Pre-RTL Formal Verification: An Intel Experience

Robert Beers
EMG CCDO Design Validation
Intel Corporation
JF4-451
2111 NE 25th Ave
Hillsboro, OR 97124
robert.beers@intel.com

ABSTRACT

During the recent development of a next-generation Intel processor, the project's formal verification team verified a new coherence protocol and portions of its RTL implementation against the protocol's specification within project deadlines. Typically, FV teams apply formal property verification (FPV) after RTL is coded and, though it continues to be an effective complement to pre-silicon validation, this late application prevents it from keeping pace with the continual complexity increases in hardware designs. Our discussion centers around how applying FV early in the development cycle of this processor enabled continual verification as the design progressed, culminating with the targeted RTL verification. We also present the languages and methodologies used, the reasons behind the choices, and where improvements can be made.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*verification*

General Terms

Algorithms, Design, Verification

Keywords

Formal verification, protocol verification, microarchitecture verification, TLA+, TLC, explicit state enumeration

1. INTRODUCTION

Intel has been active in formal verification for over 11 years[3, 13]. Research teams have verified common components in Intel processors and several high-level protocols and algorithms destined for Intel products [8]. Development teams have applied formal property verification (FPV) to a continually increasing number of processor and chipset designs as a means of complementing and supplanting dynamic validation. Particular emphasis has been given to floating-point verification where standardization and ingenuity have led to the creation of a framework connecting RTL verification to the IEEE specs[9, 10].

Outside this specialized area, attempts to expand formal efforts to similar scope have run into the usual difficulties.

- Architectural features lack standardized specifications and evolve significantly on successive projects.
- Projects apply FV only after RTL delivery, paralleling the dynamic validation engagement process.
- Design development deadlines afford little time for the investment necessary to fully verify a feature.

This paper discusses how an Intel FV team created a verification framework for a new coherence protocol and applied it during development of a next-generation processor. As the development progressed from protocol to project architecture, to microarchitectures, and to RTL, they continually applied the framework and verified the project's protocol implementation as its details emerged. Applying the framework before RTL coding prevented protocol and microarchitecture bugs from reaching the RTL and allowed effective RTL FV to expose remaining design bugs. Areas of the design covered by the framework demonstrated quicker convergence trends and lower bug rates than the rest of the design though they were new microarchitectures.

Validation and verification traditionally occur after RTL is available, more than half-way through the project timeline. On the project discussed here, formal verification occurred throughout project development. However, this discussion is about more than just applying FV early in the project to verify architecture features and microarchitectures. It is about verifying them during their development instead of afterward. Approaching project FV this way overcame the difficulties RTL FPV faces in keeping up with Moore's Law.

The remainder of this paper is structured as follows. Section 2 provides relevant background on product development stages and the TLA+ specification language. Section 3 covers the verification steps in the framework through the pre-RTL stages. Section 4 describes the ways in which the framework affected RTL verification. Section 5 gives results. Section 6 draws conclusions from the experience.

2. BACKGROUND

Given how different this formal verification effort is from typical FV efforts on RTL or existing protocols or algorithms, some background information is necessary to provide frames of reference. This section describes the following:

DAC 2008, June 8–13, 2008, Anaheim, California, USA.

The project's development stages to establish the abstraction levels FV encountered during the project, the engineering process at work in all stages to clarify where the FV activity engaged in the process, and the TLA+ specification language and TLC tool to provide a basis for the FV work that took place.

2.1 Project Stages

Prior to first fabrication, there are four development stages most relevant to this experience: planning, architecture definition, microarchitecture design, and RTL development. Activities in the three pre-RTL stages are as follows:

Planning Primarily a company-wide effort rather than being associated with a particular project. Proposed architecture technologies develop and await projects to intercept and join for implementation.

Architecture definition Architects define what the product is, including subsets of features it supports, and then identify necessary microarchitecture blocks and behaviors each must handle.

