Coverage at the Formal Specification Level

Rolf Drechsler, Julia Seiter, Mathias Soeken

Institute of Computer Science
University of Bremen, Germany
{drechsle,jseiter,msoeken}@informatik.uni-bremen.de

Cyber-Physical Systems
DFKI Bremen, Germany
{Rolf.Drechsler,Mathias.Soeken}@dfki.de
Outline

• Motivation
  – Coverage on the RTL
• Formal Specification Level
• Coverage Metrics for the FSL
• Algorithms
• Tool-Support: The USE-Plugin
• Evaluation
• Conclusion
Moore’s Law

If transistors were people

If the transistors in a microprocessor were represented by people, the following timeline gives an idea of the pace of Moore’s Law.

- **1970**: Intel 4004
- **1980**: Intel 286
- **1990**: Pentium III
- **2000**: Core i7 Extreme Edition
- **2011**: Population of China

Now imagine that those 1.3 billion people could fit onstage in the original music hall. That’s the scale of Moore’s Law.

Source: Intel
Verification Becomes Difficult

- Increasing complexity = increasing verification effort
- Correctness more and more important
- New approach: Completeness-driven design
Completeness of System Design

Are we done yet with verification? Coverage!
Completeness-driven Design

• Requires completeness for each design stage

• Completeness is measured in coverage

• No full coverage = don’t advance to the next stage
Coverage on the RTL

- Well-researched topic
- Lots of coverage metrics:
  - code coverage
  - functional coverage
  - state coverage
  - path coverage
Completeness in Property Checking

- Here: Proving completeness of properties in formal verification of **Counting Heads for Railways**

- Property specification using PSL

- Bounded Model Checking – SystemC-based verification flow

- Estimation of the functional coverage of properties
SystemC & PSL

- **SystemC – C++ class library**
  - Supports Hardware-/Software Co-Design
  - Modeling at different abstraction levels
  - Efficient simulation kernel
- **Property Specification Language (PSL)**
  - Temporal properties
  - If assumptions hold, commitments have to hold as well

```c
always (  
  // assumptions  
  x == 1  
) -> (  
  // commitments  
  next[2] (y == 2)  
);
```
Bounded Model Checking

- Sequential problem transformed to combinational problem by unrolling
- Observation window: $t$ to $t+c$
Functional Coverage

• Completeness of property set “ensured” manually

• Coverage estimation:
  1. Determine all properties reasoning over particular signal
  2. Prove using BMC
  3. Generate coverage property
Counting Head (CH)

- CH-inputs from double wheel detector
- CH-outputs connected to evaluation computer
- CH used to automatically determine the state of track vacancy detection sections
Counting Head (CH)

- Counting axles with double wheel detectors

- Impact on detectors determine state of the FSM
Formal Verification

System 1

System 2

both_systems_affected

system1_affected

unaffected

system2_affected
Formal Verification

property state_0 =
always (
    system1 == 0 && system2 == 0
    && reset == 0
) -> (next[1](state == 0));
property state_1 =
always (
    system1 == 1 && system2 == 0 && reset == 0
) -> (next[1](state == 1)) ;
Formal Verification

• Several properties needed:
  – for state system2_affected
  – for state both_systems_affected
  – for initial state
  – for stable state (no changes)
• Overall: 6 properties
  – Assumption: properties for state functional complete
→ Proof!!!
Verification Coverage in Formal

- Have I written enough properties?
- First approaches based on mutations (e.g. Hoskote, DAC’99)
  - Perform small design modification
  - Check if properties detect change
  - Too expensive
- Alternatives: Claessen, FMCAD’07; Bormann et al., DVCON’07; Große et al. TCAD-2008
  - Analyze whether each output is uniquely determined by properties
Formal Coverage Checking

Approach from Große et al. TCAD-2008

- Coverage problem as BMC problem
- Generate coverage property for each output $o$
  - If coverage property holds, properties uniquely determine $o$
  - If coverage property fails, counter-example corresponds to uncovered scenario
- More counter-examples can be generated using blocking clauses, but expensive:
  - Enumeration of all possible values for datapath variables
  - Huge effort to identify relevant signals of a scenario
Coverage Estimation

- \( \text{property state}_{-1} = \)
  - \( \text{always} ( \)
    - \( \text{system}_1 == 1 \&\& \text{system}_2 == 0 \)
    - \( \&\& \text{reset} == 0 \)
  - \( \Rightarrow ( \)
    - \( \text{next}[1](\text{state} == 1) \)
- \( \text{T} \)
- \( \text{D} \)
- Assume transformed properties

//state_1
((\text{system}_1 == 1 \&\& \text{system}_2 == 0 \&\& \text{reset} == 0)
 ? \text{next}[1](\text{state} == 1)
 : 1)
Coverage Estimation

