Covalidation of Hardware-Software Systems

Ian G. Harris

Department of Computer Science
University of California Irvine
Outline

• Hardware/Software Codesign and Covalidation
• Evaluation of Fault Models
• Timing Fault Model
• Automatic Test Generation
• Conclusions
```c
int foo (int in1, int in2)
int a, b, c;
a = in1 + in2;
b = 0; c = 0;
while (c < a)
```

**Specification**: Describe the behavior of each task

**Partitioning**: Group tasks to be implemented together

**Allocation**: Map partitions to hardware

**Scheduling**: Ordering the execution of tasks

**Communication Synthesis**: Map data transfers to comm. structures
Hardware-Software Codesign/Covalidation Flow

- Design is refined at each step
- Each design step may introduce errors
Validation/verification is a bottleneck in the design process

- High cost of design debug (designers, time-to-market)
- High cost of faulty designs (loss of life, product recall)

Hardware/Software covalidation problem is more acute

- Hardware and software are often used together
- Hardware and software are designed separately
- Covalidation is performed late in the process, necessitating long redesign loops
General Verification Approaches

- **Formal verification**
  - Time complexity is high
  - Confidence is high for specified properties

- **Simulation based validation**
  - Full-chip validation
  - Confidence is lower

- Pentium 4 bugs found by FV (492) vs. Validation (5809) [1]

Test generation is either manual or automatic.

Covalidation fault model is needed to direct test generation.
Challenges in Hardware-Software Covalidation

- **Covalidation Fault Model**
  - Describes a set of potential design defects
  - Provides goals for test generation

- **Automatic Test Generation**
  - Create a test sequence which guarantees detection of design defects
  - Allows the covalidation process to be more fully automated
A covalidation fault models the behavior of a set of design errors.

Properties of a covalidation fault model:

- **Modeling Accuracy**
  - Detection of all faults should imply the detection of a set of modeled faults.

- **Model Size**
  - Few faults should model many design errors.

---

“Covalidation Fault Models”

Diagrams illustrating the relationships between covalidation faults, design errors, and unmodeled errors.
The defect associated with each line is non-specific. Detection of the fault is assumed if the statement is executed.

Assume that a defect is associated with a single statement in the behavioral description.

```c
int foo (int in1, int in2)
1. int a, b, c;
2. a = in1 + in2;
3. b = 0; c = 0;
4. while (c < a)
5.   c = c + in1;  Each line could have a defect.
6. if (c < in2)
7.   return (a + b);
8. else
9.   return (a + c);
```

- The defect associated with each line is non-specific.
- Detection of the fault is assumed if the statement is executed.
Fault Model Evaluation

**Goal**: To measure the accuracy of a fault model.

- Quantitative evaluation is essential for model development
  - Identify weaknesses in existing models
  - Build confidence in the use of a fault model
  - Avoid the fate of software test (use manufacturing test as a model)
Method for Fault Model Evaluation

• Compare *fault coverage* to *error coverage*
  ▪ Fault coverage is a fast approximation of error coverage
  ▪ Compute both fault and error coverage for many benchmarks and test sequences
  ▪ Examine the *difference* between the fault and error coverages and the *standard deviation of the difference*
Error Coverage Computation

• A design error model is needed to compute error coverage

• Design Error Model Requirements
  ▪ Must describe a well defined subset of real design errors
  ▪ Must be small enough to be tractable

• Error coverage is computed by injecting design errors and performing simulation
  ▪ Error is detected if the output of the correct behavior is different from the output of the erroneous behavior

Example: \(a = b \times c\) (correct) \(a = x \times c\) (incorrect)

• Error is not detected if \(c = 0\) or if statement is not executed
Design Error Model

• “Goof” errors - simple typographical mistakes
  ▪ Accounted for 12.7% of design errors found in the Pentium 4*

• Goof errors are described in research in *mutation analysis*

1. Arithmetic Operator Replacement - Each +, -, *, / is replaced by each other
2. Relational Operator Replacement - Each >, <, =, !=, … is replaced by each other
3. Variable Replacement - Each variable is replaced with each other variable of same type

* B. Bentley, “Validating the Intel Pentium 4 Microprocessor”, DAC 2001
Evaluation Experiments

• We evaluate statement coverage and branch coverage
• We use three small examples written in Java

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Statements</th>
<th>Branches</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCD</td>
<td>19</td>
<td>1</td>
<td>162</td>
</tr>
<tr>
<td>Diffeq</td>
<td>21</td>
<td>7</td>
<td>502</td>
</tr>
<tr>
<td>TLC</td>
<td>65</td>
<td>14</td>
<td>214</td>
</tr>
</tbody>
</table>

• 20 random test sequences are used
• Each sequence achieves 80% - 100% fault coverage
  • Only high coverage values are interesting
### Evaluation Result Summary

- Experiment demonstrates the type of information that can be gained from this evaluation technique
- Data is not sufficient to draw conclusions

