High Level Formal Verification of Next-Generation Microprocessors

Tom Schubert
DPG CPU Design Validation
Intel Corporation
JF4-451
2111 NE 25th Ave
Hillsboro, OR 97124-5961

Tom.Schubert@intel.com

ABSTRACT

Formal property verification has been an effective complement to pre-silicon validation of several Intel® Pentium® 4 CPU designs at Intel Corporation. The principal objective of this program has been to prove design correctness rather than hunt for bugs. In the process, we have evolved our tools and methodology and are now applying FPV techniques to protocol level properties. Moving forward, new technologies such as GSTE and SAT offer the potential to significantly increase the scope of what can be formally verified. This paper will discuss the application of FPV to validation of the Intel ® Pentium® 4 microarchitecture and some approaches being considered to broaden the application of FV techniques, particularly at a higher level of design abstraction.

Categories and Subject Descriptors

B.5.3 [**Register-Transfer-Level Implementation**]: Design Aids – *Verification.*

General Terms

Design,Verification.

Keywords

Formal Property Verification.

1. INTRODUCTION

Microprocessor designs continue to increase in complexity with the demands to provide higher performance at lower power. Historical data from Intel IA-32 processor design projects suggests a 3 to 4 fold increase in detected pre-silicon logic bugs per IA-32 generation [2] (see figure 1). Advances in methodology and tools notwithstanding, the impending challenge to validation in future projects is clearly considerable.

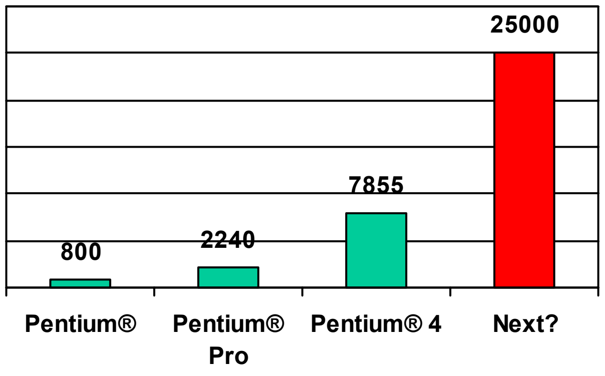

Figure 1. Pre-silicon logic bugs per generation.

Formal Property Verification (FPV) played a significant role in the Pentium® 4 pre-silicon validation effort. From our experiences, it is increasingly clear that improved FPV techniques and methodologies coupled with greater synergy with traditional simulation-based validation techniques will go a long way in meeting this challenge..

The Pentium® 4 Processor was the first in a line of CPU designs at Intel on which formal property verification was applied on a wide scale. The approach employed was not for FPV to replace any traditional simulation-based validation work, but to augment pre-silicon validation by focusing on proving the total correctness of high risk functionality. Design validation would catch the bulk of the bugs quickly while FPV would follow up, "leaving no stone unturned," to ensure that no corner cases remained. We developed our tools and methodologies with this objective in mind and are fast reaching a stage where we can apply our techniques to verifying challenging chip level protocols. Cutting edge next generation FV tools based on STE, SAT and traditional model checking techniques will enable us to significantly stretch the microprocessor verification solution space.

Our FPV results have been very encouraging. For the Pentium® 4, FPV was primarily responsible for verifying the correctness of the entire arithmetic logic. We were able to ensure 100% design coverage in these cases. Furthermore, FPV also concentrated on verifying complex chip level protocols which sometimes were beyond the scope of effective simulation-based validation. The pre-silicon effort exceeded its targeted goals of verifying the 25% most critical properties in two-thirds of the chip. Properties in the remaining third were verified after initial silicon. To date, there have been no design errors detected in areas of the chip that have been formally verified. During the course of the FPV program, a number of difficult to discover design errors were detected. Some of these errors were classified by authoritative experts in the project as probable pre-silicon escapees had FPV not been used in pre-silicon validation. Prior to first silicon, FPV uncovered 18 such critical bugs in the design, thus reducing the cost of post-Si debug or possible escape to customers. Some of these included:

- Floating Point Multiplication rounding error
- Floating Point Adder incorrect carryout
- Instruction Length Decoder incorrect length for an illegal instruction in a specific alignment
- Microinstruction sequencer executed on wrong target after misprediction (branch recovery "safety" property)
- Trace Cache dropped valid data - deadlock (proof that data is neither created nor destroyed)

The pre-silicon effort generated a large database of specifications (some 14,000), proof strategies, and proof system infrastructure. While the entire database hasn't been reusable on new designs, significant portions of it have been reused and new proofs have been constructed in considerably less time (less than ½ the original time). The floating point proof infrastructure in particular has been highly reusable, both on Pentium® 4 proliferations and on other processor designs at Intel. This reuse also identified a number of high quality bugs. In the process, numerous lessons were learned and are now being applied to improve the effectiveness and productivity of subsequent FPV efforts.

The remainder of this paper will first review the Intel® Pentium® 4 processor formal verification program methodology and the techniques we found useful. We will then discuss new tools and ideas we are considering for improved FPV productivity on future microprocessor projects.

2. FPV METHODOLOGY

Most of the non-floating point verification work was done with an LTL model checker tool and a bounded model checker developed at Intel [3,4]. We have found bounded model checking effective as a first step to speed up total correctness proof development by eliminating some of the reverse engineering work required to flesh out correct property specifications. To complete floating point verification, the Forte verification system was used. Forte is a combined model checking (STE) and theorem proving system built on top of the Voss system [5]. These tools support various abstraction mechanisms to reduce the design complexity for model checking, but significant human effort is necessary to guide the proof process.

Evolving methodology was particularly critical for successful application of our formal verification tools to industrial use. There were a number of environmental factors that needed to be considered:

- Our task required that we work with a very low-level netlist RTL. This enabled designers to optimize performance, but at a cost of simplicity and clarity.
- Properties were not selected based upon what was easy to formally verify. The FPV program's objective was to verify the most critical properties. Even with human ingenuity, tool capability limitations set limits on what could be verified.
- Proofs needed to be robust. Designs changes are frequent and low-level. The team's methodology expected proofs would be maintained on a regular basis and reused across CPU projects.

The following subsections will briefly describe techniques we found effective for verifying floating point operations and more control centric functionality. Greater detail can be found in [1],[6],[7], and [8].

2.1 Verification of Floating Point Arithmetic

Formal verification provides the only practical means of checking arithmetic operation results for the complete operand data space. The FPV team developed a conceptual framework that integrates theorem proving and model checking techniques to formally verify the correctness of all Pentium® 4 floating-point micro-operations from RTL to an IEEE specification. This work covers multiply, divide, remainder, square root, addition, subtraction, and a set of auxiliary micro-operations. The basic approach taken was to insert an intermediate model between the high level specification and the low level RTL (see Figure 2). This intermediate model consists of two parts: a reference model and a circuit API. The circuit API is the glue between the reference models and the implementation specific RTL. It adds design specific temporal control information about signal names, timing, etc. The reference model is purely algorithmic and has been reused for several different design implementations. STE was used to prove the correctness of the RTL with respect to the intermediate model and theorem proving was used to relate the intermediate model to the high level specification.

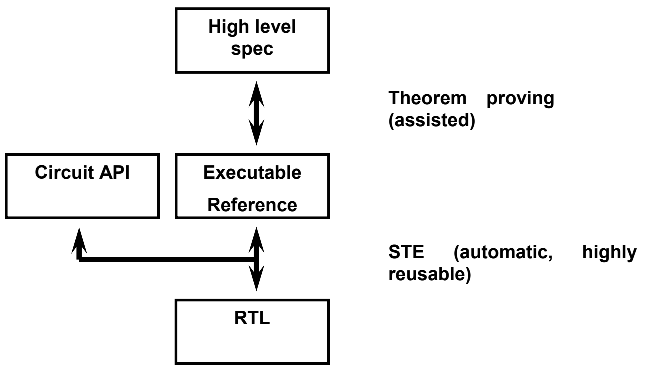

Figure 2. Intermediate reference model.

