Formal Verification Adoption

Mike Bartley
TVS, Founder and CEO
Agenda

- Some background on your speaker

- Formal Verification
  - An introduction
  - Basic examples
  - A FIFO example

- Adoption
Your speaker: Mike Bartley

- PhD in Mathematical Logic
- MSc in Software Engineering
- MBA

- Worked in software testing and hardware verification for over 25 years
  - Verification of safety-related hardware and software
  - Formal verification of both software and hardware

- Started TVS in 2008
  - Software testing and hardware verification products and services
  - Offices in India, UK, France and Germany
Number of Employees by quarter

- USA - 2014
- UK - 2008
- Germany - 2011
- France - 2012
- China
- South Korea
- India - 2011
- Singapore - 2014
- Japan 2015

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEES</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>
Functional Verification Approaches

- Verification
  - Static
    - Reviews
    - Code Analysis
      - Linters
    - Equivalence Checking
  - Formal
    - Property Checking
  - Dynamic Formal
    - Theorem Proving
  - Dynamic
    - Simulation
    - Prototyping
      - Silicon
      - FPGA
      - Emulation
The Role of Simulation

- Most widely used verification technique in practice
- Complexity of designs makes exhaustive simulation impossible in terms of cost/time.
  - Engineers need to be selective
  - Employ state of the art coverage-driven verification methods
  - Test generation challenge
- Simulation can drive a design deep into its state space.
  - Can find bugs buried deep inside the logic of the design
- Understand the limits of simulation:
  - Simulation can only show the presence of bugs but can never prove their absence!
Simulation Depth-first vs. Formal Breadth-first

- Where the nodes are states in the simulation
- And the arcs are clocked transitions
- **But the trees are**
  - Very wide
  - Very deep
Introduction: Simulation vs Functional Formal Verification

In practice, completeness issues and capacity limits restrict formal verification to selected parts of the design.

Naive interpretation of exhaustive formal verification:

Verify ALL properties.

Challenge 1:
Specify properties to cover the entire design.

Challenge 2:
Prove all these properties.

Only selected parts of the design can be covered during simulation.

Formal Verification – terms of reference

- **Model Checking**
  - Requirements of a design are expressed in a formal mathematical language
  - Tools are used to analyze whether there is a way that a model of the design fails to satisfy the requirements

- **Not covered here**
  - Equivalence Checking
    - Tools are used to analyze whether one model of a design is a “correct” implementation of another
  - Formal modelling and proofs

Currently mainly RTL-Gates, Gates-Gates
Checking of sequential retiming possible
But SystemC to RTL is appearing
Inputs to Formal

- **3 inputs to the tool**
  - A model of the design
  - A property or set of properties representing the requirements
  - A set of assumptions, expressed in the same language as the properties
    - typically constraints on the inputs to the design

- **For example**
  - Usually RTL
  - Items are transmitted to one of three destinations within 2 cycles of being accepted
    - \((\text{req\_in} \&\& \text{gnt\_in}) \Rightarrow \##[1;2] \left(\text{rec\_a} \mid\mid \text{rec\_b} \mid\mid \text{rec\_c}\right)\)
  - The request signal is stable until it is granted
    - \((\text{req\_in} \&\& \neg\text{gnt\_out}) \Rightarrow \##1 \text{req\_in}\)
    - We would of course need a complete set of constraints
Model Checking – Outputs from the tools

- **Proved**
  - the property holds for all valid sequences of inputs

- **Failed($n$)**
  - there is at least one valid sequence of inputs of length $n$ cycles, as defined by the design clock, for which the property does not hold.
  - In this case, the tool gives a waveform demonstrating the failure.
  - Most algorithms ensure that $n$ is as small as possible, but some more advanced algorithms don’t.

- **Explored($n$)**
  - there is no way to make the property fail with an input sequence of $n$ cycles or less
  - For large designs, the algorithm can be expensive in both time and memory and may not terminate
Some example properties

- a_busy and b_busy are never both asserted on the same cycle
- if the input ready is asserted on any cycle, then the output start must be asserted within 3 cycles
- if an element with tag \( t \) and data value \( d \) enters the block, then the next time that an element with tag \( t \) leaves the block, its data value is the same as the output of a reference piece of combinatorial logic for which the input is \( d \)
- stall cannot remain high indefinitely

\[
\begin{align*}
\text{Can be checked during simulation (but not proved by simulation)} & \\
\text{A liveness property} & \\
\end{align*}
\]
Safety Properties

- **Safety**: Something bad does not happen
  - The FIFO *does not* overflow.
  - The system *does not* allow more than one process to use a shared device simultaneously.
  - Requests are answered within 5 cycles.

- **More formally**: A safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property.

