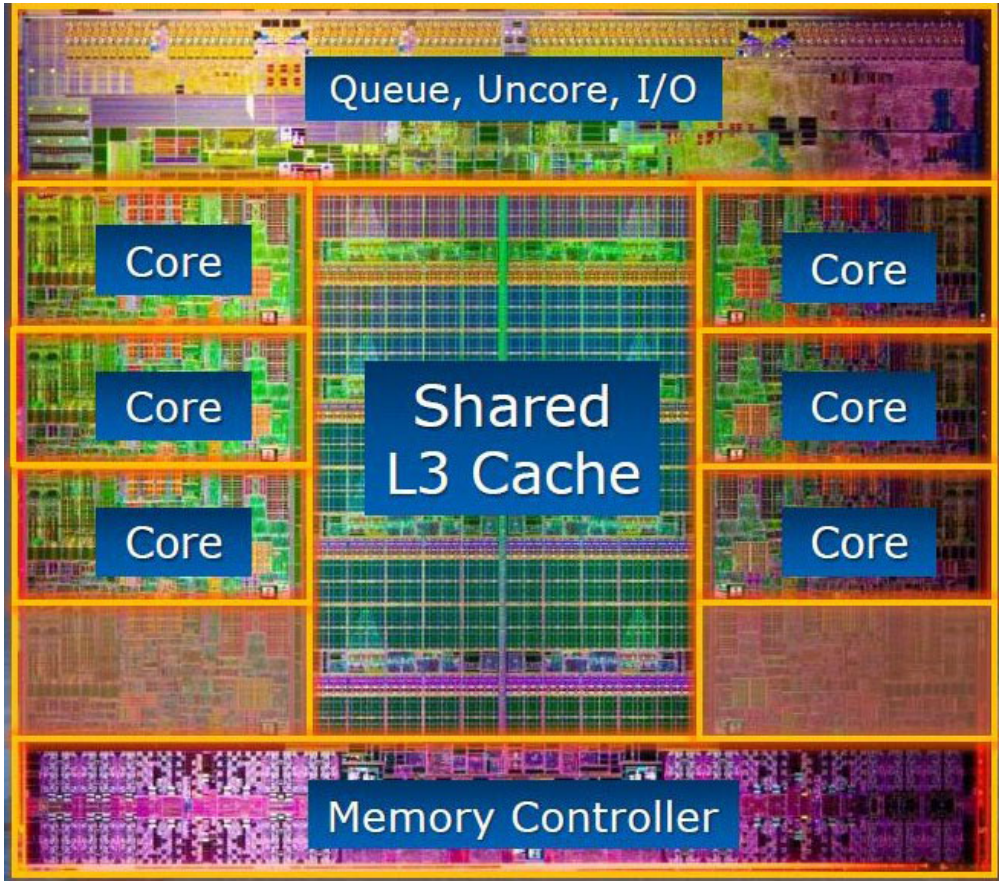
At-scale Formal Verification for Industrial Semiconductor Designs

Tom Melham

University of Oxford

A Modern Microprocessor



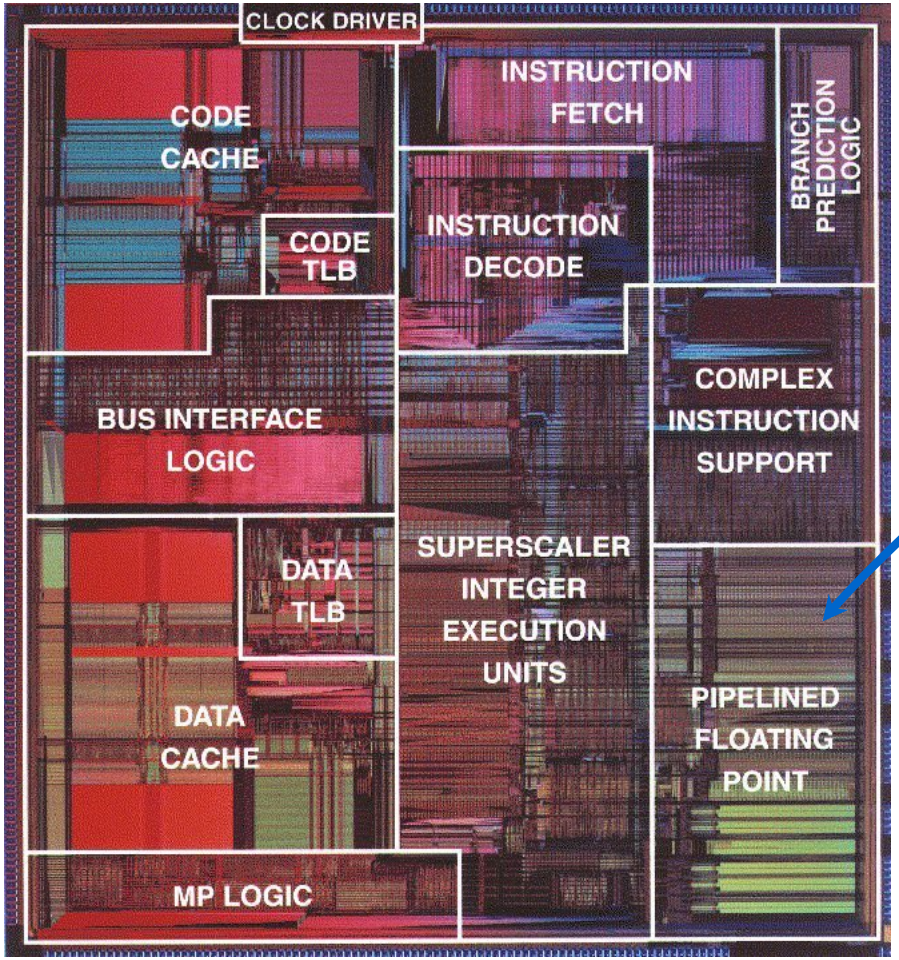
Intel Core i7-3930K

- 6 processor ‘cores’
- 2.27 billion transistors
- ≈ 2cm × 2cm

- 32nm process
- 3.20 GHz
- 12MB cache

One of the most complex artefacts ever made by humans…

And Hard to Get Right



Intel's famous FDIV bug

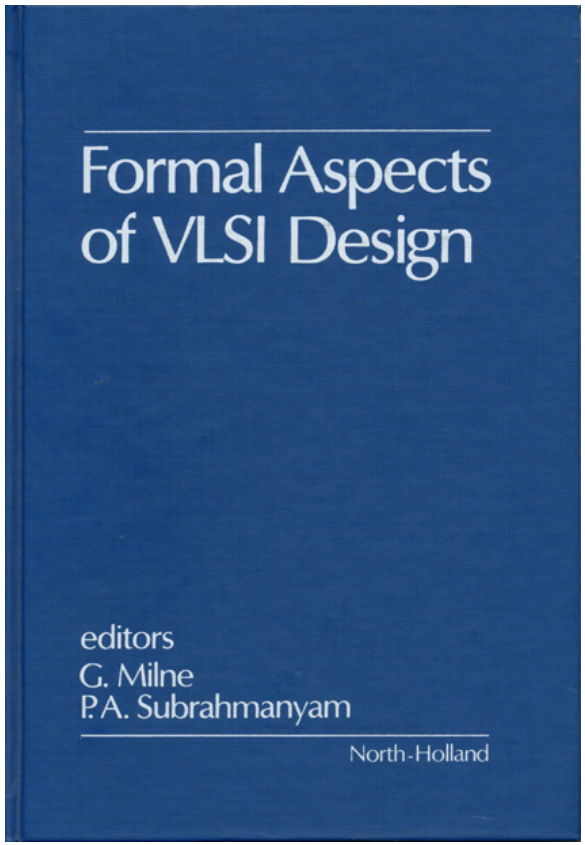

```
4195835.0 / 3145727.0

= 1.333 820 449 136 241 002 (Correct)
= 1.333 739 068 902 037 589 (Flawed)
```

Huge potential cost (100s of M$)

