Practical Formal Verification in Microprocessor Design

Robert B. Jones, John W. O'Leary, and Carl-Johan H. Seger
Intel

Mark D. Aagaard
University of Waterloo

Thomas F. Melham
University of Glasgow

Practical application of formal methods requires more than advanced technology and tools; it requires an appropriate methodology. A verification methodology for data-path-dominated hardware combines model checking and theorem proving in a customizable framework. This methodology has been effective in large-scale industrial trials, including verification of an IEEE-compliant floating-point adder.

FUNCTIONAL VALIDATION is one of the major challenges in chip design today, with test generation, test bench construction, and simulation consuming a significant portion of the design effort. Throughout the 1990s, formal verification emerged as a promising complement to conventional simulation-based validation.[1] Most formal verification research concerns algorithms and focuses on tool capacity limits. Yet almost any serious verification effort faces many practical difficulties besides capacity. In a large verification project, the effort required to organize the multitude of tasks, specifications, and verification scripts can limit the quality and productivity of the work.

We tackle this problem by coupling our research on verification algorithms and tools with research on verification methodology. Our goal is to address the realities of design practice—rapid changes and incomplete specifications—while producing high-quality results and improving verification productivity. Our methodology systematically organizes a large verification effort's many interdependent activities and provides a guiding structure for the verification process.

Any formal verification tool researcher is keenly aware that what is a routine verification for the technology expert or tool developer may be very difficult for others to duplicate. Our methodology addresses this problem by tailoring a formal, custom-built verification framework, Forte, to industrial-scale circuits and industrial design environments. Forte combines an efficient, linear temporal logic model-checking algorithm, called symbolic trajectory evaluation (STE),[2] with lightweight theorem proving. FL—a custom, general-purpose functional programming language—tightly integrates the model checker and the theorem prover. This combination of model checking, theorem proving, and a general-purpose programming language makes the verification environment customizable and lets large verification efforts be organized effectively.

Our methodology has evolved over several years of use on fully custom, high-performance

0740-7475/01/$10.00 © 2001 IEEE

microprocessors. Verification projects' life spans range from a few months to several years, as some verifications progress from initial designs to subsequent proliferations. Focusing our methodology on data-path-intensive circuits, we usually begin formal verification after the circuit has undergone initial validation by conventional simulation.

The methodology we advocate involves

- a process of circuit assessment and encapsulation that we call *wiggling*,
- scalar (1s and 0s) verification,
- symbolic model checking using Boolean expressions, and
- theorem proving.

Each phase has a specific purpose, associated tasks, and resulting artifacts. We illustrate this methodology by applying each phase to the verification of an IEEE-compliant, extended-precision floating-point adder, one of many large-scale industrial verifications we have carried out.

We adapted this article from an earlier work,[3] whose references expand on different aspects of this article.

Methodology

Creating an industrially effective formal verification methodology is difficult because it must simultaneously satisfy several requirements:

- *Realism.* The methodology must conform to the constraints imposed by a production design environment. It cannot depend on resources that are usually not available, such as complete, up-to-date specifications or unlimited access to design engineers.
- *Transparency and soundness.* The verification engineer should know what has been proved and what has not. The methodology must also be sound so that erroneous verification results are impossible.
- *Structure.* An effective methodology structures the overall verification effort without imposing undue constraints. It must help new users to learn the system and experienced users to increase their productivity.
- *Incrementality and recoverability.* Preliminary results are needed early in a verification effort, and even incomplete verifications should deliver debugging value. If circuit or specification changes cause a previously successful verification to fail, test cases from earlier in the verification development process should be available to help isolate the problem.
- *Debugging and feedback.* The bulk of any verification effort is debugging, so optimizing the verification environment for failure, rather than success, is crucial. The system not only must quickly discover failures but also should provide focused feedback and a tight debugging loop.
- *Top-down and bottom-up decomposition.* The methodology must allow decomposition of complex verification tasks into smaller, simpler tasks. Because subtle design features and model-checking capacity limits are discovered through bottom-up exploration, a realistic methodology must let the decomposition strategy evolve through a combination of bottom-up and top-down development.
- *Regression and reuse.* A methodology should produce easily maintainable verification artifacts that adapt to changing specifications and designs. Specifications and high-level decomposition strategies are particularly important candidates for reuse.