  [Accellera PSL-1.1 2004]

Safety properties can be falsified by a finite simulation run.
Liveness Properties

- **Liveness**: Something good eventually happens
  - The system *eventually* terminates.
  - Every request is *eventually* acknowledged.

- **More formally**: A *liveness property* is a property for which any finite path can be extended to a path satisfying the property.
  

In theory, liveness properties can only be falsified by an infinite simulation run.

  - Practically, we often assume that the “graceful end-of-test” represents infinite time.
  
  *If the good thing did not happen after this period, we assume that it will never happen, and thus the property is falsified.*
Identifying Properties for the FIFO block

Black box view:

(P) • Empty and full are never asserted together.
(P) • After clear the FIFO is empty.
(D) • After writing 8 data items the FIFO is full.
(D) • Data items are moving through the FIFO unchanged in terms of data content and in terms of data order.
(D) • No data is duplicated.
(D) • No data is lost.
(D) • data_out_valid only for valid data, i.e. no x’s in data.

- Distinguish between design and protocol properties.
  - Similar to functional coverage.
  - Protocol assertions have high re-use value.
  - Design assertions may be re-used at high abstraction levels
    - Re-use of architectural assertions, i.e. at ISA level.
The Strengths of Model Checking

- **Ease of set-up**
  - No test bench required, add constraints as you go, VIP?

- **Flexibility of verification environment**
  - Constraints can be easily added or removed

- **Full proof**
  - Of the properties under the given constraints
  - (Can also prove “completeness” of the properties)

- **Intensive stressing of design**
  - Explored(n) constitutes a large amount of exploration of the design
  - Judgement when the number of cycles explored in a run is sufficient
    - Significant bugs already found within this number of cycles

- **Corner cases**
  - Find any way in which a property can fail (under the constraints)
Potential issues with formal verification

- **False failures**
  - Need constraints to avoid invalid behaviour of inputs

- **False proofs**
  - Bugs may be missed in an over-constrained environment.

- **Limits on size of the model that can be analysed**

- **Non-exhaustive checks: Explored(n)**
  - Interpret the results
  - *Can require significant knowledge and skill*

- **Non-uniform run times**
  - Often it cannot be predicted how long it will take for a check either to terminate or to reach a useful stage

**This can make formal unpredictable!**
A Taxonomy of Methodologies (AHAA!)

- **Bug Avoidance**
  - Improve quality before any property checks are run
    - Visualization
    - Clarification of spec

- **Bug Hunting**
  - Use model checking to look for bugs
  - Do not worry if proofs do not complete

- **Bug Absence**
  - Aim to ensure that properties are fully proven
  - Aim to get a “complete” set of properties

- **Bug Analysis**
  - For bugs in FPGA prototypes or in Silicon
    - *It may be hard to recreate the conditions that causes a bug*
    - *By writing the symptom of the bug as a property, one can generate a waveform that can be analysed*
Exploiting “Bug Avoidance”

- **Description**
  - Enable designers to do early bring-up and use of visualisation

- **Advantages**
  - Low cost, starts very early in design process
  - Potential to save huge amounts of time and cost

- **Objectives**
  - Specification clarification
  - Avoid putting in bugs in the first place

- **Strengths of formal verification exploited**
  - Ease of set-up
  - Re-use of designer assertions
Exploiting “Bug Hunting”

- **Description**
  - Perform regular checks of properties written by designers or verification engineers

- **Advantages**
  - Low cost, starts early in design process

- **Objectives**
  - To find bugs not found by other processes at this stage of design development
  - Or to find bugs more quickly and that are easier to debug

- **Strengths of formal verification exploited**
  - Ease of set-up
  - Corner cases
Avoiding issues: “Bug Hunting”

- **False failures**
  - Consider the structural level of hierarchy to run the tool
    - At a high structural level where the constraints are simple
    - Or at block level, where the designer has a good knowledge of the behaviour expected of the inputs.
  - False failures lead to wasted debug effort but do not lower the quality of the verification

- **Non-exhaustive checks**
  - Not an issue
    - Full proofs are welcome, but not an objective of this process

- **Non-uniform run times**
  - Not an issue
    - Checks are run just for the time available.

- **Completion criterion**
  - No failures
Automation for “Bug Hunting” - apps

- X propagation checks
  - Do we come out of reset correctly etc

- Interconnect checks
  - Define connectivity (in a spreadsheet) and let the tool automatically generate checks
  - Useful at SoC level
    - But limits on the size of model that can be analysed

- Dead code checks
- FSM checks
- Security checks
- Unreachable code
  - Generation of waivers for coverage analysis closure
Exploiting “Bug Absence”

- **Description**
  - Try to prove properties representing important design requirements
    - no deadlock, protocol compliance, user can’t modify registers without correct permissions, ...