Microarchitecture design Designers establish the state elements each microarchitecture block contains and devise pipelines through the elements to perform the required behaviors.

Engineers use the same design process in each of these pre-RTL stages. First, the engineers, who are active in the stage, identify and study their requirements as defined by the output from the previous stage (except for Planning stage). Next, the same engineers create an initial implementation and enter an iterative process of mapping the requirements upon it and refining areas where it is deficient. Lastly, when the engineers and their peers are confident that the implementation satisfies the requirements, they document it as the stage's specification. Because specifications are delivered at the end of a stage, doing FV during a stage implies working with implementations in development.

2.2 TLA+ and TLC

Before RTL, there are no models to formally verify. The FV team created its own FV models in Lamport's TLA+ specification language[11]. TLA+ is an extension to TLA, which is based upon first-order logic and set theory and has temporal operators to describe liveness properties. TLA+ adds definitions, support for large formulas (LET/IN construct, bulleted disjunction and conjunction), and modules to support large specifications.

A TLA+ model consists of an initial-state predicate, a next-state relation, fairness constraints, and properties the system (described by the other three components) must satisfy. In TLA+ parlance, related groups of behaviors in the next-state relation are called actions and the state elements are variables. Each of the actions' behaviors are described as guarded relations where the guard predicate asserts for applicable inputs and current states and the rest of the relation defines acceptable next states and outputs. The following action describes a counter, `cnt`, that can increment when an incoming command is "inc" and the counter has not reached its maximum:

```
CountInc == /\ Cmd = "inc"
            /\ cnt < maxcnt
            /\ cnt' = cnt + 1
```

TLA+'s expressiveness allows the structure of the next-state relation to take on any form. The FV work described here followed Lamport's suggestion of interleaving all actions to simplify the models yet allow all possible behaviors.

TLC is the evaluation tool for TLA+[14]. It supports simulating TLA+ models and model checking them using explicit state enumeration. TLC requires only the system description (initial state, next-state relation, and optional liveness constraints) and configuration values for the model constants. In model checking mode, TLC explores the entire state space by starting from the set of possible initial states and finding successor states by repeatedly applying the transition relation. If it finds a state in which the transition relation evaluates to false, meaning the model is incomplete or the system has a deadlock condition, TLC fails with a trace leading to the state. TLC also uses relations from the model as invariants and properties, which it checks on every found state. For a violating state, TLC provides the trace leading to it.

3. FRAMEWORK CREATION

While Intel planners were developing a new coherence protocol, they asked the CCDO FV team to formally verify one of the proposals. Though the team had little experience verifying protocols and had only worked with RTL until then, they agreed to do the verification to broaden their experience and eventually enable RTL verification directly against the protocol. Instead they ended up with a methodology for applying FV in each development stage that generates a verification framework and enables efficient RTL FV. The rest of this section describes their pre-RTL verification work and how the framework evolved during the project's development.

3.1 Protocol FV

When the FV team joined the protocol verification effort, two other teams were also looking at proposals. A research group, with extensive protocol verification experience, was using the Murϕ verification system to model and verify one proposal[1]. Another development team, with significant experience developing protocols, was using TLA+ to create and model a new protocol proposal and verifying it with TLC[2]. The FV team took the approach of creating an executable version of their proposal and then doing a complete proof in the HOL theorem prover[12, 5]. They used perl to model and simulate the proposal.

All three teams identified a number of bugs in their respective protocols and helped the planners solidify their proposals; however, only the Murϕ and TLA+ teams were able to complete the verification within the time allowed. Analysis of the teams' efforts concluded the following: 1) Complexity and maturity of the proposals were similar, 2) Model development progressed at similar rates, 3) Using the Murϕ or TLC model checker exposed bugs sooner than did simulating, 4) Bug fix analysis was quicker and more thorough (no lingering corner cases) when model checked, and 5) The HOL team's approach resulted in a model-then-verify process whereas using Murϕ or TLA+ with TLC combined the tasks into a single step.