Our methodology strikes a balance between these different requirements. It has four distinct, but overlapping, phases.

The *wiggling* phase's goal is to be able to predictably produce defined values on the circuit's outputs. Verification engineers must educate themselves and their tools about the circuit and its operating environment. They begin by reading design documentation and register-transfer-level (RTL) code, then consulting other verification and design engineers. In practice, documentation is frequently incomplete or outdated, and designers' time is limited. Therefore, simulating the circuit on test data and using Forte to probe the circuit's structure and behavior provides an important source of additional information. The wiggling phase's primary artifact is a circuit application programming inter-

face (API) that provides access to the circuit signals and their timings, allowing the subsequent phases to abstract from these details.

Targeted scalar verification builds on the wiggling phase by using the circuit API to simulate the circuit with scalar (bit-pattern) inputs. The goal is to develop a specification reference model against which the circuit is checked. We begin with a reference model that covers core functionality, and then we gradually incorporate additional features. Our incremental approach is motivated by the realities of circuit complexity and incomplete documentation—factors that frequently force the verification engineer to reverse-engineer certain design aspects. This phase's main artifacts are a reference model; an improved, debugged circuit API; and a set of scalar test vectors, which are saved to support regression testing. At this phase, the circuit design is usually more mature than the reference model. Although discovering real design bugs is possible, discrepancies between the circuit and the reference model usually stem from errors in the circuit API or from the reference model itself.

Symbolic model checking, the beginning of full formal verification, uncovers most hardware bugs. We present the circuit with symbolic values represented by binary decision diagrams (BDDs)[4] and compute a symbolic representation of the resulting outputs. We then check this representation against the outputs predicted by our reference model, to confirm that the circuit conforms to its specification. Because Forte allows simulation with an arbitrary mix of scalar and symbolic values, we can incrementally evolve model checking from scalar verification to exhaustive symbolic verification. The two artifacts produced in this phase are a decomposition of the verification problem into pieces small enough for model checking, and a set of verification scripts.

Theorem proving accomplishes two purposes: mechanically checking the decomposition strategy's soundness and verifying the reference model's correctness by proving implementation-independent properties about it. For example, we might relate our reference model to a higher-level benchmark such as the IEEE floating-point standard, the PCI bus protocol specification, or a programmer's reference manual. The artifacts produced in this stage include the proof script and any associated high-level properties or specifications. Because this work is the most removed from the circuit details, it is the most likely to be reusable across multiple designs in the same class.

Forte verification environment

An effective methodology requires a robust, capable verification platform. Forte, which we evolved from Seger's Voss system,[5] enables the methodology described here, through its architecture and features. Forte integrates model-checking engines, BDDs, circuit manipulation functions, theorem proving, and the FL functional programming language. Forte compiles hardware-description-language source code into formal circuit models and includes tightly integrated graphical interfaces to display circuit structures and waveforms.

FL programming language

FL is a strongly typed functional programming language. A distinguishing feature is that BDDs are built into the language, and every Boolean object is represented as a BDD. FL provides a flexible interface for invoking and orchestrating model-checking runs, serves as an extensible macrolanguage for expressing specifications, and provides the control language for Forte's theorem prover. Using a general-purpose language as the primary interface helps make Forte extensible and customizable to individual verification problems.

Graphical interfaces

Forte includes graphical tools for FL programming and debugging. It also has extensive interactive circuit-visualization capabilities through three different interfaces: a node browser, a waveform viewer, and a circuit browser. These are shown in Figure 1. The node browser acts as a search interface for circuit signals, hardware constructs, and module interfaces. The waveform viewer displays bit and vector waveforms from both scalar and symbolic simulation. The circuit browser draws gate-level representations of circuits and provides facilities for traversing hierarchy, display-

ing special circuit constructs, and dynamically displaying values on circuit nodes.