Floating point multiply and divide type operations required additional effort as the hardware may need to iterate to complete the operation. To complete the proofs we used a variant of traditional pre-postcondition framework for formulating temporal aspects of the specification [7]. Figure 3 below presents an abstract view of the multiplier hardware.

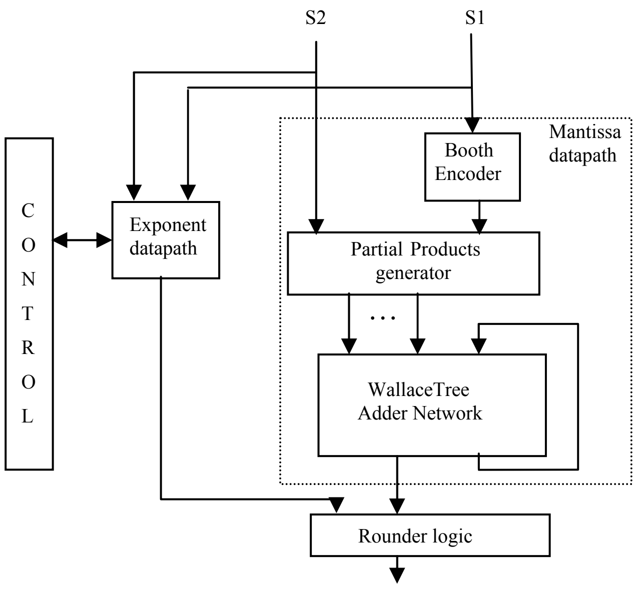

Figure 3. Floating point multiplier implementation

A sample multiplication verification statement we proved can be stated as:

IF a floating point multiplication operation is started AND *the inputs to the multiplier are S1 and S2,* AND *expected internal constraints to the circuit hold initially,* AND *expected environment constraints hold throughout the execution of the operation,*

THEN *at the time, the circuit produces output W, the following equation holds:*

$$\hat{W} = round(\hat{S1} * \hat{S2})$$

The circuit output W, encodes the result of the multiplication including flags and faults. To complete the proof, STE was used to prove properties on the low level RTL (steps 1 and 2) and theorem proving (steps 3 and 4) used to connect to the IEEE specification.

1. Verify the RTL satisfies a low level bit vector relation between each partial product and the input sources.
2. Verify that the expected bit-vector relation holds between the partial products and the rounded product.
3. Show that the bit vector relations in A and B imply the corresponding mathematical relations.
4. Prove that the mathematical relations between the input sources and the partial products and, the mathematical relation between the partial products and the rounded product imply the expected mathematical relation between the input sources and the rounded product.

The approach raises both the quality of our specifications and proofs. The framework (tools, methodology, and proofs) was constructed for the first Pentium® 4 processor, but designed to be very portable with minimal overhead, and will be the basis of future floating point proof work on proliferations. The team has successfully applied the framework to several other processor designs. Several unexpected, complex bugs have been quickly identified that might have otherwise have gone undetected during pre-silicon validation. Examples include a specific interaction between micro-operations on different threads resulting in data corruption and a dataspace multiply bug.

Machine-checked ('theorem-proved') proof compositions now require a far higher standard of proof than earlier methods. Specifications are at a higher level, corresponding more closely to an intuitive understanding of the operations and are more easily reviewed for correctness and completeness. Proof scope is extended to deal with issues previously ignored in datapath FV proofs (control behavior, dependencies between micro-operations).