Based upon various criteria, the planners selected the proposal modeled in TLA+. Seeing the success of the other development team with TLA+ provoked the FV team to work with them in creating a TLA+ protocol model both teams

could use on their respective projects. First they made sure the shared protocol model had no project-specific details and was, thus, a specification for both projects' implementations. Next they used TLA+'s modules to divide the protocol's state elements into logical blocks for the caching agents, link layer, coherence master, and physical memory controller. Using modules allowed both teams to replace abstract blocks with blocks containing project-specific details. Thus, by the end of planning, they had a verified protocol and the beginnings of a TLA+ verification framework for it.

3.1.1 Verification Methodology

Using TLA+ modules to allow replacing blocks in the abstract model with blocks containing project-specific details dictated the choice of verification methodology in the TLA+ framework. The FV team was familiar with the methodology of demonstrating that an agent in the abstract protocol model exhibits every behavior the implementations exhibit. However, that methodology would require converting TLA+ to another language and then using an FV tool other than TLC. A verification methodology, to which TLA+ modules and the TLC model checker are ideally suited, is conformance[6, 7]. In conformance, or I/O equivalence checking, the protocol model is verified in TLC and then acts as a TLC verification environment for the detailed blocks. Thus, the abstract model is verified twice, once with all its abstract blocks and once with one (or more) of its abstract blocks replaced by detailed blocks. To ensure correctness, whenever one of the abstract blocks used in both verifications changes, the modeler must run TLC on both models.

A drawback to this approach is state space explosion. The protocol model by itself generates billions of states. Replacing one of its abstract blocks with a few detailed blocks has a multiplicative effect. As with all FV tools, these limitations require model manipulations to fit the proof into available resources. One common manipulation is verifying with several partial configurations instead of one complete configuration, for example, verifying the full protocol with a small number of agents and then verifying larger groups of agents with subsets of the request types. Using partial configurations risks not covering all possible cases that can occur in the complete configuration. The FV team relied upon them only when absolutely necessary and only when months of working with the models provided enough confidence in their soundness. To further increase confidence, they ran large, complete configurations of the models in TLC's simulation mode.

3.1.2 Visualization

Distributed protocols or algorithms are difficult to understand and even harder to explain. Discussing a protocol's constraints or a failing case in the protocol requires someone to describe a scenario well enough for everyone to comprehend it and draw the same conclusions. Even when described perfectly, keeping attention and following along is nearly impossible for someone familiar with the system. (Formal verification of distributed systems is so useful because of the FV tool's ability to explore all possibilities without having to comprehend them.)

Discussions with the engineers about problem scenarios inevitably required a whiteboard and protocol drawings. The reason is simple: What can require 15 minutes to understand through listening can be understood immediately with a 1-minute diagram. Even the verifier familiar with the protocol and TLA+ model, when looking at a TLC counterexample, often draws a diagram to understand what the system is doing and narrow down possible root causes. Either a realization of these patterns or a request from the engineers led to the development of a tool, Prograph, to automatically create communication graphs from the counterexample TLC generated. Most notably, Prograph greatly increased FV debug efficiency. Architects and protocol developers appreciated having the diagrams to study and save. Prograph also allowed for creating diagrams of interesting *valid* scenarios by constraining the model's transition relation to guide TLC's state exploration and then triggering an artificial failure on the scenario's completion state.

3.2 Project Pre-RTL FV

At the end of the planning stage, the FV team had a complete TLA+ model of the cache coherence protocol, substantial experience with and knowledge of the protocol, and a framework and methodology for verifying project-specific details. When architects worked on the project's architecture definition, the FV team applied the framework and verified the features by the end of the stage. When designers worked on the feature's microarchitecture designs, the FV team applied the framework, now extended with models for the features, and verified microarchitectures by the start of RTL coding. In both stages, the FV team paralleled the engineering processes. As discussed in Section 2.1, the process in both stages is the same, thus the stages' verification efforts were similar though at different levels of abstraction.

