Simulation-Based Verification Technologies at IBM

Moshe Levinger
Hunting for bugs …

Playing against odds ... and making the right bets

Winning big time

Verification Expert = Warrior + Gambler

Applying a variety of tools
Talk Outline

- Driving Forces behind Verification Innovation
- Ready-Made Test-Program Generators
- Coverage-Directed Generation by Feedback
- Coverage-Directed Generation by Construction
- Summary & Future Directions
Key Ingredients for Successful Verification

- Automation
- Quality
- Productivity

Better Products  Shorter TTM  Lower Costs
Mapping Quality and Productivity Needs into Requirements for Verification Technologies

Quality

- Bug-aware
- Coverage-driven
- Domain expertise

Productivity

- Verification IP reuse
- Verification task description
- Verification task fulfillment
- Updates and maintenance
Typical Flow of Simulation-Based Verification

- Test Plan
- Directives
- Biased-Random Test-Program Generator
- Checking, Assertions
- Design Under Test
- Simulator
- Coverage Information
- Coverage Reports
- Fail
- Pass

Checking, Assertions

Coverage Analysis Tool

Test

Coverage Reports

Directives
Ready-Made Test Program Generators

- Model-based test-case generators which are applicable for a variety of architectures and designs
- Two key representatives of this concept:
  - Genesys-Pro for Processor Verification
  - X-Gen for System Verification
- Generic architecture-independent test generation engine
- External formal and declarative description of the DUV
- Powerful test description language
  - Ranging from completely random to fully specified test cases
- Open architecture for incorporation of Testing Knowledge
Genesys-Pro Usage Scheme

- **Def file:**
  - 100 instructions
  - 5 tests

- **Users**

- **HDL simulator**

- **Checking:**
  - On-the-fly
  - Expected results

- **Test 1**
  - 100 x instruction instance

- **Genesys-Pro**
Genesys-Pro Output: Sample Test File

Resource initialization

Instruction sequence

Predicted Results
Model-Based Test Generation – High Level Concept

Model of the Verified Subject

Terminology
Application Details

Test Generator Developer

Verification Engineer

Request File

CSP Solver
Test Generation Engine

Test

Application Engineer
Innovative Test Template Definition

- Allows the user to define delicate verification scenarios using the full power of programming-like language:
  - Sequence, permute, select, repeat, if-then-else
  - Variables, assignments, expressions, conditions
Industry Leading CSP-based Technology

- Allows the user to request any possible set of constraints defining a verification scenario
- Provides uncompromising and powerful solutions for complex sets of constraints
- Copes with unique CSP characteristics:
  - Random, uniform distribution solution - as opposed to one, all, or “best” solution
  - Huge variable domains, e.g., address spaces
Declarative, Form-based Modeling Environment

- A modeling environment for
  - Instructions, Resources
  - Components, Interactions
  - Testing knowledge

- Different ‘forms’ are provided to describe various aspects of the design under verification

- No need to write code
  - But, if needed - special cases can be modeled through C++ hooks
Generic Testing Knowledge

- A set of mechanisms that aim at improving test-case quality
- Capitalize on recurring concepts:
  - Address translation
  - Pipelines
  - Caches
- The basic mechanism: non-uniform random choice
  - Bias towards ‘interesting’ areas
- Examples:
  - Resource contention
  - Translation table entry reuse
  - Data placement

Space of valid tests

‘interesting’ areas
Meeting the Quality and Productivity Requirements

**Quality**
- Bug-aware
- Coverage-driven
- Domain expertise

**Productivity**
- Verification IP reuse
- Verification task description
- Verification task fulfillment
- Updates and maintenance
- Ready-made concept
- Rich input language
- Powerful CSP engines
- Declarative model
- Testing Knowledge
Typical Flow of Simulation-Based Verification

Directives

Biased-Random Test-Program Generator

Test Plan

Test

Checking, Assertions

Design Under Test

Coverage Information

Coverage Analysis Tool

Coverage Reports

Pass

Fail
CDG -- Coverage-Directed-Generation: Closing the Loop using Machine Learning Techniques

- **Motivation**
  Coverage analysis tools can assess the quality of a set of test cases but cannot recommend how to improve the set.

- **Objectives**
  Introduce a feedback loop to tune test generation
  - Stimulate hard-to-reach coverage events
  - Improve rate of coverage
  - Control coverage space distribution
Coverage-Directed-Generation: Closing the Loop using Machine Learning Techniques

Approach

- **Use** Bayesian networks to represent the CDG ingredients
- **A natural and compact representation of the distribution space**
- **Enables encoding of essential domain knowledge**

```cpp
cp_cmd_enable = 
  { // sift relative 
    // mode weight
    { 0x8, 30-35 },
    { 0x2,  7 },
    { 0x1,  00 },
    { 0xE,  1-10 };

cp_core_enable = 
  { // sift relative 
    // mode weight
    { 0x1, 10-100 },
    { 0x2,  10-100 },
    { 0x3,  10-100 };
```
Employing Bayesian Networks for CDG