2.2 Hierarchical Formal Verification Techniques

We now rely on STE in combination with theorem proving to verify floating point properties. We have also found STE to be very effective outside of floating-point logic, but it is limited to verification of finite length time interval properties. For the bulk of non floating properties, we used an LTL model checker that supported a richer set of temporal specifications [4]. Unfortunately, model checking tool limitations restrict verification to portions of logic with on the order of hundreds of input and state elements. Thus, to verify properties, both the properties and the implementation must be decomposed into a potentially great number of cases. [1] presents several examples from the Pentium® 4 processor, covering decomposition strategies used to verify a parallel ready queue, a floating-point adder, and memory arrays.

Our model checking tools combine the RTL logic with any assumptions on the logic and the conjectures. The tool automatically reduces the problem using *cone of influence* reduction---only the part of the design that may affect the conjecture is considered. Users also augment the list of assumptions with pruning directives that can further reduce the logic. By hand crafting a list of assumptions and “free” directives, a user can prove a set of simpler properties that when combined prove the more complex property. A property is then proven correct when the top-level specification, decomposition, and all of the assumptions have been proven.

During our verification effort, we found that many difficult to prove properties could be verified through clever decompositions. However, the decompositions could be fairly complicated. These decompositions were often fragile and required complex changes as the RTL evolved. Figure 4 presents example property decomposition and the dependencies between the sub-specifications. A property management system was also created to report the status of each assumption and assure that all specifications were proven.

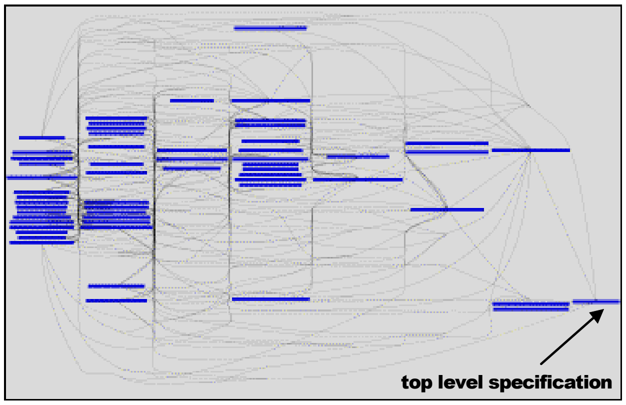

Figure 4. Property assumption proof chain.

The guaranteeing logic needed to prove a property frequently spans multiple logical units. Often only simple properties from other units needed to be verified independently. To minimize the need for complicated protocol proofs, we developed compilation tools that would combine multiple units together or extract only the needed guaranteeing logic from multiple units so that a property could be proved in a single proof session. There were a number of properties that required verification of complex, multiple unit protocols. These proofs assured the correctness of individual unit properties and also found high quality bugs. One example is a cross cluster 21 cycle protocol deadlock (infinite stall) due to resource dependencies. This bug was the result of complex resource dependencies coded by multiple designers. Designers tend to be segregated in their areas, and it is nearly impossible for them to consider all 21 cycle deep possibilities. As machines become wider and deeply pipelined, there is a greater likelihood that bugs of this nature will be inadvertently coded into the design and evade informal inspection.

3. FUTURE FPV APPLICATION

Validating the next-generation IA-32 microprocessor promises to be a significant challenge. We are pursuing a two pronged approach to enable us to realize our goal of improving the quality of CPU designs. Future CPU microarchitecture specifications will be likely written in a more abstract level. [2] outlines our belief that a more abstract level of micro-architectural specification (HLM) would help slow the rate of growth for bugs and to enable us to find bugs earlier in the design cycle An HLM will help verification handle much larger design areas than previously possible. With the development and early adoption of cutting edge FPV techniques, specifically the STE family of tools, the SAT based and the next generation LTL based model checking tools, and theorem proving capabilities, we anticipate contributing significantly to the validation solution.

During the Pentium® 4 project, we treated FPV and dynamic verification as essentially independent activities. Dynamic verification targeted bug-hunting and broad coverage objectives while formal verification targeted total correctness of critical functionality. Though there were properties that only FV could cover entirely, there was a fair amount of overlap between the two efforts. This was deemed to be appropriate as DV remains a more efficient means to identify the “obvious” bugs.