Symbolic trajectory evaluation

STE performs model checking with an algorithm, based on symbolic simulation, that is significantly more efficient and flexible for data path verification than conventional model-checking algorithms.[2] As a scalar simulator, STE computes the result of executing a circuit with scalar (0, 1) test vectors as inputs. As a symbolic simulator, it computes signal values as symbolic expressions in terms of arbitrary Boolean inputs. As a model checker, it automatically checks the validity of a simple temporal logic formula for arbitrary inputs—computing an exact characterization of the disagreement if the formula is not unconditionally satisfied. STE thus provides a straightforward migration path between scalar simulation, symbolic simulation, and verification.

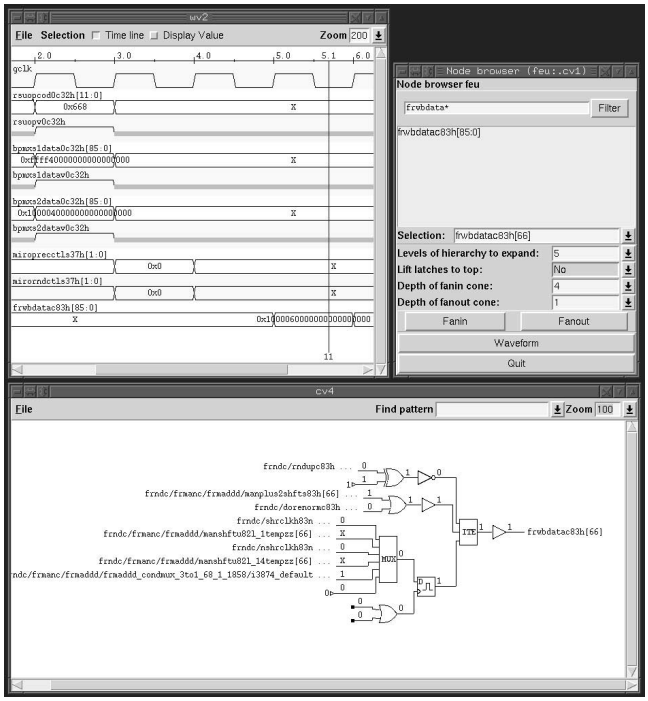

Figure 1. Forte's graphical user interface.

ThmTac

Model checking's capacity limits often force verification engineers to decompose verifications. Therefore, Forte includes ThmTac, a higher-order-logic theorem-proving system. We call this a *lightweight* theorem prover, because it is optimized for composing and decomposing model-checking results rather than formalizing arbitrary mathematical theories. Implemented in FL, ThmTac is tightly integrated with the STE model checker. As a result, ThmTac works directly on model-checking results: there is no need for translation or reformulation as would be necessary in a typical approach that uses a theorem prover with separate logic.

Verifying a floating-point adder

We illustrate the four phases of our methodology with a floating-point-adder verification. The adder performs IEEE-compliant floating-point addition and subtraction at single, double, and double-extended precision, and supports four rounding modes: toward 0, toward –∞, toward +∞, and toward the nearest real number representable in floating-point format. This adder was verified as part of a large-scale verification effort undertaken on a recent Intel IA-32 processor.[6]

In the methodology's first three phases, we verify a gate-level implementation of the adder against a reference model (data path specification). In the fourth phase, we verify the reference model against an FL interpretation of the IEEE standard for floating-point addition.

Wiggling

The primary purpose of the wiggling phase is to develop an API that insulates the rest of the verification effort from the circuit's low-level implementation details. The circuit API for the adder has two FL functions: one for inputs and one for outputs.

The first API function, **`fadd_fsub_protocol`**, describes the circuit inputs' behavior during an addition or subtraction operation. This function, shown in Figure 2 (next page), takes the following arguments:

- **`uop`**, the executing instruction's bit-vector opcode (**`FADD`** or **`FSUB`** in this example);
- **`pc`** and **`rc`**, the operation's precision and rounding mode; and
- **`A`** and **`B`**, the two floating-point operands.