In both stages, the first task for the incoming engineers (architects and designers) is to study and learn the specifications. Normally this is done only with the help of the previous stage's engineers. The presence of an FV team who had modeled the specifications and verified them offered a second group of teachers, ones with significant experience seeing the protocol and features working (or not working when bugs were found). Though each stage started with the two groups of engineers exchanging knowledge, when trickier cases were discussed the FV team provided examples and expounded on the general constraints and assumptions. Ultimately, this led to a three-party system of sharing knowledge and scrutinizing implementation ideas.

Assisting the engineers with learning the specifications exposed the FV team to the engineers' initial thoughts on possible implementations. When the implementation refinement iterations occurred, FV provided feedback from analyzing them against the (protocol or feature) specifications. Feedback during this phase, though based upon past FV rather than modeling and verifying current ideas, was useful because the engineers are analyzing numerous tradeoffs in complexity, size, number of flows or pipelines, state elements, etc. The FV team used the knowledge gained from prior stages to provide examples that countered expected savings or to suggest ways to simplify algorithms or take advantage of unconstrained areas of the protocol. The more common case was showing why a savings would not be realized because, as validators, the FV team is more attuned to detecting, what are essentially, bugs.

3.2.1 Model Development Methodology

Working closely with the engineers allowed the FV team to judge when aspects of the proposed implementations were

stable enough to begin modeling them. At the beginning of the project, the FV team had no experience creating their own TLA+ models, having borrowed and rewritten the TLA+ protocol model from the other development team. Borrowing from them again, the FV team devised a modeling methodology relying upon TLC results to introduce concepts in an ordered manner:

1. Determine a concept-introduction order. Because the goal is to model all concepts, the order does not matter; however, introducing concepts that are most relevant to the feature first provides the best means for learning and scrutinizing the implementation.
2. Add first concept and its initial state definition to model.
3. Run TLC model checker.
4. When TLC fails with incomplete model, expand model to handle failing state and return to step 3.
5. When TLC passes, implying model is complete for concept, add next concept on list and return to step 3.

Development of the project's architecture model serves as an example of this methodology. The first concept introduced was the highest level of the cache hierarchy. After adding its variables, their initial state definition, and a stimulus to read the cache level, running TLC resulted in an immediate failure for a cache read to an invalid line. Because TLC failed with an incomplete model, the model received an action to spawn an outbound read request, which required adding an outgoing request buffer variable and its initial state definition. Next, TLC failed on the state where the buffer wanted to send a request, which required adding an action to generate a request packet. This iterative loop continued until TLC finally passed, meaning the model could handle all cases involving a read to the cache level. Adding the next concept presented a choice between a lower level of the cache hierarchy or a second caching agent. A lower level of cache requires modeling the cache policy. A second agent generates external requests and provides a separate cache hierarchy to process requests. The team chose to add the lower cache level followed by the second agent, and they separated the second agent into two concepts by not allowing it to generate requests until TLC passed with it handling requests.

3.2.2 Project Modeling

In both stages modeling began soon after the engineers' learning phase because experienced architects and designers excel at identifying basic flows and constructing implementations for them. It was imperative that FV model these first fundamental concepts as soon as the engineers described them. The reason, which was noted during the architecture stage and reinforced during the microarchitecture stage, is that modeling and verifying these implementation *foundations* early helped build solid footing for the rest of the stage's development. Modeling and verifying the ideas in their infancy thrust them under the FV microscope, exposing new flows to consider, incomplete definitions, states with ambiguous decisions, fundamental limitations of the ideas in satisfying the specifications, resource constraints, complex or lengthy algorithms, and other manner of "bugs." Here are a few examples:

New flows Verifying the cache policy exposed combinations of states thought to be impossible. After architects analyzed the flows they updated their legal cache states tables to match and provided ideas on how the cache hierarchy would handle the flows.