Back in 1985…

Formal verification seemed a vain hope to all but a few bold pioneers



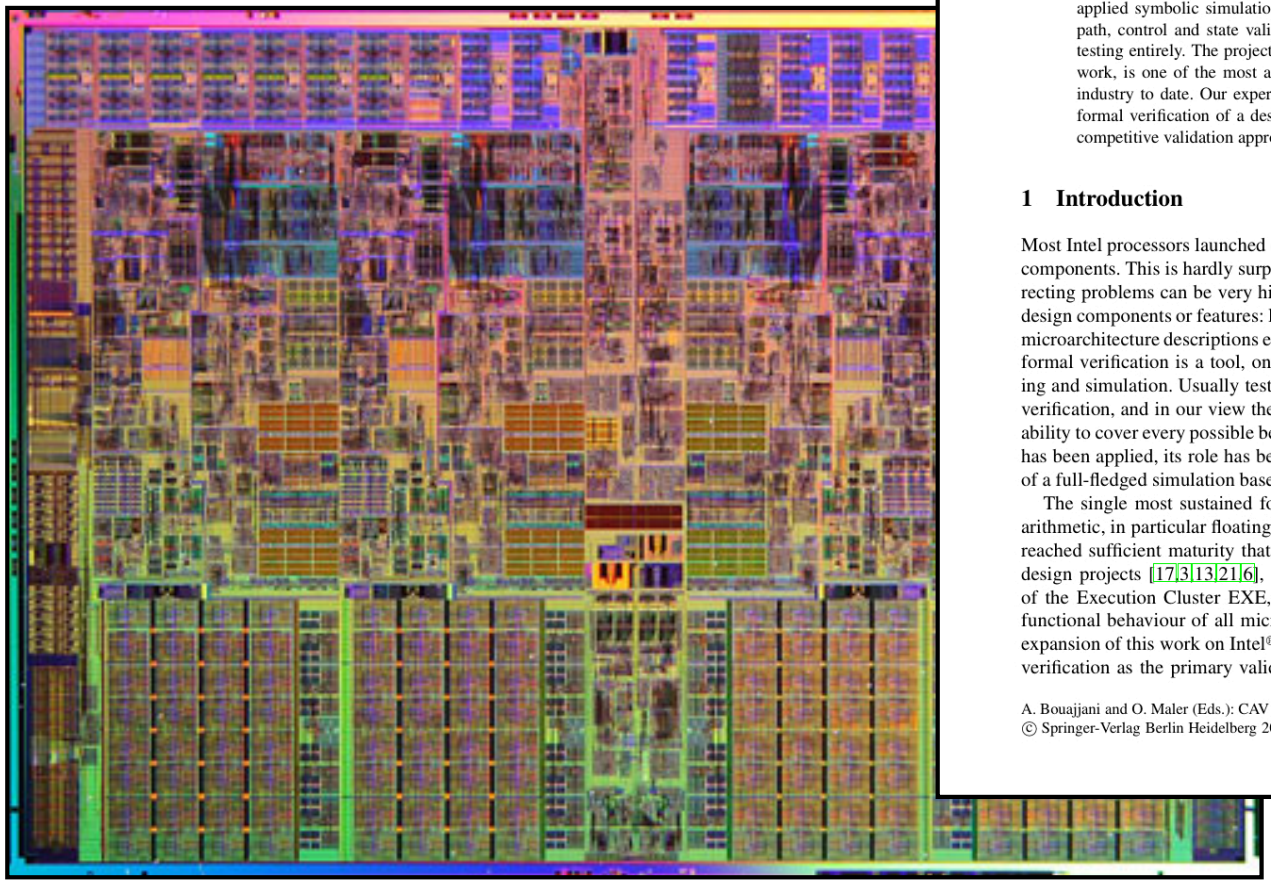
Why Higher-Order Logic is a good Formalism for Specifying and Verifying Hardware

Can a Simulator Verify a Circuit?

Automatic Circuit Verification Using Temporal Logic: Two New Examples

Advance to 2009

Full data-path, control, and state validation of the *entire i7 EXE cluster* **- Intel**



Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation

Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik

Intel Corporation, JF4-451, 2111 NE 25th Avenue, Hillsboro, OR 97124, USA

Abstract. Formal verification of arithmetic datapaths has been part of the established methodology for most Intel processor designs over the last years, usually in the role of supplementing more traditional coverage oriented testing activities. For the recent Intel® Core™ i7 design we took a step further and used formal verification as the primary validation vehicle for the core execution cluster, the component responsible for the functional behaviour of all microinstructions. We applied symbolic simulation based formal verification techniques for full datapath, control and state validation for the cluster, and dropped coverage driven testing entirely. The project, involving some twenty person years of verification work, is one of the most ambitious formal verification efforts in the hardware industry to date. Our experiences show that under the right circumstances, full formal verification of a design component is a feasible, industrially viable and competitive validation approach.

1 Introduction

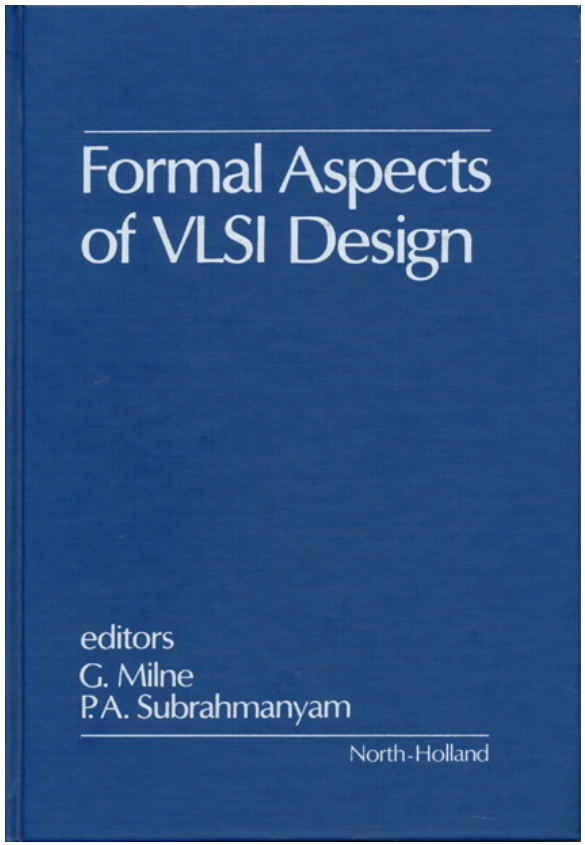
Most Intel processors launched over the last ten years have contained formally verified components. This is hardly surprising, as their reliability is crucial, and the cost of correcting problems can be very high. Formal verification has been applied to a range of design components or features: low-level protocols, register renaming, arithmetic units, microarchitecture descriptions etc. [19,4]. In an industrial product development setting, formal verification is a tool, one among others, and it competes with traditional testing and simulation. Usually testing can produce initial results much faster than formal verification, and in our view the value of formal verification primarily comes from its ability to cover every possible behaviour. In most of the cases where formal verification has been applied, its role has been that of a supplementary verification method on top of a full-fledged simulation based dynamic validation effort.

The single most sustained formal verification effort has been made in the area of arithmetic, in particular floating point datapaths. In this area verification methods have reached sufficient maturity that they have now been routinely applied for a series of design projects [17,3,13,21,6], and expanded to cover the full datapath functionality of the Execution Cluster EXE, a top-level component of a core responsible for the functional behaviour of all microinstructions. In the current paper we discuss further expansion of this work on Intel® Core™ i7 design [1]. For this project, we used formal verification as the primary validation vehicle for the execution cluster, including full

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 414–429, 2009.

Back to 1985

"Logic simulators seem like a good starting point..."



Formal Aspects of VLSI Design
G.J. Milne and P.A. Subrahmanyam (editors)

Can a Simulator Verify a Circuit?

Randal E. Bryant
Dept. of Computer Science
Carnegie-Mellon University

A logic simulator can prove the correctness of a digital circuit if only circuits implementing the system specification can produce a particular response to a sequence of simulation commands. This paper explores two methods for verifying circuits by a three-valued logic simulator where the third state X indicates an indeterminate value. The first, called "black-box" simulation, involves simply observing the output produced by the simulator in response to a sequence of inputs with no consideration of the internal circuit structure. This style of simulation can verify only a limited class of systems. The second method, called "transition" simulation, requires the user to specify the relation between states in the circuit and the specification. The simulator is then used to prove that each state transition in the specification is implemented correctly. Arbitrary systems may be verified by this method.

Introduction

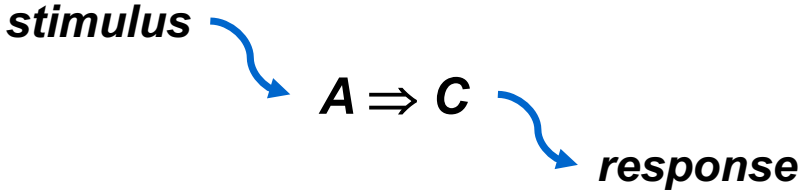
Digital designers rely primarily on logic simulators to test the correctness of their logic circuits. Typically, however, only a limited set of test cases are simulated, and the circuit is presumed correct if the simulator yields the expected results for all of these cases. This style of simulation is analogous to the unstructured way most programmers test programs. To paraphrase a saying of Dijkstra, unstructured use of a simulator can detect the presence of errors but can never prove their total absence. Stories abound of circuit design errors that remain undetected despite many hours of simulation time. Given the great effort required to design and manufacture a system of VLSI complexity, the cost of undetected design errors can be quite high. For this reason more reliable methods of circuit verification are desired.