For the next-generation project, we want to exploit synergy between the two techniques. We are planning to develop a more integrated dynamic verification and FPV plan where we reduce overlap and apply the most appropriate technique to validate a given functionality.

To improve overall productivity, we plan to exploit tool interoperability so that FV can effectively use DV simulation and test databases to refine and debug specifications. We are also investigating the use of SAT solver technology as a bridge between the two worlds. Initial results are very encouraging. SAT solvers provide much greater capacity, reducing or eliminating the need for problem decomposition. They allow us to do bug hunting (falsification) in addition to verification, and use dynamic validation to confirm or refute counter-examples.

3.1 Improved Formal Verification Capacity

The development of Generalized STE (GSTE) has provided an automated property verification tool with extremely high capacity that significantly extends STE to verify a richer set of properties [10],[11],[12]. STE properties are (antecedent, consequent) pairs, where the antecedent describes a stimulus to the circuit and the consequent specifies the expected response. STE is capable of verifying very large circuits (indeed, much larger circuits than traditional symbolic model checkers), but at a cost: antecedents and consequents can only describe behavior over finite time intervals, therefore STE properties correspond to a weak flavor of temporal logic. Nevertheless, STE has seen extensive use for property and equivalence verification in Intel, IBM and Motorola.

Generalized STE (GSTE) extends STE to verify properties over indefinitely long time intervals, while preserving STE’s high capacity. GSTE properties are assertion graphs, with each edge in the graph labeled with an STE (antecedent, consequent) pair. Also associated with edges are terminal or fairness conditions, enabling assertion graphs to capture extremely complex temporal properties. GSTE has been used successfully in a number of recent verifications that will be presented in the next section.

4. GSTE EXAMPLES

Our early success with GSTE has prompted us to deploy the technology into production use prior to its complete development. Consequently, valuable feedback has guided GSTE development and for realistic examples, we have enhanced the performance of GSTE to require significantly less memory and time than its initial release. This early use has also greatly benefited the Pentium® 4 Formal Verification programs where we have verified significantly more complex properties than previously possible. In this section, we will present three example properties proven with GSTE that cover logic from a Scheduler/Scoreboard Unit, an Instruction Queue unit, and a Memory Execution Unit.

4.1 Scheduler/Scoreboard Unit Example

As an early proof of concept, we looked at a previously very challenging Scheduler/Scoreboard unit (SSU) property that had been quite difficult to prove. The logic involved (see Figure 5) created significant tool capacity problems, requiring that the original proof be decomposed into hundreds of sub-specifications. Creating and proving this decomposition required a significant amount of time and its maintenance as the design changed has been costly. The goal of the new proof work was to show that a 10x10 priority matrix would schedule the oldest microinstruction ("uop") that was ready to be scheduled. This actually expanded on the original proof that was only able to show a weaker condition that if there are uops ready to be scheduled, then some uop, not necessarily the oldest, would be scheduled. Previous generation tool limitations also restricted the original property to specify behavior with respect to the 10x10 priority matrix rather than at the more abstracted (unit level) interface.