Ambiguous states Databuffer control analysis discovered two flows involving a buffer receiving a read request, where one flow required the buffer to deallocate and the other required it to wait for a write, yet neither the databuffer nor the read sender had sufficient information to disambiguate the cases. Analysis of the flows (and others related to them) pointed to a third agent that had the needed information and made an earlier request in all cases.

Fundamental limitations The protocol relies upon certain messages and a linking algorithm to resolve conflicts; however, the architects opted to try an address-matching approach. For all but two groups of message patterns it worked with straightforward algorithms. Handling these two groups required special algorithms (garnering their own nomenclature) that required a month to fully grasp, accept, resolve, and verify yet exist on silicon as they were defined then.

Quickly verifying the fundamental implementation ideas solidified them by producing an intense, ordered succession of questions about them. The architects and designers gave their complete focus to answering these questions because they understood the significance in getting the basic mechanisms correct. In one microarchitecture case, the designers realized that the stream of issues and questions coming up indicated a primary state machine was not converging. They called for focused meetings where all architects, designers, validators, and verifiers involved with the microarchitecture worked together all-day everyday going through all the flows and not stopping until everyone was convinced they had created a state machine that handled them all. After four weeks of continual meetings they had a solid state machine for the microarchitecture that, other than two additional states for handling special cases, remains the same three years later.

Modeling through the remainder of each stage, in comparison, was less intense though more active. Getting the basics working and verified required the most engineering iterations. The knowledge everyone gained during these iterations was substantial due to all of the examples studied and conclusions made. Most importantly, everyone acquired an intuitive understanding of how the implementation works and why it works correctly. Building the rest of the implementation was less fraught with unknowns as a result. Thus, modeling continued at a brisk pace to keep up with the invigorated architects or designers and verification generally found only *shallower* bugs.

3.2.3 Pipelines

During the microarchitecture design stage, FV proceeded as it had in the architecture definition stage. However, a new design feature had to be considered: pipelines. Pipelined logic, with reads and updates to state elements in separate pipestages, implies overlapping, non-atomic actions, which challenges the methodology of modeling the behaviors as atomic transitions. Fortunately, the methodology of mimicking the engineers provided a solution again. Designers

tend to decompose pipeline development into three steps. First they identify pipelines based upon messages coming into the design or being passed around within it. Second they create "stops" along these pipelines where the state elements exist and where the messages update the elements and send new messages down the same pipeline or other pipelines. Third, they turn the stops into pipelined logic. At their second step, the read-update behaviors at the stops are atomic. Thus, a microarchitecture model groups state elements according to the stops, defines transitions according to the pre-pipelined behaviors the designers envision, and sends messages among the groups according to what is sent along the pipelines between stops.

Modeling according to pipeline stops was sufficient to verify the microarchitectures. It also identified a number of pipeline hazards between stops, where a stop in the pipeline would update a state element that an earlier stop would read. However, because it treats the stops as atomic, it does not identify hazards within them. The FV team experimented with using models to identify possible hazards by enumerating the possible streams of messages that could be active within a stop in successive pipestages. In general, though, the designers and modeler had acquired enough knowledge by then to identify most hazards based upon the read and update pipestages. Dynamic validation found a few others in early testing, and RTL FV found one involving a state element updated in two different pipestages depending upon the incoming message type.

3.2.4 Testbed

Beyond the end of a stage, the FV team continued working on the stage's model because of inevitable changes. Causes include performance improvements, simplifications to the implementation, physical design constraints, addition of features, and, of course, bugs. The engineers came to rely upon the FV knowledge gained from working on the model as an initial check on proposed changes. Many proposals did not make it past this stage and the modeler created flows explaining why and offered options if any were apparent. Also, by checking with FV initially, the engineers made it possible to verify proposed changes before commitment.

4. RTL VERIFICATION

Model development tapered off near the start of RTL coding—testbed modeling for proposed changes continued. Thus, the FV team turned some of their attention to RTL verification and assisted the project in the following ways:

- Applying targeted RTL FPV based upon microarchitecture model transitions and knowledge from microarchitecture verification.
- Working with and joining validation teams to streamline and attain coverage goals.
- Formally verifying RTL against the microarchitecture model, to free up validation resources in that area.