Consider the possibility of rigorously verifying a circuit by logic simulation. If it could be shown that only circuits meeting the specification will produce satisfactory results for a given set of test cases, then a simulator could serve as a fully automated verification system. Since logic simulators have been developed for a wide range of circuit models and digital design styles, such an approach would not require extensive development of a new class of programs as would other verification methods. Logic simulators seem like a good starting point for developing a circuit verification program. By investigating their expressive power, we may better understand what additional capabilities are required. Thus, we are motivated to ask: "Under what conditions does a simulation constitute a proof of the circuit's correctness?"

Before answering this question, we must formulate more precise definitions of the system specification, the circuit, and the simulator. We do this in a somewhat abstract way such that the results apply to a wide variety of circuit technologies and simulators. The desired behavior is specified in terms of a finite state automaton. A circuit is also described as a finite state automaton, but with a particular binary encoding of the states. The circuit is said to implement the specification if the two automata are input-output equivalent. The simulator models the behavior of the circuit automaton, computing new state and output values in

Symbolic Trajectory Evaluation

Logic of *Simulation Properties*



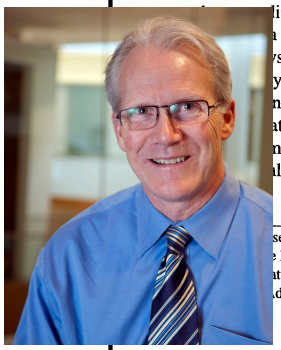
f := node is 0

| node is 1

| $f_1 \wedge f_2$

| Next f

| E → f



Formal Methods in System Design, 6, 147–1

Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories*

CARL-JOHAN H. SEGER
Department of Computer Science, University of British Columbia, Vancouver, B.C. V6T 1Z4 Canada

RANDAL E. BRYANT
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA

Abstract. Symbolic trajectory evaluation provides a means to formally verify properties of a sequential system by a modified form of symbolic simulation. The desired system properties are expressed in a notation combining Boolean expressions and the temporal logic "next-time" operator. In its simplest form, each property is expressed as an assertion $[A \Rightarrow C]$, where the antecedent A expresses some assumed conditions on the system state over a bounded time period, and the consequent C expresses conditions that should result. A generalization allows simple invariants to be established and proven automatically.

The verifier operates on system models in which the state space is ordered by "information content". By suitable restrictions to the specification notation, we guarantee that for every trajectory formula, there is a unique weakest state trajectory that satisfies it. Therefore, we can verify an assertion $[A \Rightarrow C]$ by simulating the system over the weakest trajectory for A and testing adherence to C. Also, establishing invariants correspond to simple fixed point calculations.

This paper presents the general theory underlying symbolic trajectory evaluation. It also illustrates the application of the theory to the task of verifying switch-level circuits as well as more abstract implementations.

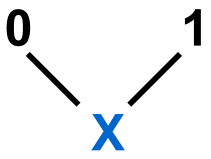
1. Introduction

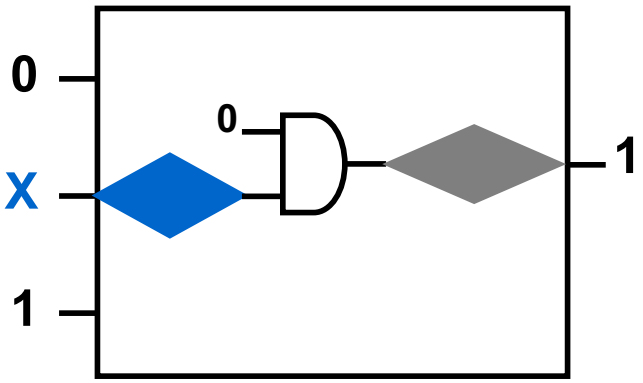
Verifying a digital system by conventional simulation is feasible only for very small systems, since the large number of possible initial states and input sequences would require massive amounts of case a nalysis. By exploiting a combination of *abstraction* and *symbolic manipulation*, on the other hand, symbolic trajectory evaluation can verify the behavior of complex systems by a modified form of simulation. This method exploits abstraction
ling the system state space to include elements representing sets of actual states,
a partially-ordered system model. A single simulation sequence can then verify
stem would produce a unique result for a set of initial states or input sequences. It
ymbolic manipulation by a modified form of symbolic simulation. The Boolean
ns appearing in the system specification are converted into symbolic patterns for
ator. Like a conventional simulation, a single run of the trajectory evaluator models
n behavior over a single state sequence, although this sequence is both symbolic
lly-ordered.

search was supported by the Defense Advanced Research Projects Agency, ARPA Order Number
National Science Foundation, under grant number MIP-8913667, by operating grant OGPO 109688
tural Sciences and Engineering Research Council of Canada, and by a fellowship from the British
dvanced Systems Institute.

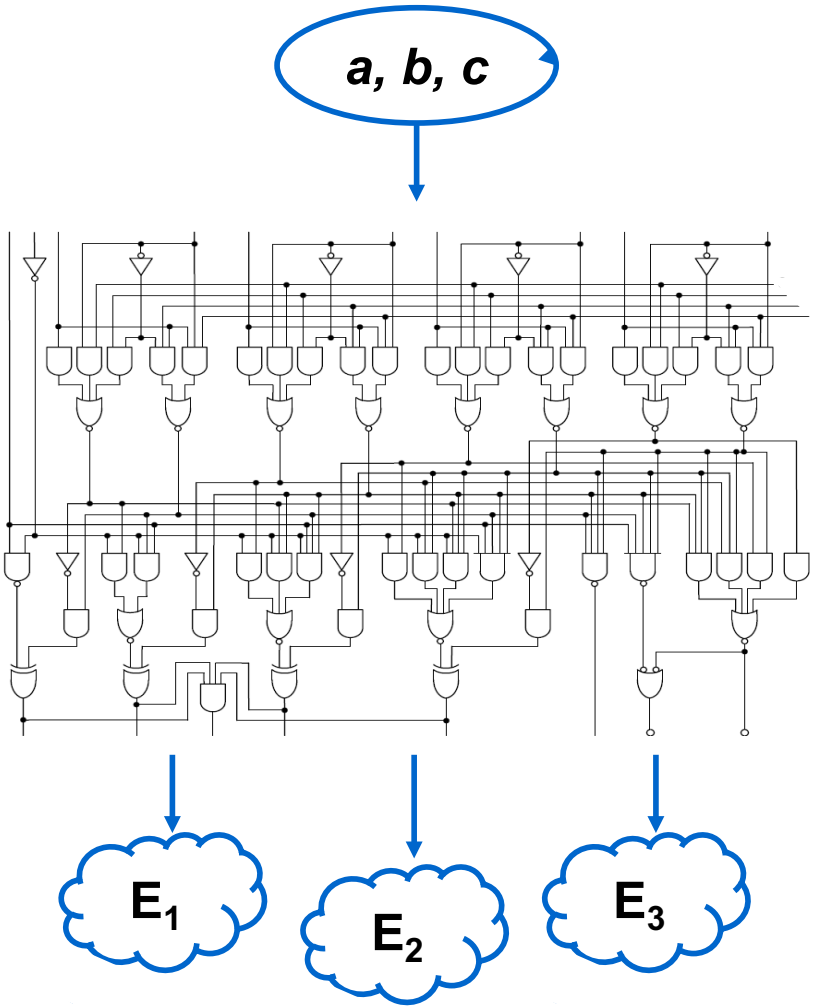
Combination of Two Good Ideas

Ternary Simulation





Symbolic Simulation



State Space *Abstraction*

Abstraction = simplification, discarding information

Discarding information about states

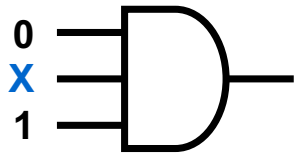
more Xs

$$\frac{A' \Rightarrow C}{A \Rightarrow C}$$

[*technically:* $\delta(A') \leq \delta(A)$]

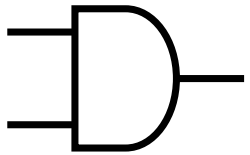
In STE, abstraction is driven by the specification

a is 0 ∧ c is 1 ⇒ out is 0

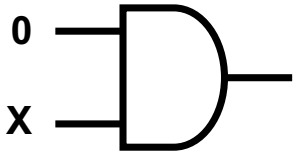


Guard Expressions $E \rightarrow f$

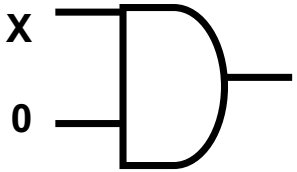
A symbolic layer *on top of* properties & abstractions



