state points

In model extraction from SLM, extracting the hardware model from SystemC or C/C++ with extensions is a key area where new techniques will be required. Four of the issues that the model extraction technology will handle are identification of persistent state, generation of state machines that are implicitly specified via wait statements, synthesis of channels and interfaces, and handling of arrays and pointers.

In sequential analysis, SEC requires technology for the efficient unrolling of finite-state machines (FSMs) to align the SLM and RTL state machines to synchronizing states, in which their output and next state functions might need to be computed and compared. It requires efficient symbolic analysis, and state and output-based induction procedures, to prove the correctness of the transactions of interest. To ensure that SEC can handle designs of sufficient complexity, it is also imperative to leverage the SLM's existence to find intermediate equivalences and decompose the problems given to the solvers.

Typical SLMs and RTL models use word-level arithmetic operators. Most existing formal verification tools synthesize these operators to gates and then use bit-level techniques, such as binary decision diagrams (BDDs) or bit-level SAT, to solve the resulting problems. To allow SEC to perform enough sequential unrolling, it is important to reason about arithmetic operators at the word level, as well as to develop solvers that can efficiently switch between word-level and bit-level techniques.

Finally, because the SLM and RTL model have different interface protocols, it is important for an SEC tool to provide mechanisms for a designer to specify the temporal relationships between corresponding I/Os and internal state points. Since specifying such relationships can be cumbersome and error-prone, it is also useful for the SEC tool to automatically infer these mappings. In flows where SEC is used in conjunction with high-level synthesis tools, it is often possible for the HLS tool to generate the I/O and state maps. This automation is crucial to make SEC usable in an HLS flow.

Deploying and using SEC

Design teams should follow four guidelines to effectively use SEC or other simulation-based techniques for keeping SLMs and RTL models functionally equivalent:

Coordination between SLM and RTL design teams. The teams designing and maintaining the SLM and RTL model must be well coordinated, and both need to realize the value of keeping the SLM and RTL models equivalent. This is naturally the case in HLS-based flows, but in flows where the RTL model is manually created, this needs to be enforced for SLM-to-RTL verification to work. The SLM team must be aware that often bugs can be found in the SLM when comparing the SLM and RTL model. Further, the modeling style might need to be adjusted to allow effective use of formal tools such as SEC.

Consistent design partitioning. SEC is a block-level verification tool due to capacity limitations of formal technology. To effectively use SEC, it is crucial that the SLM and RTL model be consistently partitioned into subfunctions and submodules. Clean and consistent design partitioning provides an

opportunity to use sequential equivalence checking at the level of individual SLM/RTL blocks.

Creation of SLMs with hardware intent. To perform SLM-to-RTL equivalence checking using a sequential equivalence checker, the SLM must be written to let the tool infer a hardware-like model statically from the source. This requires that the team create the SLM to follow certain coding guidelines that allow the SLM's static analysis. Use of statically sized data structures instead of dynamically allocated memory, explicit use of memories to reuse the same storage for multiple arrays instead of pointer aliasing, and statically bounding loops are some examples of constructs that make SLMs more amenable to SEC and HLS tools.

Orthogonalization of communication and computation. Clear separation between the computational and communication aspects of the SLM allows easier refinement of the communication protocol, if needed, to make the interface timing more closely aligned with that of an RTL model.

Challenges in SEC usage

The main challenges in the usage of SEC-based flows stem from the fact that SEC has capacity limitations. SEC's complexity is a function of the following factors:

- *Size of the SLM/RTL blocks being compared.*
- *Latency and throughput difference between the SLM and RTL.* The greater the sequential differences between the SLM and RTL, the larger the sequential depth to which SEC needs to explore the SLM/RTL state machines—and hence, the larger the runtime and memory usage is in SEC.
- *Difference in level of arithmetic abstraction.* If both the SLM and RTL represent their computations at the operator level, the complexity of using SEC is lower than the cases where, in the RTL model, the arithmetic operators have been decomposed into bit-level constructs.
- *Amount of correspondence between internal states of SLM and RTL model.* If the SEC tool can detect internal states and signals in the SLM and RTL model that are identical, then the verification problem can be decomposed and simplified.