This API's first few lines impose the conditions that the circuit is clocked correctly and that no

```
// tie FL identifiers to circuit signals
let opcod = "rsuopcod0c32h[11:0]";
let opcodv = "rsuopv0c32h";

let fadd_fsub_protocol [uop, pc, rc, A, B] =
   // Drive control signals
      generate_clocks and         // clocks toggle correctly
      no_resets and               // reset is false
   // Drive opcode and data inputs
      ( (  (opcod isv uop ) and   // opcode
           (opcodv is TRUE) and   // opcode is valid
           (s1 isv A ) and        // operand 1
           (s1v is TRUE) and      // operand 1 is valid
           (s2 isv B ) and        // operand 2
           (s2v is TRUE) )        // operand 2 is valid
        in_cycle 2) and
// Round and precision control come one cycle later
   (  (  (roundc isv rc ) and     // rounding mode
         (precc isv pc ) )        // precision control
      in_cycle 3);
```

Figure 2. Function for the circuit API's inputs.

resets occur during an operation's execution. The API then drives the circuit's inputs with the opcode, the two input operands, and the rounding and precision controls—each in the appropriate clock cycle and accompanied by data-valid signals. The FL library functions called by the API (**is**, **isv**, and **in_cycle**) map specification values to RTL signals and establish a unit-delay temporal abstraction, which lets signal timing be specified in terms of clock cycles rather than absolute time points. The waveform browser illustrates the protocol described by the API.

The second API function, **fadd_fsub_result**, simply observes output **wbdata** at the appropriate clock cycle, checking it against the given argument, **res**. The API also checks that the valid bit, **wbdatav**, is set.

```
let fadd_fsub_result res =
  (  (  (wbdata isv res ) and
        (wbdatav is TRUE) )
     in_cycle 5);
```

We began wiggling the adder by adding 1.0 and 0.0, along with other trivial operations. In the API's first version, we toggled the clocks for more cycles than were needed and drove the data and control signals with constant values for additional clock cycles. At first, the circuit produced undefined outputs, which we detected because unspecified circuit nodes in STE are assigned a special undefined value similar to X in VHDL (VHSIC hardware description language) simulation. By analyzing circuit behavior, viewing circuit structure with the circuit browser, writing FL functions to probe the circuit, studying the RTL code, and consulting with the designers, we identified additional control signals that needed to be driven, and we added them to the API's **no_resets** portion. Once we had fully defined (non-X) outputs, we gradually refined the API by reducing the number of clock cycles during which signals were driven.

When the API could reliably generate fully defined values on the output signals, we moved from wiggling to targeted scalar verification.

Targeted scalar verification

This second phase develops a reference model and uses the API to connect it to the circuit. For the adder, the function **fadd_fsub_result** was given a result value, **res**, calculated by **fspec**—a function encapsulating our FL reference model for floating-point addition and subtraction. The reference model began as a straightforward adaptation of a textbook algorithm.[7] At first, it supported only double-precision operations and rounding toward zero, and we tested it only for true addition (addition of operands with like signs, or subtraction of operands with different signs). By verifying various corner cases, we extended our reference model to support single, double, and double-extended precisions, and the four rounding modes. For example, $1.0 \times 2^{30} - 1.0 \times 2^{0}$, in single precision, should yield 1.0×2^{30} when rounded toward $+\infty$, and $1.11111111111111111111111 \times 2^{29}$ when rounded toward $-\infty$. We usually proceeded by identifying a corner case, working out the correct behavior with pencil and paper, coding the desired behavior into the reference model, and then checking the circuit against the revised reference model. As a safeguard against inadvertently incorporating circuit bugs in the reference model, we later used theorem proving to check the reference model against an FL implementation of the IEEE 754 standard.