$v \rightarrow$ a is 0 $\wedge$ $\overline{v} \rightarrow$ b is 0 $\Rightarrow$ out is 0



a is 0 $\Rightarrow$ out is 0



b is 0 $\Rightarrow$ out is 0

Result - *Partitioned Abstraction*

Task – verify this circuit

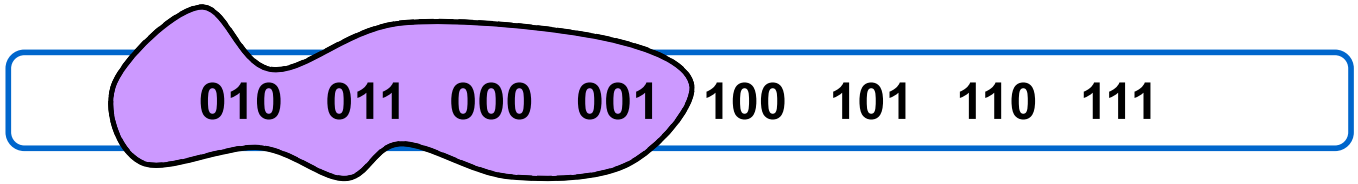
Idea – test all 8 possible input patterns

010 011 000 001 100 101 110 111

Result - *Partitioned Abstraction*

Task – verify this circuit

Idea – test all 8 possible input patterns

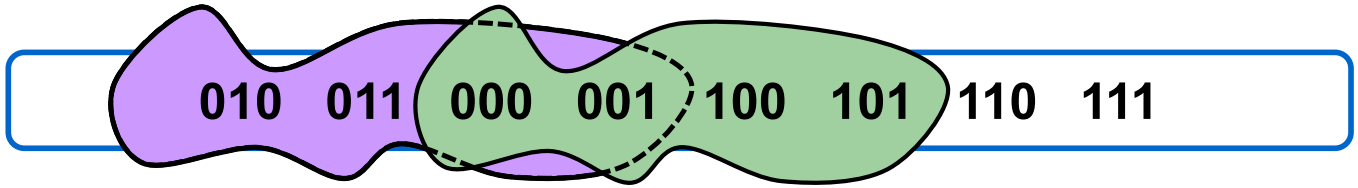


Better idea – exploit *don't cares*

Result - *Partitioned Abstraction*

Task – verify this circuit

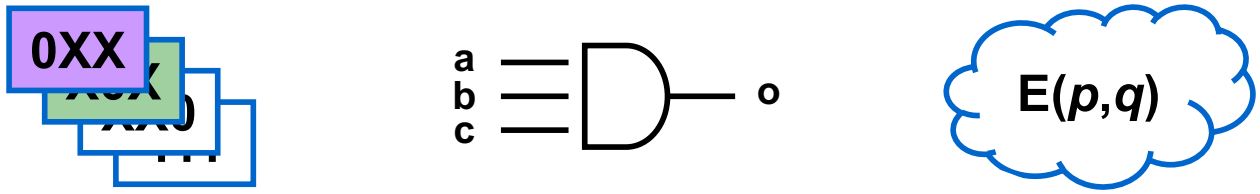
Idea – test all 8 possible input patterns



Better idea – exploit *don't cares*

Key Technical Idea - Exploit Symmetry

Combine all four abstractions symbolically



$$A(p,q) \Rightarrow C(p,q)$$

Big win when the representation has lots of sharing.

Excels at complex data path verification.

Intel's *Forte* Tool

Major successes

Full Core i7 EXE unit

Core 2 duo EXE + MS units

Itanium FP fused multiply/add

IA32 Instruction Length Decoder

and many others…

A programming environment:

- **simulation property logic**
- **symbolic simulation**
- **abstraction**
- **SAT and BDDs**
- **functional scripting**
- **…**

IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 24, NO. 9, SEPTEMBER 2005 1381

An Industrially Effective Environment for Formal Hardware Verification

Carl-Johan H. Seger, Robert B. Jones, *Member, IEEE*, John W. O'Leary, *Member, IEEE*, Tom Melham, Mark D. Aagaard, *Member, IEEE*, Clark Barrett, and Don Syme

***Abstract*—The Forte formal verification environment for datapath-dominated hardware is described. Forte has proven to be effective in large-scale industrial trials and combines an efficient linear-time logic model-checking algorithm, namely the symbolic trajectory evaluation (STE), with lightweight theorem proving in higher-order logic. These are tightly integrated in a general-purpose functional programming language, which both allows the system to be easily customized and at the same time serves as a specification language. The design philosophy behind Forte is presented and the elements of the verification methodology that make it effective in practice are also described.**

***Index Terms*—BDDs, formal verification, model checking, symbolic trajectory evaluation, theorem proving.**

I. Introduction

FUNCTIONAL validation is one of the major challenges in chip design today, with conventional approaches to design validation a serious bottleneck in the design flow. Over the past ten years, formal verification [1] has emerged as a complement to simulation and has delivered promising results in trials on industrial-scale designs [2]–[6].

Formal equivalence checking is widely deployed to compare the behavior of two models of hardware, each represented as a finite state machine or simply a Boolean expression (often using binary decision diagrams (BDDs) [7]). It is typically used in industry to validate the output of a synthesis tool against a "golden model" expressed in a register-transfer level hardware description language (HDL), and in general to check consistency between other adjacent levels in the design flow.

Property checking with a model checker [8]–[11] also involves representing a design as a finite state machine, but it has wider capabilities than equivalence checking. Not only can one check that a design behaves the same as another model, one can also check that the hardware possesses certain desirable properties expressed more abstractly in a temporal logic. An example is checking that all requests are eventually acknowledged in a protocol. Model checking is currently much less widely used in practice than equivalence checking.

Theorem proving [12], [13] allows higher level and more abstract properties to be checked. It provides a much more expressive language for stating properties—for example, higher order logic [14]—and it can deal with infinite-state systems. In particular, it allows one to reason with unknowns and parameters, so a general class of designs can be checked—for example, parameterized IP blocks [15]. Industrially, theorem proving is still viewed as a very advanced technology, and its use is not widespread.

Equivalence checkers and model checkers both suffer from severe capacity limits. In practice, only small fragments of systems can be handled directly with these technologies, and much current research is aimed at extending capacity. Of course, it is unrealistic to expect a completely automatic model-checking solution. Instead, one needs to find good ways of using human intelligence to extract the maximum potential from model-checking algorithms and to decompose problems into appropriate pieces for automated analysis. One approach is to combine model-checking and BDD-based methods with theorem proving [16]–[18]. The hope is that theorem proving's power and flexibility will enable large problems to be broken down or transformed into tasks a model checker finds tractable. Another approach is to extend the top level of a model checker with ad hoc theorem-proving rules and procedures [19].

This paper describes a formal verification system called Forte that combines an efficient linear-time logic model checking algorithm, namely symbolic trajectory evaluation (STE) [20], with lightweight theorem proving in higher-order logic. These are interfaced to and tightly integrated with FL [21], a strongly typed, higher order functional programming language. As a general-purpose programming language, FL allows the Forte environment to be customized and large proof efforts to be organized and scripted effectively. FL also serves as an expressive specification language at a level far above the temporal logic primitives.

The Forte environment has proven to be highly effective in large-scale industrial trials on datapath-dominated hardware [3], [22], [23]. The restricted temporal logic of STE does not, however, limit Forte to pure datapath circuits. Many large control circuits are "datapath-as-control," and these can also be handled effectively. In addition, the tight connection to higher-order logic and theorem proving provides great flexibility in

anuary 20, 2004; revised June 23, 2004. This paper ociate Editor J. H. Kukula.

Jones, and J. W. O'Leary are with Strategic CAD Hillsboro, OR 97124 USA (e-mail: Carl.Seger@ s@intel.com; John.W.O'Leary@intel.com).
Oxford University Computing Laboratory, Oxford Tom.Melham@comlab.ox.ac.uk).
h the Department of Electrical and Computer En- Waterloo, Wateroo, ON N2L 3G1, Canada (e-mail:
Department of Computer Science, Courant Institute s, New York University, New York, NY 10012 USA du).
osoft Research, Cambridge CB3 0FB, U.K. (e-mail:

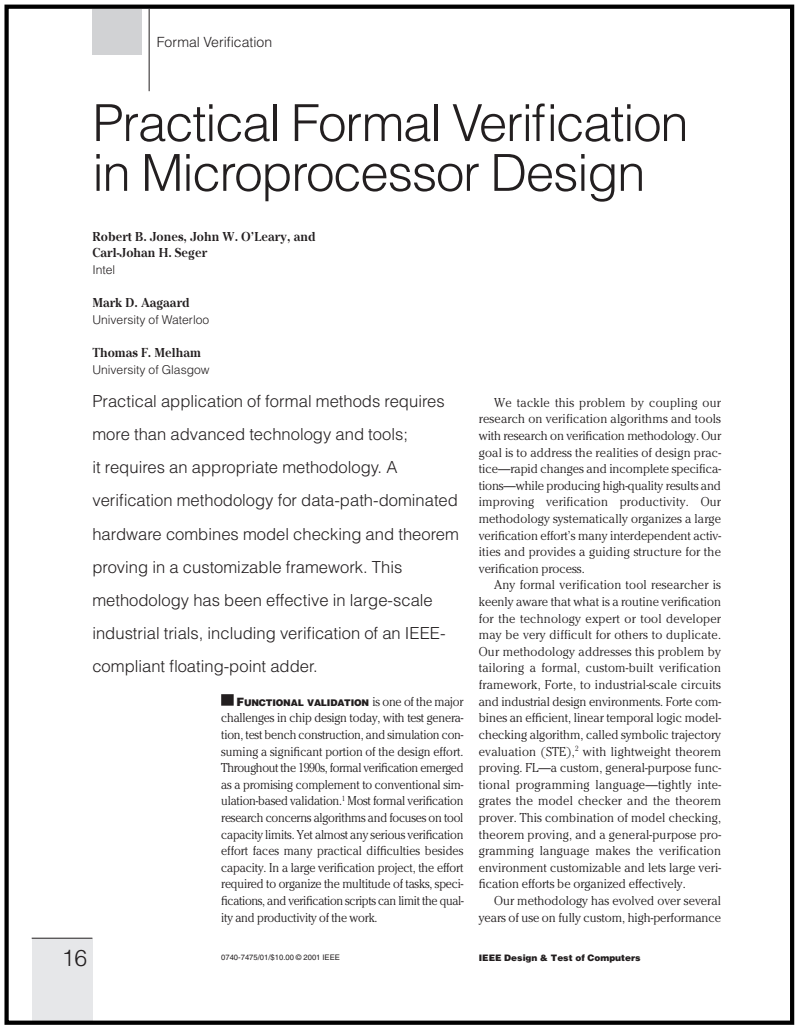
r 10.1109/TCAD.2005.850814

Forte Methodology

“A systematic, pragmatic approach to organizing large-scale design verification efforts.”

Explicit principles:

- **Sound and Transparent**
- **Structured**
- **Realistic**
- **Incremental**
- **Provides good feedback**
- **Top-down and bottom up**
- **Supports regression**
- **Allows effort reuse**