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Statement Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average diff</td>
<td>St. Dev. diff</td>
</tr>
<tr>
<td><strong>GCD</strong></td>
<td>16.35</td>
<td>3.17</td>
</tr>
<tr>
<td><strong>Diffeq</strong></td>
<td>8.88</td>
<td>11.59</td>
</tr>
<tr>
<td><strong>TLC</strong></td>
<td>10.72</td>
<td>5.06</td>
</tr>
</tbody>
</table>

- St. dev. is more important than average
  - Low st. dev shows consistency/fidelity
- St. dev is roughly proportional to the number of errors
Development of Hardware/Software Fault Models

- **Must be formulated for a behavioral description**
  
  Manufacturing test has focused on logic level

- **Must consider both hardware and software features**

- **Hardware-Oriented Language Features:**
  
  Timing - signals vs. variables
  Concurrency - processes
**Static Faults vs. Timing Faults**

- **Static faults**
  - Independent of absolute event timing

- **Timing faults**
  - Depends on a specific timing of events

![Diagram showing differences between static and timing faults]

- **Correct Path**: $x = y + 1$
  - [$t_1$, $t_2$]

- **Static Fault**: $x = y + 2$
  - [$t_1$, $t_2$]

- **Timing Fault**: $x = y + 1$
  - [$t_1$, $t_2$]
Timing Fault Example

- Producer/Consumer with FIFO to allow rate mismatch
- Delay on “empty” signal impacts synchronization
**Mis-Timed Event (MTE) Fault**

- Signal refs are classified as either **Definition** or **Use**
- Two types of MTE faults can occur
  - \( \text{MTE}_{\text{early}} \) – definition occurs earlier than the correct time
  - \( \text{MTE}_{\text{late}} \) – definition occurs later than the correct time

**FIFO Description**

- **Def** -> empty <= 1

**Proc. Y Description**

- **Use** -> if (empty = 0) then 
  p := ReadFromFIFO() ;
MTE Fault Detection

• All definition-use (du) pairs must be executed during testing
  • MTE early faults are detected by du pairs
  • MTE late faults are detected by ud pairs

• Time difference between the definition and use must not exceed the error span threshold δ
  • Def and Use must be close in time so that a small time variation will reorder the def and use.
  • Magnitude of δ determines sensitivity to timing variation
Error Span Threshold $\delta$

- Determines the certainty that a def-use pair is reordered in the presence of a fault

  incorrect definition

  \begin{align*}
  &\text{Def} \\
  &\text{error span } \delta
  \end{align*}

- Large $\delta$: small change in def time may not reorder the def and use
  - Testing requirements are less stringent, high coverage

- Small $\delta$: small change in def time is more likely to reorder def and use
  - Testing requirements are more stringent, low coverage
## MTE Coverage, Experimental Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of stmt.</th>
<th># of signals</th>
<th># of pairs</th>
<th>MTE cov. *</th>
<th>Stmt. Cov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAL1</td>
<td>538</td>
<td>22</td>
<td>1732</td>
<td>0.1</td>
<td>0.70</td>
</tr>
<tr>
<td>switch</td>
<td>402</td>
<td>29</td>
<td>200</td>
<td>0.65</td>
<td>0.93</td>
</tr>
<tr>
<td>risc8</td>
<td>306</td>
<td>37</td>
<td>1032</td>
<td>0.54</td>
<td>0.60</td>
</tr>
<tr>
<td>DTMF</td>
<td>1257</td>
<td>17</td>
<td>262</td>
<td>0.39</td>
<td>0.54</td>
</tr>
</tbody>
</table>

- Industrial examples in Verilog
- Functional testbenches provided with each example

* Error span threshold is infinitely large. FC is maximized.
Impact of $\delta$ on MTE Coverage, Data Switch
Automatic Test Generation

- Create test sequences to detect MTE faults
- Formulate the problem as a constraint logic programming (CLP) problem
- Use a generic CLP solver to perform test generation
- Solver searches the space of all computations to identify one which detects each fault
Test Generation Process

Input: a behavioral system specification
Output: test sequence to detect a timing fault
CCG: Computation Constraints Generator
Computation Constraints describe system behavior
Co-design Finite State Machine: A system is defined as a network of CFSMs where each CFSM describes a concurrent process in the system. The CFSMs communicate via events on signals.

- An Example of CFSMs System: traffic light controller

Each edge is a cause-reaction pair.

- $\text{MTE}_{\text{early}}$ fault on *short signal: asserted while the highway is in the green state
Computation Model

- Each computation of the system must be described by the values of a set of integer variables
- Computation Variables:
  - **State Variable** contains the value of state for each CFSM $c$ at a given time step $t$. $SV_{highway, 0} = \text{"green"}$
  - **Edge Variable** the edge in each CFSM $c$ which is traversed at a given time step $t$. $EV_{highway, 1} = \text{"e1"}$
  - **Signal Variable** the value of signal at a given time step.
    - **trigger signal** *short_0 = 0
    - **value signal** havecar_2 = 1
State Constraints of CFSM Computation

A CFSM can be in state $s$ at time $t$ ($SV_{c,t} = s$) if one of the following statements is true:

- The CFSM is in state $s$ at time $t-1$ and the CFSM does not traverse an edge at time $t-1$;
- The CFSM is in a state $s_p$ at time $t-1$ and an edge from state $s_p$ to $s$ is traversed at time $t-1$.
Example of State Constraints

Example: state constraints of CFSM highway at time step 2,

\[(SV_{highway,2} = \text{“yellow”}) \rightarrow (SV_{highway,1} = \text{“yellow”}) \cap (EV_{highway,1} = \text{NULL}) \]
\[\cup (SV_{highway,1} = \text{“green”}) \cap (EV_{highway,1} = e1)\]

\[(SV_{highway,2} = \text{“red”}) \rightarrow (SV_{highway,1} = \text{“red”}) \cap (EV_{highway,1} = \text{NULL}) \]
\[\cup (SV_{highway,1} = \text{“yellow”}) \cap (EV_{highway,1} = e2)\]

\[(SV_{highway,2} = \text{“green”}) \rightarrow (SV_{highway,1} = \text{“green”}) \cap (EV_{highway,1} = \text{NULL}) \]
\[\cup (SV_{highway,1} = \text{“red”}) \cap (EV_{highway,1} = e3)\]
a CFSM will traverse an edge e if all of the following statements are:

- The CFSM is in state $s_p$ at time $t$, where $s_p$ is the predecessor state of edge e;
- All of the trigger conditions of edge e are satisfied at time $t$.

![Diagram showing state transition](image)

<table>
<thead>
<tr>
<th>state</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_p$</td>
<td>...</td>
</tr>
<tr>
<td>$s_p$</td>
<td>...</td>
</tr>
<tr>
<td>$s_p$</td>
<td>...</td>
</tr>
<tr>
<td>$s_p$</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>edge</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>...</td>
</tr>
<tr>
<td>e</td>
<td>...</td>
</tr>
<tr>
<td>e</td>
<td>...</td>
</tr>
<tr>
<td>e</td>
<td>...</td>
</tr>
</tbody>
</table>

*$x$*
Example of Edge Constraints

Example: edge constraints of CFSM highway at time 1,

\[
\begin{align*}
(EV_{\text{highway},1} = \text{“e1”}) &\rightarrow \\
& (SV_{\text{highway},1} = \text{“green”}) \cap (\text{car}_1 = 1) \cap (\text{havecar}_1 = 1) \quad (1) \\
(EV_{\text{highway},1} = \text{“e2”}) &\rightarrow \\
& (SV_{\text{highway},1} = \text{“yellow”}) \cap (\text{short}_1 = 1) \quad (2) \\
(EV_{\text{highway},1} = \text{“e3”}) &\rightarrow \\
& (SV_{\text{highway},1} = \text{“red”}) \cap (\text{long2}_1 = 1) \quad (3) \\
(EV_{\text{highway},1} = \text{“null”}) &\rightarrow \text{NOT1} \cap \text{NOT2} \cap \text{NOT3}
\end{align*}
\]
two types of signals exist in CFSM system

- **Trigger Signal**: such as *short
  
  \[ *tsig_t = 1 \text{ if at least one edge emitting this signal is traversed at time (t-1)}; \]
  
  \[ *tsig_t = 0 \text{ otherwise.} \]
Example of Trigger Signal Constraints

Example: trigger signal *short constraints of CFSM highway at time 3,

\[
\begin{align*}
\text{short}_3 &= 1 & \Rightarrow & \ (EV_{\text{timer},2} = \text{“e8”}) \cup (EV_{\text{timer},2} = \text{“e11”}) \\
\text{short}_3 &= 0 & \Rightarrow & \ (EV_{\text{timer},2} \neq \text{“e8”}) \cap (EV_{\text{timer},2} \neq \text{“e11”})
\end{align*}
\]
Formulation of Fault Detection Constraints

- **Fault Detection Constraints**: i.e. *short occurs early

<table>
<thead>
<tr>
<th>time step</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>highway state</td>
<td>-</td>
<td>green</td>
<td>… …</td>
</tr>
<tr>
<td>*short</td>
<td>-</td>
<td>1</td>
<td>… …</td>
</tr>
</tbody>
</table>

- **Equations**: \( SV_{highway,1} = green, \) short_1 = 1.
Test Generation Result, Traffic Light Controller

- **Fault:** *short occurs early when highway is green*
- **Fault Detection Constraints:** equations given earlier
- **Environment:**
  Machine P4, 2GHz CPU, 256MB MEM
  GNU-Prolog version 1.2.1
- **Performance:** 45 ms

<table>
<thead>
<tr>
<th>time step</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>highway state</td>
<td>-</td>
<td>green</td>
<td>-</td>
</tr>
<tr>
<td>*short</td>
<td>-</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>street state</td>
<td>red</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>highway state</td>
<td>green</td>
<td>green</td>
<td>yellow</td>
</tr>
<tr>
<td>signal</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>*short</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| *tick | 1 | 1 | 1 |
|*car | 0 | 1 | 0 |
| car | 0 | 1 | 1 |
Summary

• A method to evaluate the accuracy of a fault model

• A fault model for timing and synchronization errors

• A CLP-based test generation tool