Our final reference model comprises five

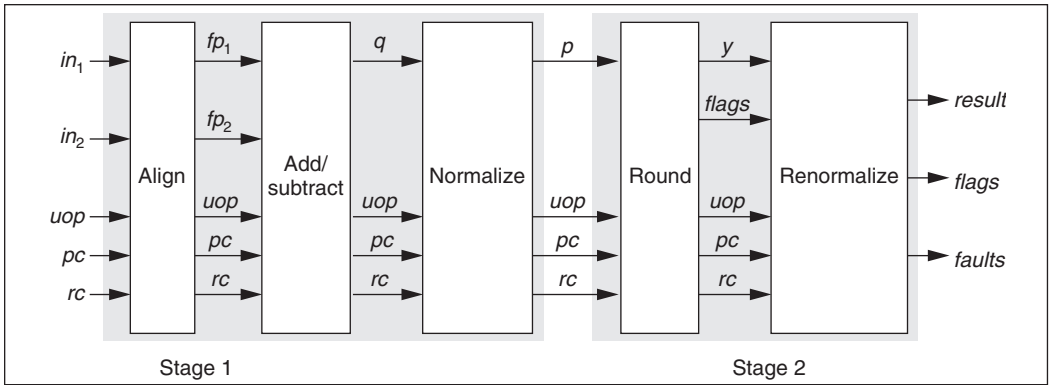

Figure 3. Reference model structure for floating-point addition.

```
let isNORM fp   = (exp fp '> 0) AND (exp fp '< MAX_EXP) AND (Jbit fp);
let isZERO fp   = (exp fp '= 0) AND (man fp '= 0);
let LEGAL_FP fp = (isZERO fp) OR (isNORM fp);
```

Figure 4. Constraints on operands of the floating-point adder.

functional steps, as shown in Figure 3. The first step inspects the operands and shifts the smaller operand's mantissa to the right in preparation for addition. The second step adds or subtracts the mantissas. The third step normalizes the mantissa, shifting it left or right to align its significant bits with the binary point. The fourth step rounds and truncates the mantissa according to the given rounding mode and precision. The fifth step renormalizes the mantissa, if required.

Symbolic model checking

Model checking faces two major challenges: capturing environmental constraints and managing capacity.

Environmental constraints. Many circuits work properly only under certain environmental constraints. For example, circuits associated with decoding instructions may require that their inputs be legal instructions, or a pipelined execution unit may require a certain delay between consecutive operations.

The floating-point adder's environmental constraints require that each operand is either zero (all exponent and mantissa bits are zero) or normal (the exponent is between 0 and maximum, and the mantissa has the form $1.b_1b_2b_3 \ldots b_m$).

```
let isAdd opcod  = (opcod = FADD);
let isSub opcod  = (opcod = FSUB);
let LegalInput   = (isAdd opcod OR isSub opcod) AND
                   (LEGAL_FP A) AND (LEGAL_FP B);
```

Figure 5. Floating-point adder's environmental constraint.

The FL function **LEGAL_FP** expresses these constraints, as shown in Figure 4.

For the floating-point-adder verification, we are interested only in the circuit's response to **FADD** and **FSUB** opcodes. The environmental constraint shown in Figure 5, therefore, restricts the opcode and ensures that operands **A** and **B** are either zero or normal.

In addition to constraints on data and the opcode, various other conditions typically restrict the input and state spaces for individual verification runs, such as excluding cases that are already known to be incorrect or that are not yet included in the specification.

Capacity management. Once full symbolic model checking has begun, BDD sizes start exceeding the tool's capacity limits. Good variable orderings prevent immediate BDD explosion, but a top-level decomposition strategy to combat verification complexity quickly becomes