Formal Verification

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

16

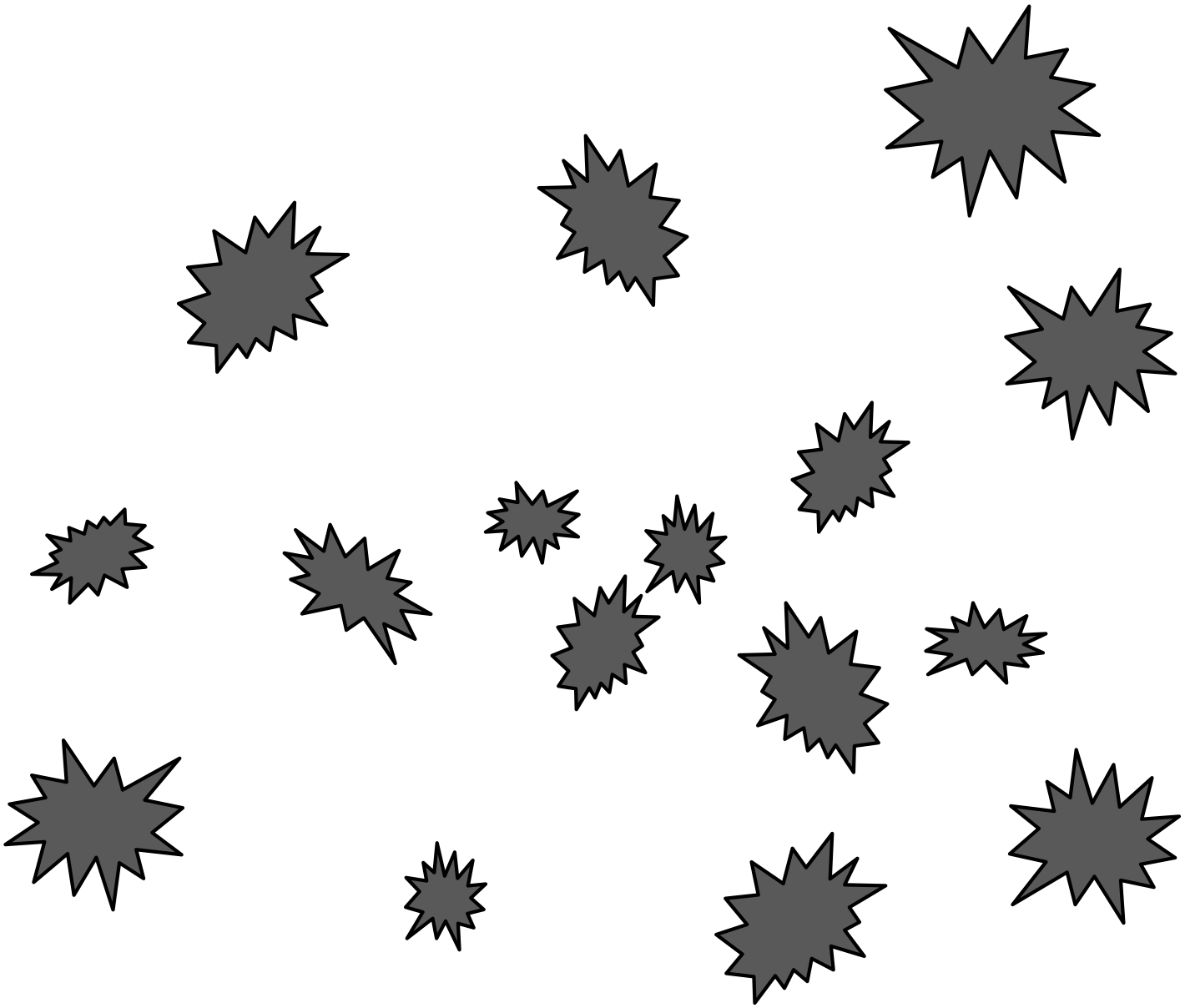
0740-7475/01/$10.00 © 2001 IEEE

Key Drivers of Progress

1. Algorithms
2. Abstractions
3. Methodology
4. Moore’s Law

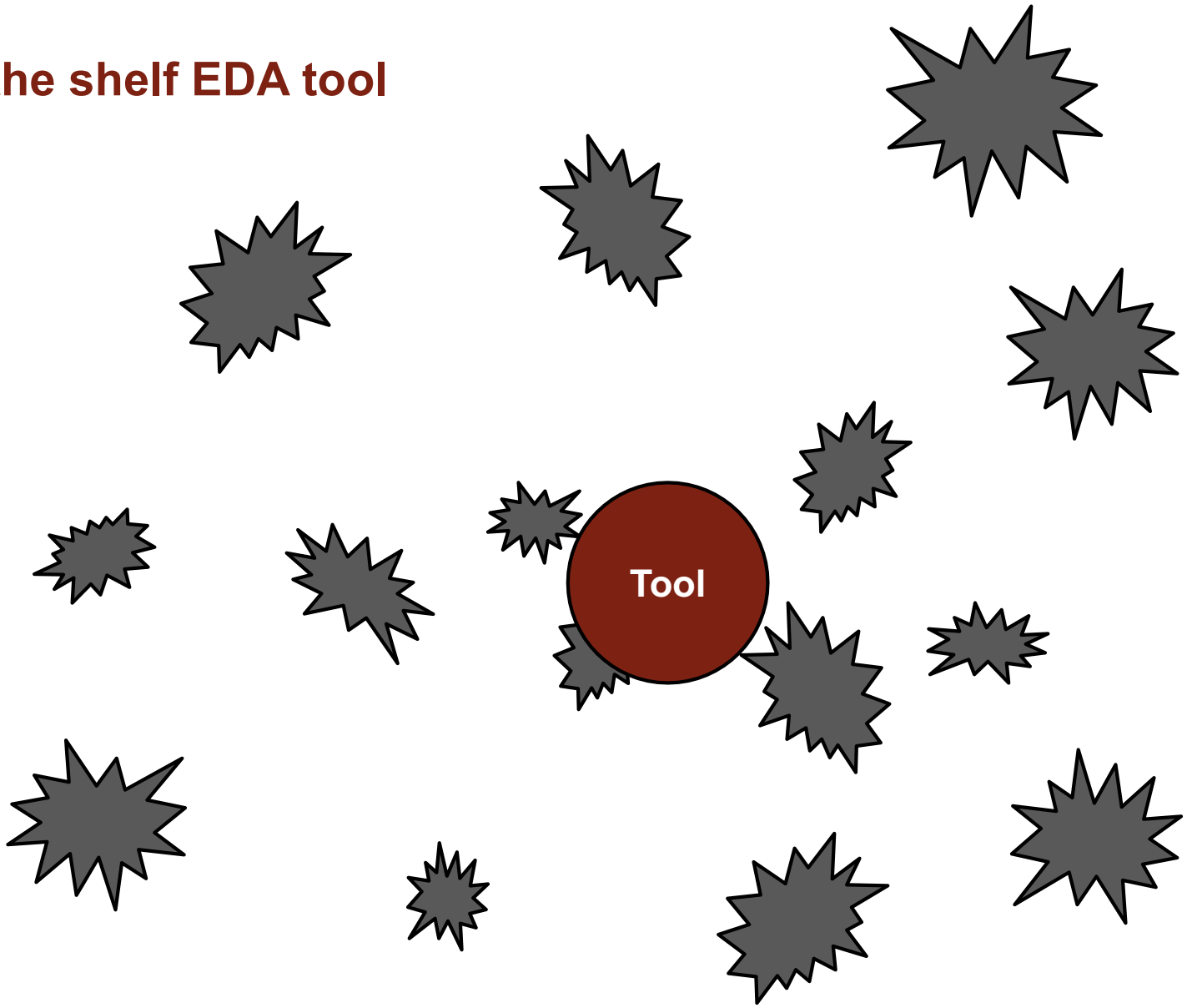
Formal Verification in Practice

Problems, problems…



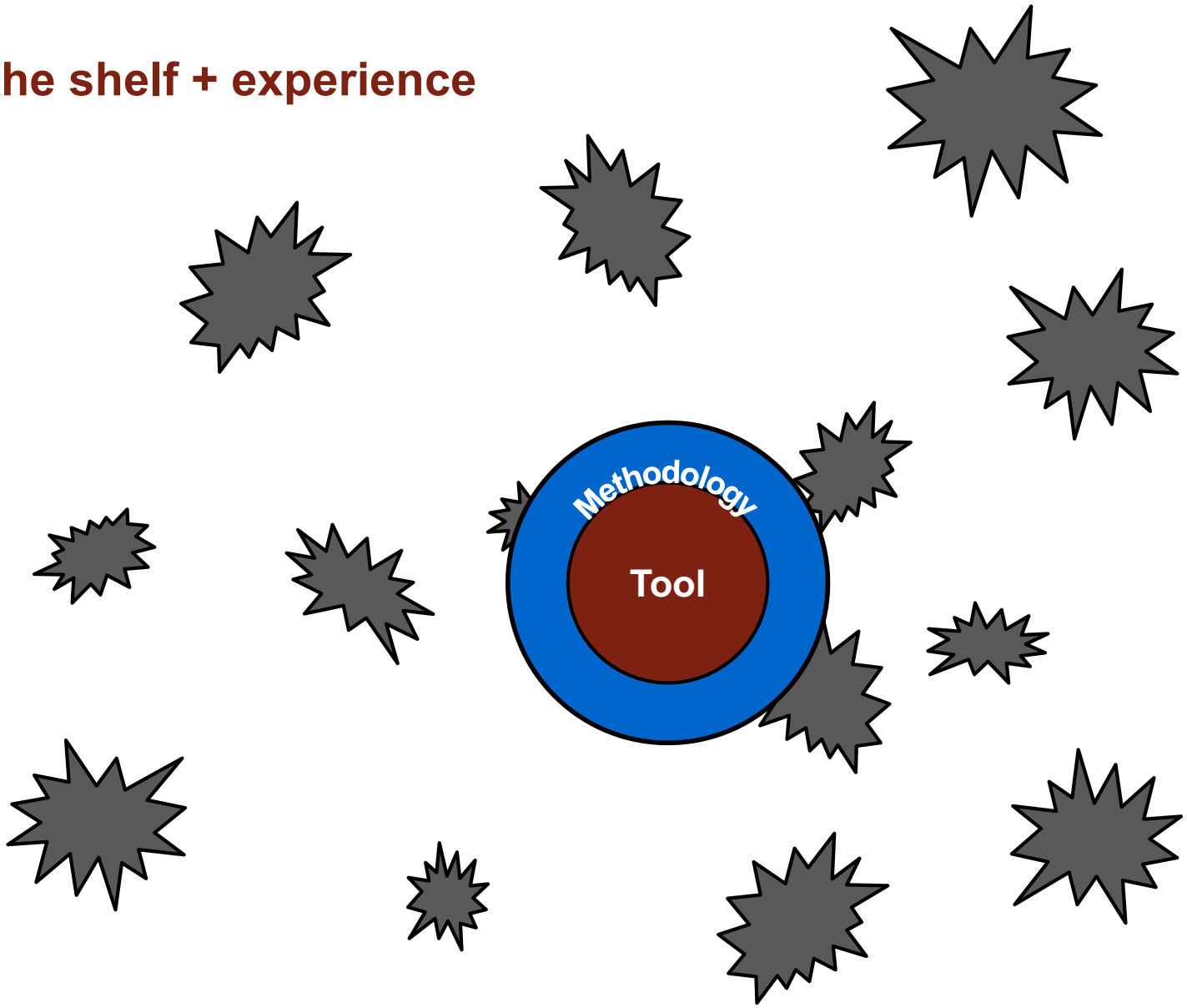
Region of Productivity

Off the shelf EDA tool



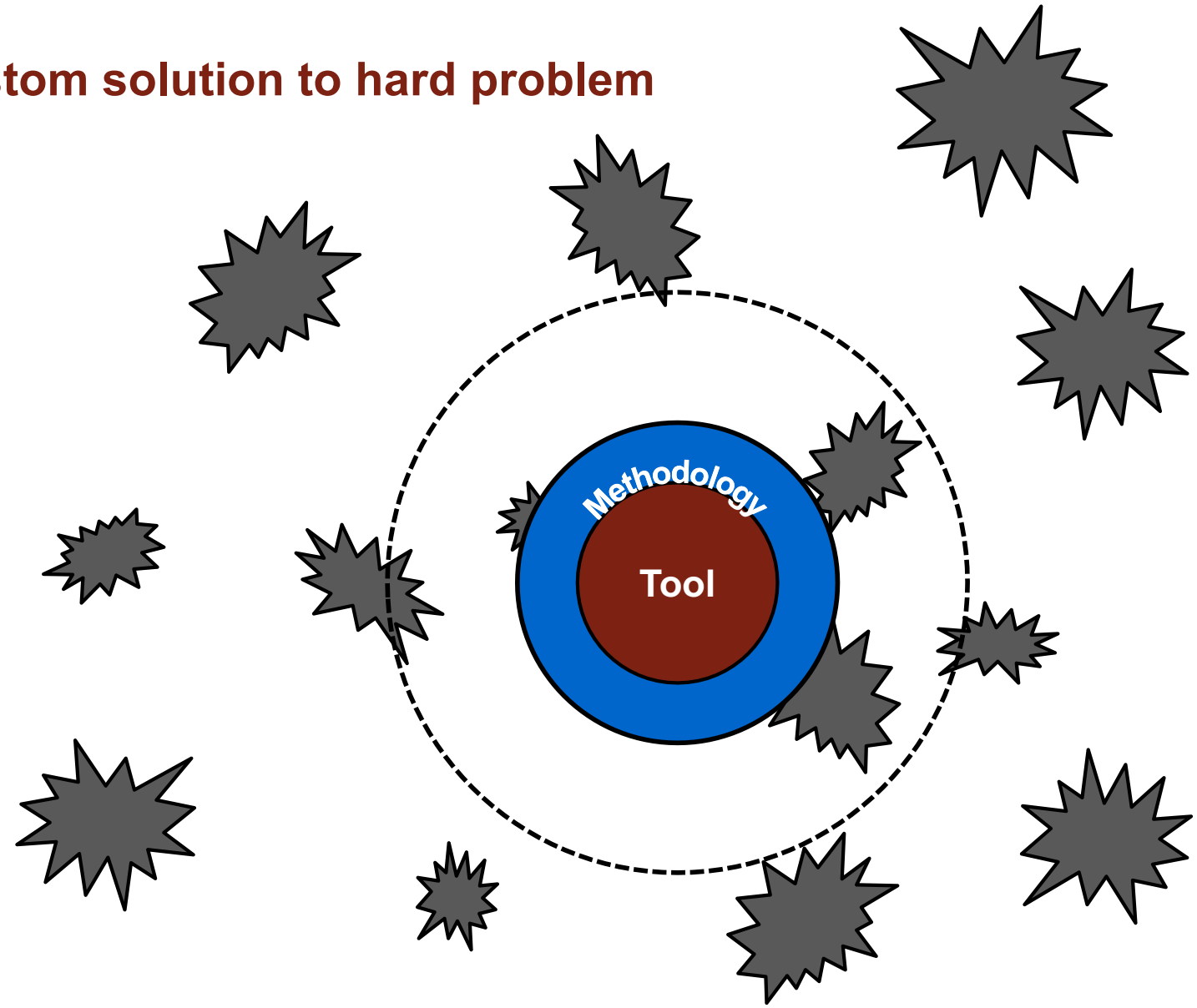
Region of Productivity

Off the shelf + experience



Region of Innovation

Custom solution to hard problem



Life in the Region of Innovation

Tackle problems where traditional methods fail

- mission critical functionality
- areas with no good solution
- areas where formal can be cheaper, faster

Needs experience and intellectual ‘tool kit’

- insight into problem structure
- mastery of theoretical and conceptual vocabulary
- feel for technology fit, capacity, and limits

New applications drive technological innovations

- often a combination of technology and methodology

Example - SoC Formal Verification

Conventional formal verification

- traditionally operates at block level
- focuses on exhaustive verification of functionalities, or bug-finding
- performed by designers or DV engineers of the particular design

SoC level verification

- performed at subsystem or system level
- focuses on how blocks work together in the system, rather than the functionality of a particular block
- performed by system engineers who may not be familiar with the individual components of the system

[Laurence Loh – Jasper Design Automation, 2010]

Potential Roadblocks

Applicability

Many tasks in system-level verification may seem unrelated to what traditional formal does.