Currently, SEC tools can prove equivalence between SLM and RTL models when the RTL model gate count is between 500K and 700K gates and where the latency or throughput difference between the models is in the hundreds of cycles. If the SLM and RTL model to be verified exceed these limits, the design teams should find smaller subfunction (SLM) or submodule (RTL) pairs where SEC can be applied. This requires the SLM and RTL model to have consistent partitioning and clean interface mappings at the subfunction/submodule level.

Deploying SEC in HLS flows

High-level synthesis generates an RTL description from a C/C++/SystemC model.[1] By using HLS, a designer's development time is greatly reduced via coding at a higher abstraction level than RTL. However, to take advantage of system-level design and its time savings, formal verification between the two design descriptions is required. Verification helps reveal any bugs in HLS tools and permits any manual modifications at the synthesized RTL description.

Figure 3 illustrates a working model in which an HLS tool is used. Algorithmic C/C++ models of the application (such as video codecs and wireless digital baseband modems) are the starting point. After manual hardware and software partitioning, blocks to be implemented as IP blocks are individually split (A, B, C in the figure). These C/C++ models are used

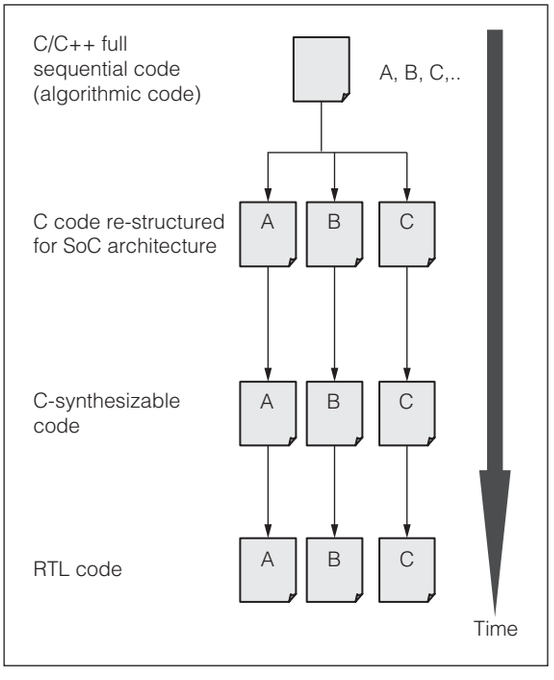

Figure 3. Typical HLS-based design flow.

early in the design phase for integration in transaction-level modeling platforms, which permits parallel development of firmware.

Accordingly, hardware IP designers refine the code, in parallel with transaction-level modeling, to make it synthesizable by an HLS tool. HLS-based flows, however, pose new verification challenges. One such challenge is that, since traditional C/C++/SystemC models are not synthesizable, the use of HLS tools often requires the creation of a new synthesizable SLM model. Designers must ensure that this synthesizable SLM is functionally equivalent to the original SLM. Also, the SLM often needs to be refined to obtain an optimal RTL description from HLS. Typical SLM refinements include optimizing the widths of intermediate signals in fixed-point computations.

A second challenge is in using HLS tools to synthesize RTL descriptions because it requires the specification of input-to-output latency and the throughput rate at which inputs are consumed and outputs produced by the RTL. It is important to catch errors in the specification and interpretation of these HLS directives as well as potential mismatches between the SLM and RTL model produced by HLS resulting from bugs in HLS tools.

SEC can address both of these verification issues in an HLS-based flow. The use of SEC for verification of SLM refinements is beneficial because the refinement is manual and so increases the likelihood of bugs being introduced.

SEC in HLS flows

Deployment of SEC in HLS flows can be automated because the HLS tools can automatically generate the wrappers and interface mapping needed for SEC. In addition, in HLS-based flows, the SLM and RTL model are expected to be completely bit-accurate, and there are no issues with the SLM not being synthesizable.

Deploying SEC in HLS-based flows involves several key issues. First, the design team must ensure that the capacity of SEC tools is compatible with the typical block sizes that design teams want to push through HLS flows.