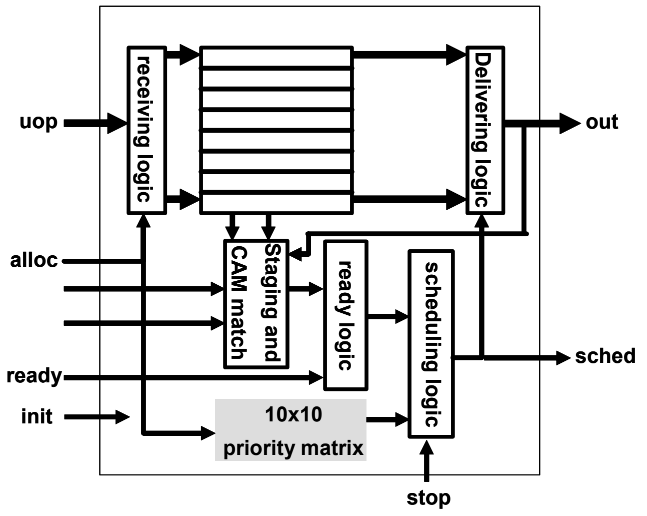

Figure 5. Pentium® 4 SSU Scheduler

Using GSTE, the broader property was proved on the complete design presented in Figure 5. The resulting proof has been much more maintainable and scalable to other Pentium® 4 proliferation products.

4.2 Instruction Queue Unit Example

The second example covers a new cross-unit property on FIFO buffers between the instruction queue unit (IQ), which holds and dispatches microinstructions to the Execution Cluster, and the Scheduler/Scoreboard unit (SSU), which schedules them (see Figure 6). The property we wanted to prove states that uops are neither dropped nor duplicated.

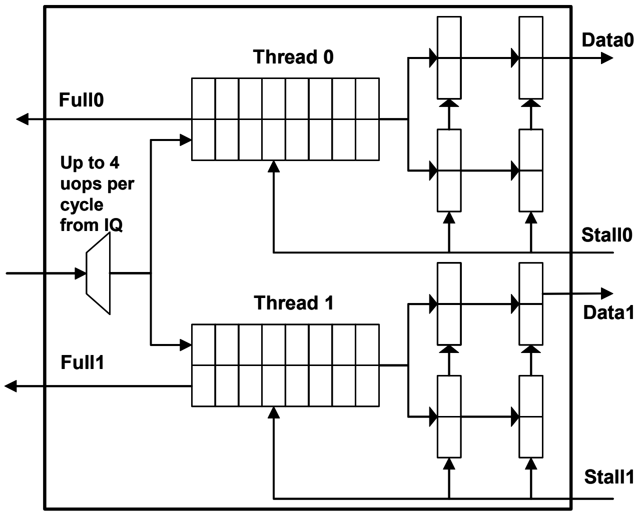

Figure 6. Pentium® 4 Instruction Queue

The logic is rather complex as it is highly optimized for performance and as a result, a proof was well beyond the scope of previous approaches and tools. Using GSTE, a proof was completed and has been maintained with minimal effort. Also, like the SSU scheduler property, we have been able express the property abstractly in terms of the input and output behavior of the FIFO.

4.3 Memory Execution Unit Example

The third example covers a property on the memory execution unit showing that uops are cleared out of a set of registers when their execution should be halted. The logic involved is again, very complex involving a 20 stage pipe and complicated control logic with feedback (see Figure 7).

This property was originally completed with previous generation FV tools, but the proof had required significant decomposition and approximately 12 person-quarters to complete (excluding ramp up on the architecture and tools). The increased complexity of the logic on proliferations made porting the proofs impractical, so a GSTE approach was taken. A port to a first proliferation required only 5 person-quarters and a second proliferation (with yet more logic changes), only 1.5 person-quarters.

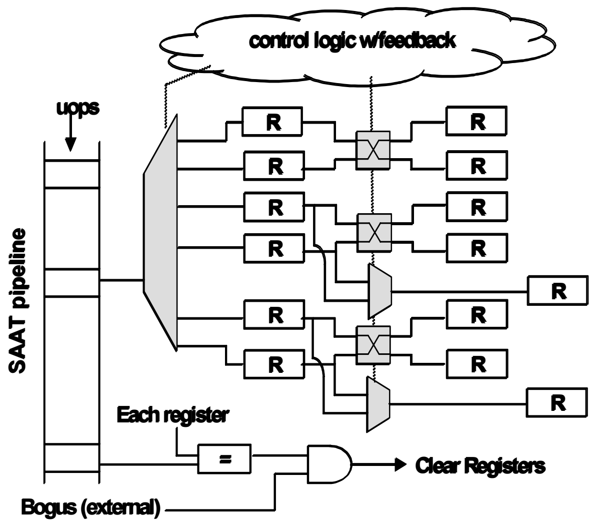

Figure 7. Pentium® 4 Memory Unit

5. CONCLUSIONS

Formal Property Verification has been an effective component in the Pentium® 4 validation effort. Our objectives have been to prove the total correctness of critical properties, maintain these properties during the course of development project, and reuse specifications and proofs on proliferations.