Capacity

Full system may be 10s of millions of gates, considered beyond most normal formal capacity.

Setup for verification

SoC-level verification environment is often very involved and stimuli are generated at a much higher level.

Expertise

[Laurence Loh – Jasper Design Automation, 2010]

Overcoming Roadblocks

Capacity

- Full-chip functions structurally involve most of the chip
- Divide-and-conquer, but achieve full verification by post-processing
- Each application needs something specific to address capacity

Complexity and sequential depth

- Using intermediate information (points/events) to optimize formal algorithm and under-the-hood abstractions

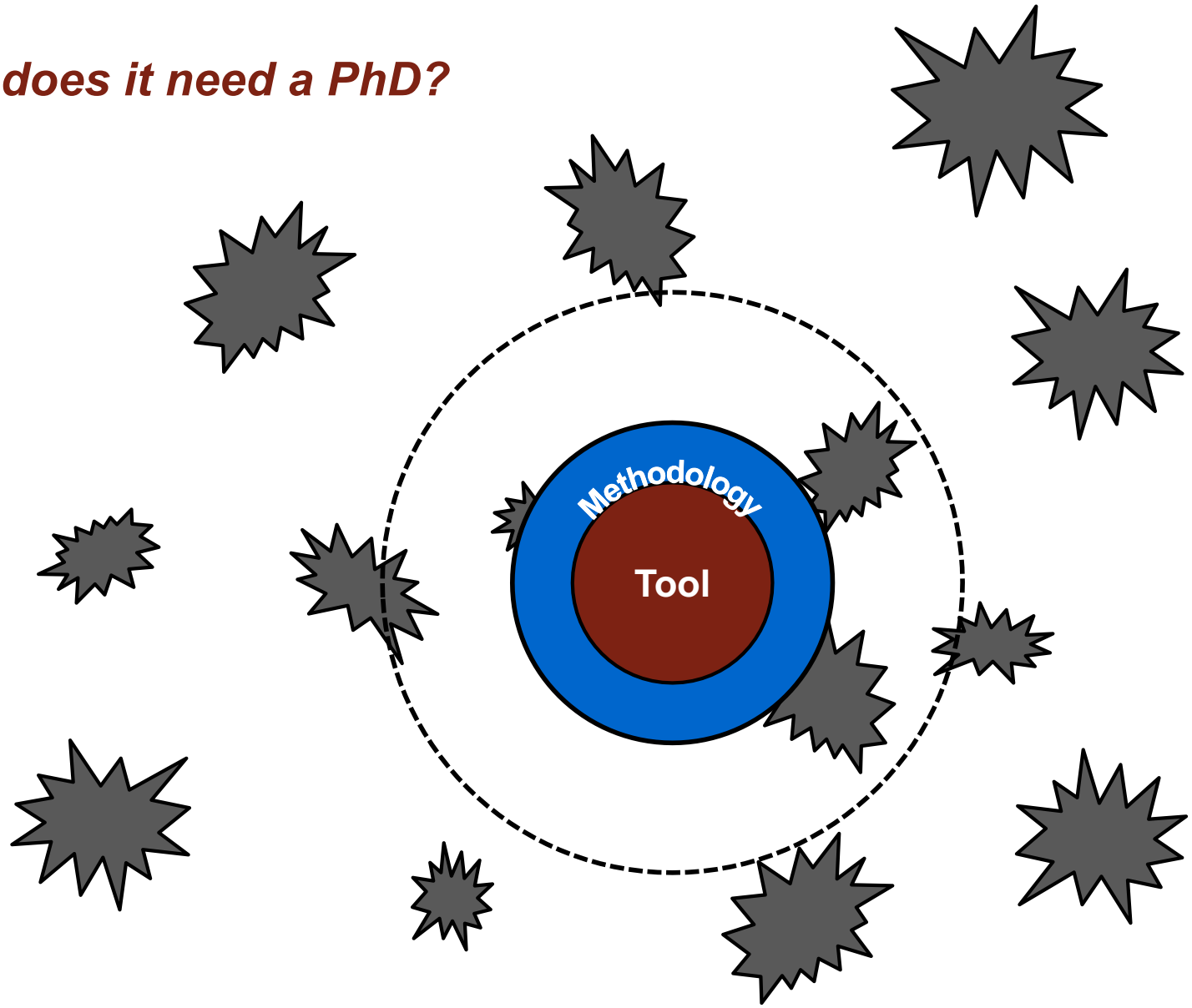
Setup and expertise

- Automation allows reuse on subsequent SoC, or ongoing regression
- Replacing items on testplan
- Hide formal so that users do not need to be experts

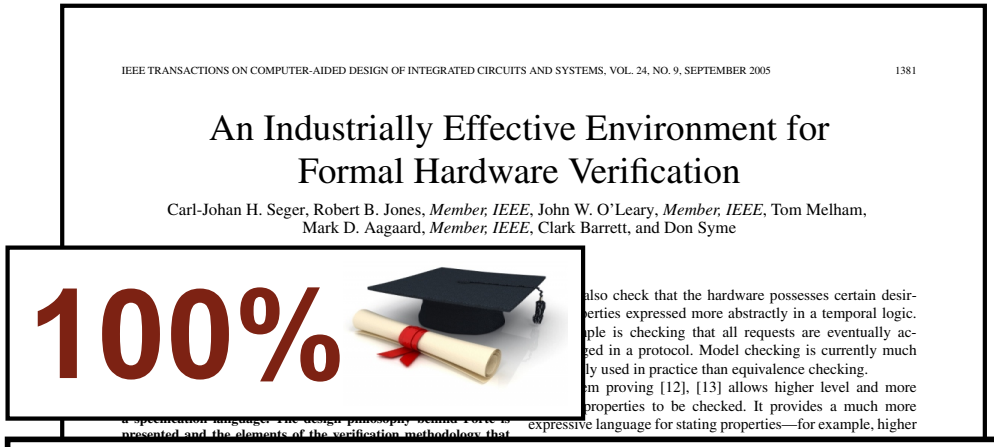
[Laurence Loh – Jasper Design Automation, 2010]

Region of Innovation

But does it need a PhD?

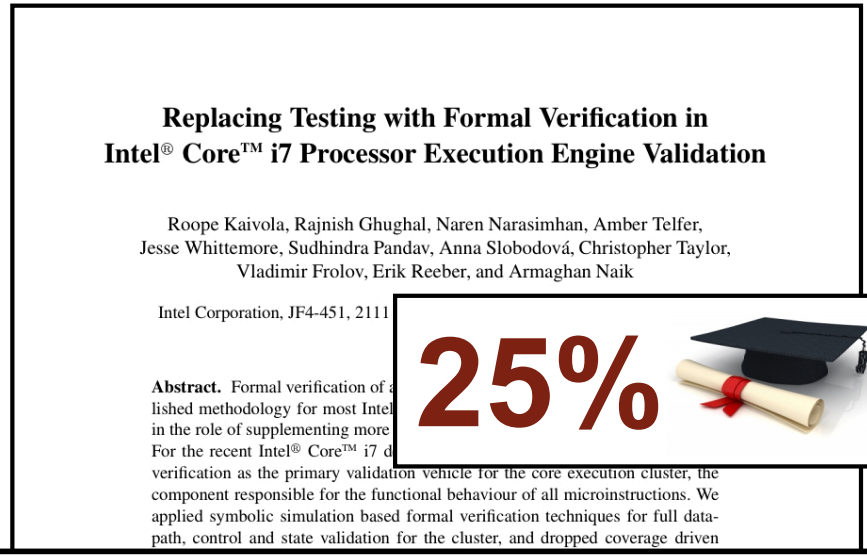


Forte Data Points


IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 24, NO. 9, SEPTEMBER 2005 1381

An Industrially Effective Environment for Formal Hardware Verification

Carl-Johan H. Seger, Robert B. Jones, *Member, IEEE*, John W. O'Leary, *Member, IEEE*, Tom Melham, Mark D. Aagaard, *Member, IEEE*, Clark Barrett, and Don Syme


Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation

Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik

Intel Corporation, JF4-451, 2111

Abstract. Formal verification of a

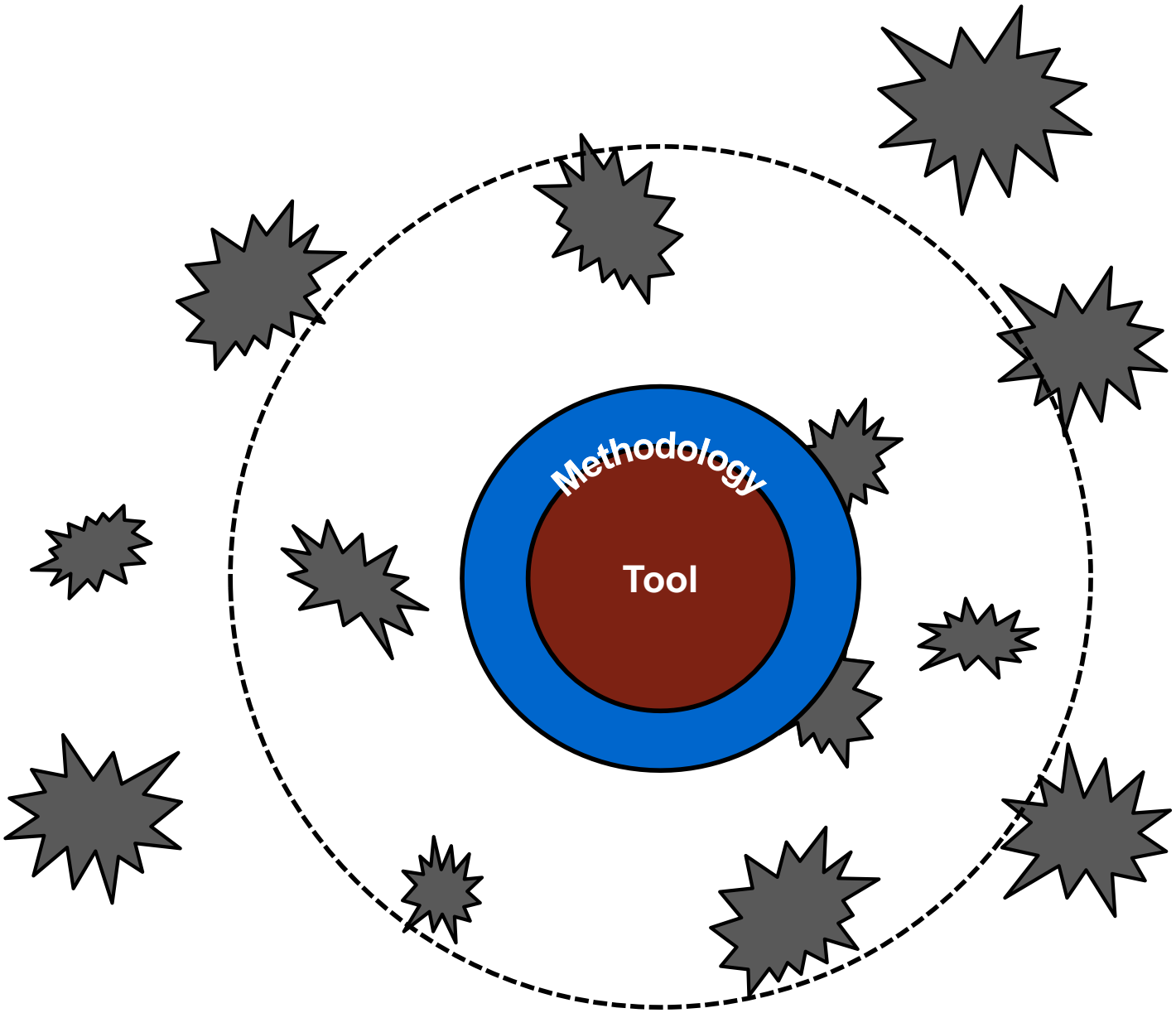
0%

"We have some very successful groups elsewhere, doing complete datapath (and some non-trivial control) STE verification, whose members have bachelor degrees only.

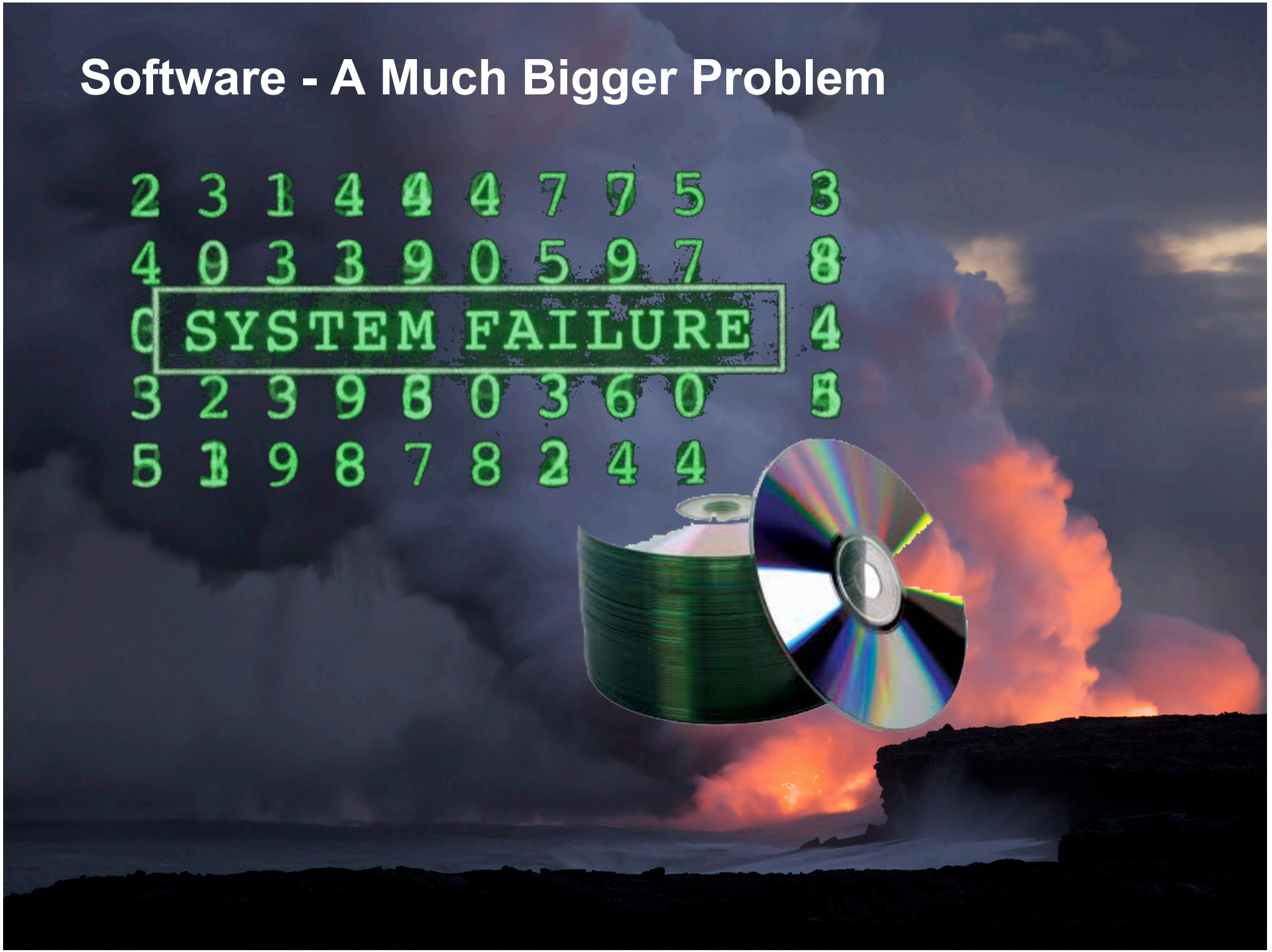
There is … a perception that a PhD is needed. Maybe, and only maybe, I'll agree for the initial development. However, when it comes to creating new verifications of new designs in new areas, micro architectural knowledge is far more important than your degree."

- Carl Seger, Intel

Region of Research?



Software - A Much Bigger Problem



Embedded Software

What we know

- Is - or will be - a great pain point for systems design
- Very diverse space of application problems
- The general problem is very hard
- Potentially strategic for EDA, but business model unclear

No established methodology for FV of low-level software

- What are the right problems to attack?
- How are the benefits realized?
- What is the right technology?

Low-Level *Firmware*

It’s everywhere now

Working Definition

Software code that is

- distributed and stored inside the product
- essential for device function
- hidden from the end-user
- intimately entangled with hardware devices
- low level, C + inline assembly

Growing software-as-hardware paradigm

- software-controlled ‘device divers on-a-chip’ (of devices)
- validation/analysis needs both HW and SW models

Software development gated by HW design time

Debugging is difficult

Effective Validation of Low-Level Firmware

Luke Ong
Oxford

Sharad Malik
Princeton

Tom Melham
Oxford

Daniel Kroening
Oxford

Alan Hu
UBC

Moshe Vardi
Rice

Intel Sponsor

Jim Grundy

Ongoing Work

Start of an ongoing, integrated effort in Oxford on this topic