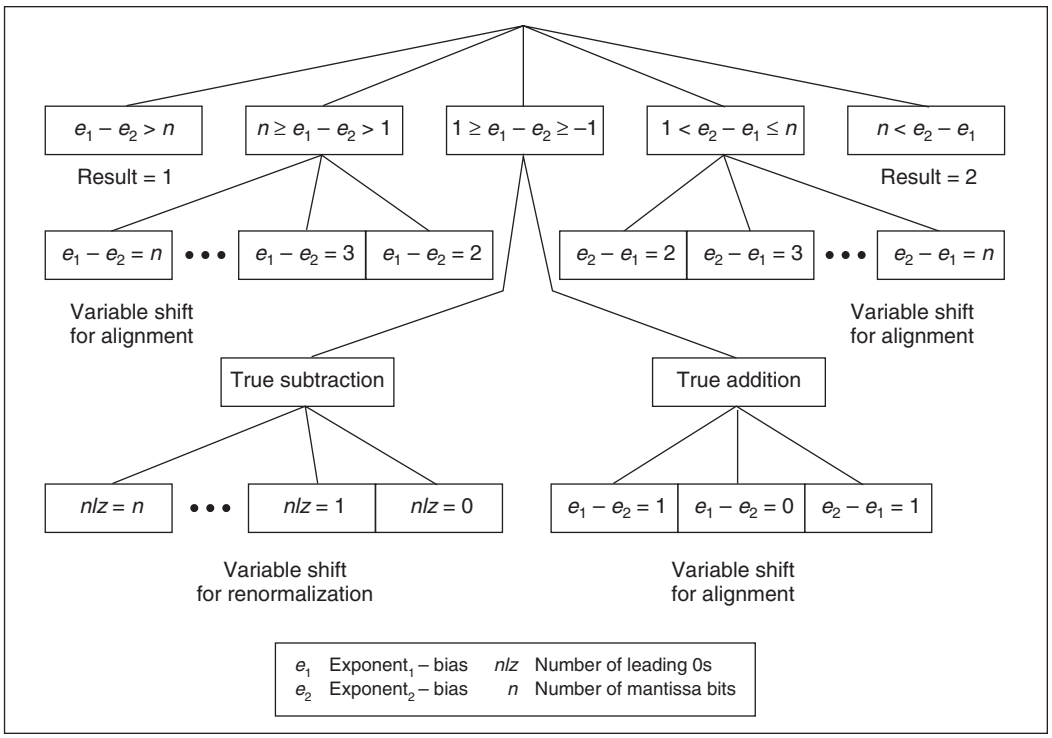

Figure 6. Case analysis for floating-point addition and subtraction.

crucial. Typical decomposition strategies include case splitting, structural decomposition, induction, and algorithm-specific techniques.

For the floating-point adder, we divided the verification into several hundred cases, according to the difference between the two exponents, as shown in Figure 6. Each case required a different BDD variable ordering. We easily generated the case splits and variable orderings using an FL script. By iterating over the difference between the exponents, the operation performed, and the number of leading zeros in the unnormalized mantissa, the script builds a Boolean predicate and variable order for each case.

During symbolic model checking, we invoke STE through a verification script built atop the reference model and circuit API. The inputs to a verification function usually include

- one or more Boolean conditions prescribing case splits and environmental constraints,
- necessary information for generating the BDD variable ordering, and
- auxiliary flags to control counterexample generation and the STE algorithm.

For each case, the verification script installs the BDD variable order, calls the model checker, checks the circuit result against the reference model, and optionally creates a pass/fail entry in a log file. The script can also generate counterexamples or other debugging information for the user.

Theorem proving

Our first theorem-proving goal was to prove that the case analysis just described covered all possible legal situations and hence that our verification's decomposition into several hundred model-checking runs was sound. To prove that our case analysis was exhaustive, we computed the disjunction of the legality conditions for all cases. Because each condition is represented as a BDD in FL, a simple BDD computation sufficed to reduce the disjunction to TRUE.

Our second goal was more ambitious: verify the FL reference model against the significantly more abstract IEEE standard for floating-point addition. Our formalization of the standard had four cases, one for each rounding mode mandated by the standard. In each case, we compared reference model output *result* with the infinitely precise result, $in_1 + in_2$. For

example, when rounding toward $-\infty$, our formalization said

$$\begin{pmatrix} result \le in_1 + in_2 < result + ulp \\ \wedge \\ \begin{pmatrix} result\_sign = -1 \wedge result\_mantissa = 2^{precision} \\ \Rightarrow \\ in_1 + in_2 < result + \frac{1}{2} ulp \end{pmatrix} \end{pmatrix}$$

where *ulp* (unit of least precision) is the distance between consecutive representable numbers that share *result*'s exponent, and *precision* is the number of fractional bits in *result_mantissa*. The first conjunct says that the inputs' true sum, $in_1 + in_2$, lies between the computed result, *result*, and the next representable floating-point number, *result* + *ulp*, as required by the IEEE standard. The property's second conjunct addresses a special case: when *result* is negative (*result_sign* = −1) and the fractional portion of *result_mantissa* is zero, the exponent of *result* is one larger than the exponent of the next-largest representable value. In this case, the distance from *result* to the next-largest representable value is one-half the distance between the representable numbers that share *result's* exponent.