Where FPV tools were unable to consume all the guaranteeing logic in one verification run, properties were decomposed to reduce the amount of design required to prove each sub-property. The resulting list of assumptions was then verified separately. In many instances, the guaranteeing logic involved spanned multiple design components and multiple logic unit protocols were verified.

We have found the greatest successes where proof strategies have been developed that are amenable to change and minimize the effort required on new architectures. Use of abstraction techniques, such as separating data/control issues and building intermediate models has required significant up-front investment, but has been seamlessly reused by other FPV teams at Intel.

We anticipate using a mix of approaches to speed up productivity and have been very encouraged by our experiences with our tool suite of FPV techniques. In addition to the tools available to us, methodology will play a key role in furthering the impact of formal methods in future CPU projects. We are currently engaged in constructing a cohesive framework for the existing tools and anticipate they will provide a dramatic improvement on what we have accomplished thus far.

6. ACKNOWLEDGEMENTS

The work described in this paper was completed by a large FPV team over the past 5 years---too many to name here. I would particularly like to thank Naren Narasimhan, Rajnish Ghughal, Bob Bentley, John O'Leary, Eli Singerman, and Jin Yang and for their valuable feedback and assistance in writing this paper.

7. REFERENCES

[1] R. Beers, R. Ghughal, and M. Aagaard, "Applications of Hierarchical Verification in Model Checking." FMCAD 2000 (published in CHARM 2001 proceedings).

[2] B. Bentley, "High level validation of next-generation microprocessors", IEEE International Workshop on High Level Design Validation and Test, 2002.

[3] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi, "Benefits of Bounded Model Checking in an Industrial Setting", International Conference on Computer-Aided Verification (CAV), 2001.

[4] R.Fraer, G.Kamhi, B.Ziv, M.Vardi, L.Fix, "Efficient Reachability Computation Both for Verification and Falsification", Proceedings of International Conference on Computer-Aided Design, (CAV'00).

[5] S. Hazelhurst and C-J Seger, "Symbolic trajectory evaluation." In T. Kropf, editor, Formal Hardware Verification, chapter 1, pagers 3-78. Springer Verlag; New York, 1997.

[6] R. Kaivola and K. Kohatsu, "Proof Engineering in the Large: Formal Verification of Pentium®4 FP Divider", CHARM 2001.

[7] N. Narasimhan and R. Kaivola, "Verification of Pentium®4 Multiplier with Symbolic Simulation & Theorem Proving", DATE 2001.

[8] J. O'Leary, S. Zhao, R. Gerth, and C.-J. H. Seger, "Formally Verifying IEEE Compliance of Floating-Point Hardware", Intel Technology Journal, Q1 1999.

[9] C.-J. H. Seger and R. E. Bryant, "Symbolic trajectory evaluation", Formal Methods in System Design, 1994.

[10] J. Yang and A. Goel, "GSTE through a case study", International Conference on Computer-Aided Design ICCAD, 2002.

[11] J. Yang and C.-J. H. Seger, "Introduction to Generalized Symbolic Trajectory Evaluation", International Conference on Computer Design (ICCD), 2001. A revised version will appear in IEEE Transactions on VLSI.

[12] J. Yang and C.-J. H. Seger, "Generalized Symbolic Trajectory Evaluation: Abstraction in action", Fourth International Symposium on Formal Methods in Computer-Aided Design (FMCAD), 2002.