In two microarchitectures, FV developed mature enough models to consider verifying RTL against the model. One had logic that was extremely important for the protocol's correctness but was situated such that it would be difficult to exercise completely via random testing. During RTL coding the FV team proposed formally verifying this logic to supplant validation coverage efforts.

Their approach, which relied upon the association between TLA+ actions in the microarchitecture model and "stops" in the design's pipelines, was to demonstrate that a "stop" in the RTL behaved according to its TLA+ action. For this they developed a framework for mapping each of the TLA+ actions onto the appropriate pipeline stages and then verified RTL correspondence to the TLA+ description. The proof methodology, termed "lightweight flushing," is based upon Burch-Dill pipeline flushing[4]. For each action disjunct, the framework drops variables upon the input RTL signals, flushes them to the current state pipestages, drops variables on the state elements, and then flushes everything through to the next state and output stages. It then lifts the input, current state, next state, and output values from the RTL evaluation up to the TLA+ level, flushes the TLA+ for one cycle to effect the specification, and compares the results. Because the microarchitecture was modeled behaviorally in TLA+ (essentially an enumeration of cases the microarchitecture handles), verifying the cases depended upon architecture and microarchitecture assumptions made by RTL. Identifying the assumptions was an iterative process of analyzing the RTL failure, proposing an appropriate assumption, attempting the RTL verification with it, and verifying the assumption in the microarchitecture model.

Owing to the number of cases this logic had to handle, FV against its microarchitecture model found four hard-to-hit bugs in the RTL. One example involved a piece of logic that existed for a common case scenario but also had to catch two subtle cases. It worked for the common case and the less subtle of the other two cases, but for a three-way conflict case that could occur in only one of the supported configurations and only with some drastically delayed messages, it generated the correct output but incorrectly transitioned a state machine. FV developed this framework and verified the logic block in all product configurations with two quarters remaining before fabrication.

5. RESULTS

Quantifying the success of the pre-RTL FV efforts is difficult due to the verification taking place before RTL—bugs are tracked only against RTL—and on implementation ideas and proposed changes as a means to further develop them. The following is an assortment of success indications. The FV team filed 45 "issues" that the project used before RTL to track engineering problems requiring special attention to solve. A design lead for one of the verified microarchitectures, when asked for about the FV work, said, "They found hundreds of bugs before RTL and played an important role in creating a stable microarchitecture." This microarchitecture had the lowest ratio of bugs per line of RTL even though the unit was new. On RTL and silicon there have been no coherence protocol or architecture feature bugs. In the areas covered by pre-RTL FV, only one microarchitecture bug has been found on silicon, attributed to accepting an unfounded claim that the affected logic would never interact with particular microarchitecture protocols.

A historical cost-versus-saving analysis is not available because all pre-RTL work was done in new areas. The protocol and architecture were entirely new, as were the five microarchitectures worked on. Though the lower bug counts likely reduced the effort overall, to what extent is unquantifiable. One design lead's experience can provide a rough datapoint. The design lead managed two units that stemmed from the

Table 1: Project Resources (person-years)

Project Task	Verified unit	Non-verified unit
Planning FV	0.5	0
Architect	0.5	0.75
Architecture FV	0.5	0
Microarch. Design	1	1.25
Microarch. FV	1	0
RTL Design	1.5	2
RTL FPV	1	2
RTL Validation	1.5	3.5
Total	7.5	9.5

coherence protocol initiative. Only one received FV attention before RTL (due to resource limitations). The design lead has pointed out that, though the units are of similar size and complexity, designers on the verified unit were working on other projects while designers of the other were fixing bugs in silicon. Table 1 compares the units' person costs for the development tasks prior to fabrication.