Second, HLS is typically used on arithmetic-heavy computation blocks (for instance, image scaling, image interpolation, and video codecs). Such designs have large latencies when implemented in RTL because they deal with large input streams. To perform SEC on such blocks currently requires that the input data size be scaled down. This scaling, however, reduces the verification coverage and requires manual intervention.

Third, input-data-dependent variability in RTL latency can complicate the interface mapping for SEC. As a result, the user often needs to identify a handshake signal indicating that the RTL computation is done.

Finally, we have the issue of providing full coverage of HLS data types and libraries. HLS tools sometimes have proprietary data types for bit-vectors and fixed-point data. Supporting all such data types efficiently is a challenge for SEC tools.

Usage of SEC: Results

For the past three years, STMicroelectronics has used Calypto Design Systems' SLEC tool, a sequential equivalence checker, on real SLM-to-RTL verification projects—both manual RTL flows and HLS-based flows. Table 1 shows some of the bugs SLEC found when applied to designs synthesized through an HLS flow. Clearly, although some of the bugs arose from problems in the synthesis tool, a few resulted from ambiguities in the SLM. In particular, the shift by a negative number is undefined in C, but was interpreted and synthesized by the HLS tool.

Table 2 shows the design size, latency and throughput, and SLEC runtime for runs on blocks in an STMicroelectronics project. These were synthesized through Mentor Graphics' Catapult HLS tool. Catapult

Table 1. Bugs found by SLEC in HLS flows.

Application	Bugs found
Multimedia processor	Dead-end states created in RTL resulting in mismatch between C and RTL.
Fast Fourier transform	Bad sign extension logic.
Quantize	Divide by zero defined in RTL, but undefined in C code. C model needed to be fixed.
Ultrawideband filter	Shift left or right by *N* bits, when the value being shifted is less than *N* bits.
Multimedia processor	Shift by an integer in the C-code could be a shift by a negative number that is undefined in C. C model needed to be fixed.

Table 2. Statistics of SLEC runs on HLS designs.

Design	Size (no. of gates)	Latency (no. of clock cycles)	Throughput (no. of clock cycles)	Runtime (minutes)
UWB—weiner	403K	71	71	8
UWB interleaver	43K	69	69	2
UWB mapper	44K	2	2	49
Video_pipe—full	400K	20	20	300
Video_pipe—DCT	137K	321	321	13
Scalar	1.4M	172	172	120

* UWB: ultrawideband; DCT: discrete cosine transform

can automatically generate the I/O mappings and SLM and RTL latency and throughput information needed by SLEC. Most of the designs in Table 2 are from signal- or image-processing applications and have significant arithmetic content. The SLEC runtimes ranged from a few minutes to 5 hours. The sizes of these blocks ranged from 40,000 gates to 1.4 million gates. The SLM was a completely untimed C model in all cases. Moreover, in all of these cases, SLEC was able to prove complete equivalence between the C model and the HLS-generated RTL model.

SEC IS A KEY TECHNOLOGY needed to keep SLMs and RTL models consistent and to quickly weed out any RTL or SLM bugs without the need to write testbenches at the block level. As design teams deploy HLS-based flows, SEC fills several critical verification needs. SEC has been deployed in both HLS-based flows and flows in which RTL is manually created. SEC technology must continue to evolve to ensure that it can handle larger block sizes and that it can check designs with larger latency and throughput differences for equivalence. ■