To verify our reference model, we grouped its five steps into two stages, as shown in Figure 3. Stage 1 includes alignment, addition and subtraction, and normalization. Stage 2 includes rounding and renormalization. For each step, we wrote assumptions about the inputs, identified properties to prove for the outputs, and selected auxiliary properties for the algorithm's subsequent steps. We combined properties of individual steps into stage-level lemmas. We concluded the proof by combining the lemmas for stage 1 and stage 2 to derive one theorem verifying the adder reference model's outputs, and hence the RTL, against our formalization of the IEEE standard. The theorem characterizes the reference model's input-output behavior and includes all rounding modes and precisions.

In this example, the theorem-proving phase produced two major results. First, it ensured that the decomposition from the top-level correctness statement to the model-checking runs was correct. In principle, a model checker with no capacity limitations could have verified this result. However, the second result goes much further, checking the reference model against the IEEE standard. Here, we derived a theorem beyond the expressive power of BDD-based specification languages. Although theorem proving is effort intensive, our methodology mitigates the cost by reusing the verified reference model to verify multiple implementations.

Meeting requirements

One key to satisfying our requirements was tightly integrating tools and capabilities in the Forte framework. This was made possible by Forte's architecture, which focuses all activities around a general programming environment that is augmented with specialized verification functions and libraries.

We addressed the requirement for *realism* in many ways, from the scalability of Forte's tools to the availability of STE simulations when specifications aren't available. For *transparency* and *soundness*, FL plays a central role. Users can frame specifications in FL in their own terms rather than using a fixed set of predefined constructs. All proof artifacts are embodied as FL source code. The methodology gives *structure* to large verification efforts by defining the main work stages and associated artifacts. Both a skeleton work plan and concrete best-practice models are available to users for their verification code.

The methodology achieves *incrementality* by using STE for debugging with targeted scalar simulation, potentially delivering results well before full model checking is attempted. The methodology's distinct stages and code structures also foster incrementality, and aid *recoverability*. Along with Forte's visualization tools, STE's simulation capabilities enable *debugging and feedback*. When a symbolic-model-checking run fails, STE can generate a counterexample, which the user can analyze through simulations. STE can also characterize the entire failure domain, which an advanced user can explore using FL.

The methodology's stages decouple model checking and overall problem decomposition. The beginning stages are *bottom up*, grounding the whole verification effort in concrete simu-

lations. The *top-down* activity of developing an overall proof strategy starts a bit later but then proceeds in parallel. This aspect requires understanding the algorithm, but the earlier bottom-up explorations help foster this understanding and set a definite target for decomposition.

The identification of circuit APIs as a separate artifact aids *reuse*. A circuit API decouples the specification of data path functionality (which can remain constant for a design's lifetime) from signal names and timing (which change frequently). We have reused the results of several verification efforts on subsequent designs, reducing the verification effort in some instances by up to 80%. As long as a design is live, any part of the verification code may have to be adapted to track design changes. But the structure our methodology imposes on proof artifacts helps localize changes textually, thus supporting our requirement for *regression*. For example, if a signal name or the interface protocol changes, the user typically needs to modify only the API. The rest of the verification script should run unchanged.

WE DEVELOPED OUR METHODOLOGY and its requirements over a series of case studies carried out at Intel, including verifications of an IA-32 instruction-length decoder;[8,9] IEEE compliance of many of the IA-32 floating-point instructions;[6] and other, unpublished studies.

Nevertheless, our methodology is not complete. It is a useful first step, but industrial hardware verification is very challenging, and much work remains in defining the methodology and the underlying technology. Although extensive literature on design methodology and some literature on design validation methodology are available, the literature on formal-verification methodology is sparse. Most reports of formal verification in industry have been limited to technical accounts of particular verification achievements or techniques. An exception is Pixley's account of applying model checking at Motorola.[10]

We hope this article helps motivate the continuing spread of formal verification in industry, particularly in the development of effective usage methodologies. ■