6. CONCLUSIONS

Formal verification during the earliest development stages of a next-generation Intel processor presented the project's FV team a unique opportunity to verify a coherence protocol and its implementation. The protocol verification work led to the creation of a TLA+ verification framework for the protocol and a modeling and verification methodology within the framework. FV applied the methodology in parallel to the design process in the project's architecture and microarchitecture stages. In each stage they shared knowledge gained from prior FV work and modeled and verified implementation ideas as the engineers generated them. FV early in the stage intensified scrutiny of the ideas, finding many bugs before they escaped to subsequent stages and helping solidify the implementation before the next project stage. In effect, the FV team became a design aid in addition to an implementation verification team. Pre-RTL FV kept bugs out of the RTL, solidified microarchitectures before RTL, and enabled efficient RTL FV.

Even with the successes more areas for improvement are apparent. Dynamic validation's benefits from the FV efforts paled in comparison to architecture and design benefits—more ways to leverage the models and develop validation collateral must be found, especially as behavioral checkers. Better, less risky ways of managing state space explosion and model abstractions are needed, as are improved technologies for liveness analysis of these systems within project timelines. Merging models from different features for combined verification also presents unique challenges.

Regardless, this experience showed that FV can keep up with continual increases in design complexity. The two most important enabling factors are to apply formal verification during feature development and to follow the developers', architects', and designers' leads as they decompose the problem toward an implementation.

7. ACKNOWLEDGEMENTS

Many architects, designers, and validators deserve hearty gratitude for their acceptance of and support for the FV work they let us do for them. I especially wish to thank Derek Bachand, Jeremy Coriell, Dave Hill, Bob Safranek, and Aimee Wood for their support. I must thank Brannon Batson and Ching-Tsun Chou, though thanks is hardly enough, for sharing all of their expertise and guidance and, thus, making this experience possible. I would also like to thank Jesse Bingham for his exceptional work on the RTL formal connection and Annette Bunker for all her great suggestions from countless draft reviews.

8. REFERENCES

[1] M. Azimi, C.-T. Chou, A. Kumar, V. W. Lee, P. K. Mannava, and S. Park. Experience with applying formal methods to protocol specification and system architecture. *Form. Methods Syst. Des.*, 22(2):109–116, 2003.

[2] B. Batson and L. Lamport. High-level specifications: Lessons from industry. In *Formal Methods for Components and Objects*, volume 2852/2003 of *Lecture Notes in Computer Science*, pages 242–261, 2003.

[3] B. Bentley. Validating the Intel® Pentium® 4 Microprocessor. In *Proc. of the 38th Design Automation Conf.*, pages 244–248, Las Vegas, NV, June 2001.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV)*, pages 68–80, 1994.

[5] A. Camilleri, M. Gordon, and T. Melham. Hardware Verification Using Higher-Order Logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. North-Holland, 1986.

[6] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. PhD thesis, Carnigie Mellon University, 1988.

[7] G. Gopalakrishnan, E. Brunvand, N. Michell, and S. M. Nowick. A Correctness Criterion for Asynchronous Circuit Validation and Optimization. *IEEE Trans. on Computer Aided Design*, 13(11):1309–1318, November 1994.

[8] J. Harrison. Formal verification of square root algorithms. *Formal Methods in Systems Design*, 22:143–153, 2003.

[9] R. Kaivola and M. Aagaard. Divider circuit verification with model checking and theorem proving. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 338–355, 2000.

[10] R. Kaivola and N. Narasimhan. Formal verification of the Pentium® 4 floating-point multiplier. In *Design, Automation and Test in Europe Conf. and Exposition (DATE)*, pages 20–27. IEEE Computer Society, 2002.

[11] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addision-Wesley, 2002.

[12] T. F. Melham. *Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic*. PhD thesis, University of Cambridge, 1990.

[13] T. Schubert. High level formal verification of next-generation microprocessors. In *Design Automation Conference (DAC)*, pages 1–6, 2003.

[14] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 54–66, 1999.