Practical Approaches to Formal Verification

Test and Verification Solutions
Delivering Tailored Solutions for Hardware Verification and Software Testing
Static vs. Dynamic

Verification

Static

Reviews

Code Analysis

Linters

Equivalence Checking

Dynamic

Simulation

Prototyping

Dynamic Formal

Silicon

FPGA

Emulation

Formal

Model Checking

Theorem Proving
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 modeling 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
    - \((\text{req}_\text{in} \land \text{gnt}_\text{in}) \rightarrow \##[1;2](\text{req}_\text{a} \lor \text{req}_\text{b} \lor \text{req}_\text{c})\)
  - The request signal is stable until it is granted
    - \((\text{req}_\text{in} \land \lnot \text{gnt}_\text{out}) \rightarrow \##1\text{req}_\text{in}\)
    - *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

Can be checked during simulation (but not proved by simulation)

A liveness property
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 a 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: $\text{Explored}(n)$**
  - 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

- **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 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”

- **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**
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**
  - Not an issue
  - *full proofs are welcome, but not an objective of this process*

- **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

*Requires a lot of expertise and skill*
Formal in the design flow
Strategic Issues with Formal

- **What simulation do I replace?**
  - Short answer is none unless block is done completely formally
  - The metrics are too different

- **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”**
  - Easy but not of significant 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 is a longer term goal**
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

- **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