Directive Space

Directive Generator → Simulator → Coverage Tool → Events

Directive Space

Mapping

Coverage Space

D1 → C1
D2 → C2
Dm → Cn

Mapping
Example: IBM Mainframe’s Storage Control Element

- **Coverage model:**
  - All command-response combinations between 32 CPUs and 4 SCEs
  - 18816 legal and interesting events

- **CDG results:**
  - 100% of reachable space covered
  - Significantly less cycles required
  - Two large holes identified
CDG Status at IBM

- **CDG is bringing value to IBM’s projects**
  - We identified domains in which CDG can improve coverage
  - Several success stories
  - More effort is needed to become part of the standard verification methodology

- **The CDG Promise**
  - **Less time** – start hitting interesting areas early
  - **Less resources** – less cycles and human effort to reach goals
  - **Better verification** – higher coverage, more ways to hit events, hitting corner cases
## Meeting the Quality and Productivity Requirements

### Quality
- **Bug-aware**
  - -
- **Coverage-driven**
  - Coverage as a target
- **Domain expertise**
  - Bayesian Network encoding

### Productivity
- **Verification IP reuse**
  - -
- **Verification task description**
  - Existing coverage models
- **Verification task fulfillment**
  - Machine Learning scheme
- **Updates and maintenance**
  - -
Typical Flow of Simulation-Based Verification

1. Directives
2. Test Plan
3. Deep Knowledge Generator
4. Test
5. Simulator
6. Checking, Assertions
7. Design Under Test
8. Coverage Analysis Tool
9. Coverage Information
10. Coverage Reports
11. Fail
12. Pass
Deep Knowledge, Coverage-based Test Generation

Coverage Model

Domain Knowledge

Constraint Satisfaction Engine

Complete Coverage of All Tasks
FPgen: Deep-Knowledge Test Generator for Floating-Point

- **FPgen** – generic solution for floating-point verification

- **FPgen** aims to fulfill comprehensive FP test plans:
  - Definition of architecture tasks
  - Definition of micro-architecture tasks
  - Coverage model language

- **FPgen** has a deep understanding of the floating-point world
  - Enables the tool to fulfill complex FP tasks
Motivation for Building FPgen: Floating Point Bugs

- FP bugs are often uncovered in late stages of the design life

![Diagram showing number of bugs over time for different categories of bugs.]

- General bug curve
- FP bug curve

Number of bugs vs. Time
Basic Functionality for a Given Verification Task

Multiple random solution:
Control of coverage density

Denormal
Max 4 bits set
**FPgen Input: a Floating-Point Data-path Coverage Model**

**Example: All Types model**

<table>
<thead>
<tr>
<th>Operand1</th>
<th>Multiply</th>
<th>Operand2</th>
<th>Result</th>
</tr>
</thead>
</table>
| +/- Infinity  
+/- Zero  
+/- Norm  
+/- Denorm  
+/- Large number  
+/- Small number  
+/- Min Denorm  
+/- Max Denorm  
+/- Min Norm  
+/- Max Norm | **X** | +/- Infinity  
+/- Zero  
+/- Norm  
+/- Denorm  
+/- Large number  
+/- Small number  
+/- Min Denorm  
+/- Max Denorm  
+/- Min Norm  
+/- Max Norm | **X** | +/- Infinity  
+/- Zero  
+/- Norm  
+/- Denorm  
+/- Large number  
+/- Small number  
+/- Min Denorm  
+/- Max Denorm  
+/- Min Norm  
+/- Max Norm |
FPgen Overall Solving Scheme Flow

Task

Choose engine

Choose search engine

Analytic

Not found

Found

Output no solution

Output the solution

Output the solution

Output no solution

Unknown

Binary search

Stochastic Search

SAT

Reduction

...
Towards a Fully Automated Solution: The Generic Test Plan

- IEEE test suites
- IEEE standard
- Bug analysis
- Papers
- Test plans from users

Generic & alternative Implementations

Test-Plan

Theorem the number a,b,c,...
### Meeting the Quality and Productivity Requirements

#### Quality

<table>
<thead>
<tr>
<th>Bug-aware</th>
<th>Bug-driven models in GTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage-driven</td>
<td>Coverage model as input</td>
</tr>
<tr>
<td>Domain expertise</td>
<td>Deep FP knowledge</td>
</tr>
</tbody>
</table>

#### Productivity

<table>
<thead>
<tr>
<th>Verification IP reuse</th>
<th>Generic Test Plan concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification task description</td>
<td>Dedicated input language</td>
</tr>
<tr>
<td>Verification task fulfillment</td>
<td>Powerful FP solvers</td>
</tr>
<tr>
<td>Updates and maintenance</td>
<td>Full support for IEEE standard</td>
</tr>
</tbody>
</table>
Summary & Future Directions

- Three different paradigms to address the Quality and Productivity requirements:
  - Ready-Made Test-Program Generators
  - Coverage-Directed Generation by Feedback
  - Coverage-Directed Generation by Construction

- Raising the entry quality of design code
- Applying model-based, ready-made concepts to:
  - Test-bench verification environments
  - Post silicon verification

- New techniques for bug exploration and bug analysis
- Hybrid verification
- Verification of high-end multi-core and multi-threaded systems
Thank you