References

1. T Kropf, *Introduction to Formal Hardware Verification*, Springer-Verlag, New York, 1999.
2. C.-J.H. Seger and R.E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, no. 2, Mar. 1995, pp. 147-189.
3. M.D. Aagaard et al., "A Methodology for Large-Scale Hardware Verification," *Proc. Formal Methods in Computer-Aided Design*, *Lecture Notes in Computer Science*, vol. 1954, Springer-Verlag, New York, 2000, pp. 263-282.
4. R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, Aug. 1986, pp. 677-691.
5. C.-J.H. Seger, *Voss: A Formal Hardware Verification System User's Guide*, tech. report 93-45, Dept. of Computer Science, Univ. of British Columbia, 1993.
6. J. O'Leary et al., "Formally Verifying IEEE Compliance of Floating-Point Hardware," *Intel Technical J.*, 1st Quarter 1999; http://developer.intel.com/technology/itj/.
7. J. Feldman and C. Retter, *Computer Architecture: A Designer's Text Based on a Generic RISC*, McGraw-Hill, New York, 1994.
8. M.D. Aagaard, R.B. Jones, and C.-J.H. Seger, "Formal Verification Using Parametric Representations of Boolean Constraints," *Proc. 36th Design Automation Conf.*, ACM Press, New York, 1999.
9. M.D. Aagaard, R.B. Jones, and C.-J.H. Seger, "Combining Theorem Proving and Trajectory Evaluation in an Industrial Environment," *Proc. Design Automation Conf.*, ACM Press, New York, 1998, pp. 538-541.
10. C. Pixley, "Integrating Model Checking into the Semiconductor Design Flow," *Electronic Systems Technology & Design*, Mar. 1999, pp. 67-74.

Robert B. Jones is a member of the technical staff at Intel's Strategic CAD Labs in Hillsboro, Oregon. His research interests include practical application of formal methods to hardware systems specification, architecture, and verification. Jones has a PhD in electrical engineering from Stanford University. He is a member of the IEEE and the ACM.

John W. O'Leary is a member of the technical staff at Intel's Strategic CAD Labs. His research interests include formal specification and verification of hardware. O'Leary has a PhD in electrical engineering from Cornell University. He is a member of the IEEE.

Carl-Johan H. Seger is a principal engineer at Intel's Strategic CAD Labs. His research interests include formal hardware verification and asynchronous circuits. Seger has a PhD in computer science from the University of Waterloo.

Mark D. Aagaard is an associate professor in the Department of Electrical and Computer Engineering at the University of Waterloo. His research interests include high-level digital system design and verification. Aagaard has an MSc and PhD in electrical engineering from Cornell University. He is a member of the IEEE.

Thomas F. Melham is a professor of computing science at the University of Glasgow in Scotland. His research interests include software architectures for formal-methods tools, industrial-scale hardware verification, combined model checking and theorem proving, and integration of formal verification into hardware design methodologies. Melham has a PhD in computer science from the University of Cambridge.

■ Direct questions or comments about this article to Robert B. Jones, Strategic CAD Labs, Intel Corp., JF4-211, 2511 NE 25th Ave., Hillsboro, OR 97123; rjones@ichips.intel.com.

How to Contact Us

Subscription Questions
IEEE Computer Society
PO Box 3014
Los Alamitos, CA 90720

Paper, electronic, or combination subscriptions to *IEEE Design & Test* are available. Send subscription change-of-address requests to address.change@ieee.org. Be sure to specify *IEEE Design & Test.*

Membership Change of Address
Send change-of-address requests for the Computer Society membership directory to directory.updates@computer.org.

IEEE Design & Test on the Web
Visit our Web site at http://computer.org/dt/ for article abstracts, access to back issues, and information about *IEEE Design & Test.* Full articles are available online to subscribers of the magazine's electronic version.

Writers
Author Guidelines and IEEE copyright forms are available from dt-ma@computer.org, or access http://computer.org/dt/edguide.htm.

Letters to the Editor
Send letters to Managing Editor, *IEEE Design & Test*, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720. Please provide an e-mail address with your letter.

Reprints of Articles
For price information or to order reprints, send e-mail to dt-ma@computer.org or fax to *IEEE Design & Test* at (714) 821-4010.

Reprint Permission
To obtain permission to reprint an article or column, contact William Hagen, IEEE Copyrights and Trademarks Manager, at w.hagen@computer.org.

Missing or Damaged Copies
If you did not receive an issue or you received a damaged copy, contact help@computer.org.

News Releases
Mail microprocessor, microcontroller, operating system, embedded system, microsystem, and related systems announcements to *IEEE Design & Test*, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720.

IEEE Design&Test of Computers