References

1. P. Coussy and A. Morawiec, eds., *High-Level Synthesis: From Algorithm to Digital Circuit,* Springer, 2008.
2. D. Engler et al., "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. ACM SIGOPS Operating Systems Rev.,* vol. 35, no. 5, 2001, pp. 57-72.
3. C. Cadar, D. Dunbar, and D. Engler Klee, "Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proc. Operating System Design and Implementation* (OSDI 08), Usenix Assoc., 2008, pp. 209-224.
4. E.M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," *Proc. Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 04), LNCS 2988, Springer-Verlag, 2004, pp. 168-176.
5. M. Moskewicz et al., "Chaff: Engineering an Efficient SAT Solver," *Proc. 39th Design Automation Conf.* (DAC 01), ACM Press, 2001, pp. 530-535.
6. C. Barrett and C. Tinelli, "CVC3," *Proc. 19th Int'l Conf. Computer Aided Verification* (CAV 07), LNCS 4590, Springer-Verlag, 2007, pp. 298-302.
7. E.M. Clarke et al., "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking," *J. ACM,* vol. 50, no. 5, 2003, pp. 752-794.
8. P. Georgelin and V. Krishnaswamy, "Towards a C++-Based Design Methodology Facilitating Sequential Equivalence Checking," *Proc. 43rd Design Automation Conf.* (DAC 06), ACM Press, 2006, pp. 93-96.
9. D. Kroening, E. Clarke, and K. Yorav, "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking," *Proc. 40th Design Automation Conf.* (DAC 03), ACM Press, 2003, pp. 368-371.
10. E.M. Clarke, H. Jain, and D. Kroening, "Verification of SpecC Using Predicate Abstraction," *Formal Methods in System Design,* vol. 30, no. 1, 2007, pp. 5-28.
11. A. Mathur and V. Krishnaswamy, "Design for Verification in System Level Models and RTL," *Proc. 44th Design Automation Conf.* (DAC 07), ACM Press, 2007, pp. 193-198.
12. T. Matsumoto, H. Saito, and M. Fujita, "Equivalence Checking of C Programs by Locally Performing Symbolic Simulation on Dependence Graphs," *Proc. 7th Int'l Symp. Quality Electronic Design* (ISQED 06), IEEE CS Press, 2006, pp. 370-375.

Anmol Mathur is the CTO and cofounder of Calypto Design Systems. Anmol holds multiple patents and has published extensively in the areas of formal verification, low-power design, and arithmetic optimization

techniques. He received a PhD in computer science from the University of Illinois at Urbana-Champaign.

Masahiro Fujita is a professor of engineering at the University of Tokyo, which he joined in 2000. He has received several awards from major Japanese scientific societies for his work in formal verification and logic synthesis. He has a PhD in information engineering from the University of Tokyo.

Edmund M. Clarke is the FORE Systems University Professor of Computer Science at Carnegie Mellon University. He was elected to the National Academy of Engineering in 2005 for contributions to the formal verification of hardware and software correctness. He was a co-recipient of the 2007 ACM Turing Award for his role in developing model checking into a highly effective verification technology and a co-winner of the ACM Kanellakis Award in 1998 for the development of symbolic model checking. He has a PhD in computer science from Cornell University. Clarke is a Fellow of the ACM and the IEEE.

Pascal Urard is the system design director in the Analog & Mixed Signals group at STMicroelectronics' ST Technology–R&D, where he works on ESL design flows and has initiated cooperation with EDA companies to enhance and deploy HLS. His research interests include test, engineering, ASIC design, and chip architecture. He has a bachelor's in electrical engineering from ISEN (Institut Supérieur de l'Electronique et du Numérique), Lille, France.

■ Direct questions and comments about this article to Anmol Mathur, Calypto Design Systems, 2903 Bunker Hill Lane, Santa Clara, CA 95054; amathur@calypto.com.

For further information on this or any other computing topic, please visit our Digital Library at http://www.computer.org/csdl.

Running in Circles Looking for a Great Computer Job or Hire?

The IEEE Computer Society Career Center is the best niche employment source for computer science and engineering jobs, with hundreds of jobs viewed by thousands of the finest scientists each month - **in *Computer* magazine and/or online!**

> Software Engineer
> Member of Technical Staff
> Computer Scientist
> Dean/Professor/Instructor
> Postdoctoral Researcher
> Design Engineer
> Consultant

careers.computer.org
http://careers.computer.org

The IEEE Computer Society Career Center is part of the *Physics Today* Career Network, a niche job board network for the physical sciences and engineering disciplines. Jobs and resumes are shared with four partner job boards - *Physics Today* Jobs and the American Association of Physics Teachers (AAPT), American Physical Society (APS), and AVS: Science and Technology of Materials, Interfaces, and Processing Career Centers.

IEEE computer society