- **Objectives**
  - Complete assurance that the property can never be violated
    - (under the given constraints)

- **Advantages**
  - This is the only way of getting such complete assurance

- **Strengths of formal verification exploited**
  - Complete proof
  - Corner cases
Avoiding potential issues: “Bug Absence”

- **False failures**
  - Write a complete set of environment constraints

- **Non-exhaustive checks**
  - Are full proofs required?

- **Non-uniform run times**
  - Use different proof engines with the tool
  - Use “invariants” (helper properties)
  - Use safe abstractions
  - Prove under certain conditions
    - Add extra constraints

- **Completion criterion**
  - Property exhaustively proved
  - Or at least exhaustively proved under certain conditions
Formal in the design flow

- Design stages
  - Architecture definition
  - Micro-architecture definition
- Formal processes
  - Design
  - Validation
  - Maturity
  - Support
  - Bug avoidance
  - Bug hunting
  - Bug absence
  - Bug analysis
- Design stages:
  - U-arch explorations
  - Interface specifications
  - Proof convergence techniques
  - Implementation verification
  - Embedded assertion flow
  - X-propagation
  - Review
  - Proof convergence techniques
- Project timeline
  - Formal errata analysis
  - Formal specification and validation
  - Bug avoidance
  - Bug hunting
  - Bug absence
  - Bug analysis
  - Design Bring-up
Security Verification – looking at data flow

Untrusted (Wireless Radio) -> Critical (Insulin pump)
Secret data is not unintentionally leaked
Hardware Security Verification

- Security Verification mainly involves Data Flow Verification
  - Insecure data does not flow to secure location
  - Secure data does not flow to insecure location

- Very difficult to do in simulation
  - Hard to be exhaustive

- A natural choice for formal
  - But we need to extend SVA to support this
Case Study – Key Flowing Out Of Design

- **Assertion: Key only flows through AES**
  - assert iflow (key =/= $all_outputs$ ignoring aes.$all_outputs$);
  - If assertion holds, key only flows to outputs through AES first

- **Real world results**
  - State-of-the-art design with over 10 million gates
  - Actual required properties, impossible to visually inspect
Case Study – Key Flowing Out Of Design

- **Assertion:** Key only flows through AES
  - assert iflow (key =/= $all_outputs$ ignoring aes.$all_outputs$);
  - If assertion holds, key only flows to outputs through AES first

- **Real world results**
  - State-of-the-art design with over 10 million gates
  - Actual required properties, impossible to visually inspect
Case Study – Key Flowing Out Of Design

- **Assertion: Key only flows through AES**
  - assert iflow (key =/= $all_outputs$ ignoring aes.$all_outputs$);
  - If assertion holds, key only flows to outputs through AES first

- **Real world results**
  - State-of-the-art design with over 10 million gates
  - Actual required properties, impossible to visually inspect
Secure Data over Packet Channel

- No direct association between signal name and secure / non-secure!
  There’s a control / temporal component also.

- Secure Formal does not help in analyzing strength of Cryptography blocks.

**RED** = secure
**Green** = non-secure
**Blue** = mixed
Data Leakage

- Identify secure signals [sources] we are concerned about (typically not many)

- [Auto] identify all the places it might be able to reach (typically hundreds or thousands – ALL the outputs, or top level signals of the block)

- Confirm that signal can ONLY reach the places it’s supposed to, and not anyplace where the bad guys could steal it.

- **No way to directly specify this in SVA!**
Strategic Issues with Formal

- **What simulation do I replace?**
  - This needs to be part of the planning
  - Metrics are difficult to combine but becoming easier

- **We don’t know if or when it will complete**
  - Formal can take a long time to give very poor results

- **A high level of skill might be required**
  - To write the correct properties and constraints
  - To drive the tools
  - And to drive into bug avoidance in the future

- **So why bother?**
  - You can “get it for free” on the back of assertion-based verification
  - There are requirements that cannot be verified through simulation
    - Cache coherency, liveness, deadlock,…
  - We need it to cope with the increasing complexity of verification
How do I get started with Formal?

- **“Out of the Box”**
  - Target packaged solutions
  - Easy but of limited value

- **Real exploitation requires strategic investment**
  - Training for writing “bug hunting” properties
    - Standardise on when, where and how to write
  - Automation of the flows

- **Create bug absence experts**
  - Requires careful selection and training
  - Centralise the skills?
  - These people will also be good at bug analysis

- **Bug avoidance**
  - Train your designers on how to use the tools
Summary

- **Formal verification**
  - Will continue to increase in importance
  - More companies will begin to use it

- **Decide how you want to use it**
  - Avoidance, Hunting, Absence, Analysis
  - Apps provide an excellent adoption route for bug hunting

- **And then invest accordingly**

- **This is a strategic play**
  - It requires strategic investment
  - It won’t happen through the efforts of a few interested individuals