- Insert multiplexor for signal

```c
// @insertMuxForSignal: state select
```

- Assume multiplexor selects original value (at all times except $t_{\text{max}}$)

```c
(select == 1)
```

→ Proof: multiplexor selects original value at $t_{\text{max}}$

```c
next[1](select == 1)
```
property coverage_state =
   // @insertMuxForSignal: state select
always (  
   // reset
   (reset == 1)
      ? next[1](state == 0)
         : 1)
   &&
   // state_0
   (system1 == 0 && system2 == 0 && reset == 0)
      ? next[1](state == 0)
         : 1)
   &&
   // state_1
   (system1 == 1 && system2 == 0 && reset == 0)
      ? next[1](state == 1)
         : 1)
   && ...
   // coverage transformations for the
   // two remaining states properties
   && (select == 1)
) -> (  
   next[1](select == 1)
);
Formal Verification

- state can be used to verify other signals:
  - Substate
    (8 properties)
  - Counters
    (State transitions)
    (17 properties)
  - Timeouts
    (30 properties)
## Formal Verification

<table>
<thead>
<tr>
<th>#</th>
<th>signal</th>
<th>#p</th>
<th>sec.</th>
<th>MB</th>
<th>#p</th>
<th>sec.</th>
<th>MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>state</td>
<td>6</td>
<td>64.19</td>
<td>381</td>
<td>1</td>
<td>11.97</td>
<td>381</td>
</tr>
<tr>
<td>2</td>
<td>state_old</td>
<td>8</td>
<td>86.89</td>
<td>381</td>
<td>1</td>
<td>12.01</td>
<td>381</td>
</tr>
<tr>
<td>3</td>
<td>state transitions</td>
<td>17</td>
<td>191.17</td>
<td>378</td>
<td>5</td>
<td>65.75</td>
<td>389</td>
</tr>
<tr>
<td>4</td>
<td>counters</td>
<td>15</td>
<td>156.12</td>
<td>320</td>
<td>5</td>
<td>76.37</td>
<td>442</td>
</tr>
<tr>
<td>5</td>
<td>timeouts &amp; evaluation</td>
<td>30</td>
<td>312.15</td>
<td>377</td>
<td>7</td>
<td>85.29</td>
<td>439</td>
</tr>
<tr>
<td>6</td>
<td>number of axles</td>
<td>10</td>
<td>118.10</td>
<td>380</td>
<td>2</td>
<td>24.48</td>
<td>382</td>
</tr>
<tr>
<td>7</td>
<td>number of errors</td>
<td>17</td>
<td>414.43</td>
<td>377</td>
<td>2</td>
<td>100.28</td>
<td>393</td>
</tr>
<tr>
<td>8</td>
<td>idle time</td>
<td>7</td>
<td>72.92</td>
<td>319</td>
<td>2</td>
<td>21.14</td>
<td>320</td>
</tr>
<tr>
<td>9</td>
<td>counter control token</td>
<td>20</td>
<td>211.47</td>
<td>377</td>
<td>5</td>
<td>85.72</td>
<td>516</td>
</tr>
<tr>
<td></td>
<td>overall</td>
<td>130</td>
<td>1627.44</td>
<td>381</td>
<td>30</td>
<td>483.01</td>
<td>516</td>
</tr>
</tbody>
</table>
Observation

- Verification of an axle counting system using BMC
- Properties proven to be complete
Coverage on the ESL

- Not quite as many results as on the RTL

- Several existing metrics:
  - code coverage
  - path coverage
  - transaction-based coverage
  - cover groups
Coverage on the FSL

- Only few metrics from software testing
- What about...
  - code coverage?
  - functional coverage?
  - further metrics?
- Clear definition lacking!
Design Flow

• **Formal Model** = UML/SysML

• **Introduce correctness as early as possible**

• **Check the formal model for various common problems**
The FSL Is Different

- System described as model
- Properties on the FSL:
  - Is a good state reachable?
  - Is an operation executable?
  - Is it possible to create a consistent instance?

Metrics from other levels are not applicable!
Typical Verification Tasks

- **Structural Problems:**
  - Does a state exist which...?

- **Behavioral Problems:**
  - Does a sequence of states exist which...?
Is an Operation Executable?

Processor
- pc: Integer
- instr: Integer
- prepareMemory()
- fetch()
- process()

Controller
- address: Integer
- dataout: Integer
- write(content)
- read()

Cell
- address: Integer
- content: Integer

Find a sequence of operations which contains process().
Operation Sequences

- Initial state = starting point
- Call n operations
- Operation call creates new system state
Find a Particular State

- Start from initial state
- Call up to k operations

Does one of these operation calls lead to the goal state?
Use a Particular Operation

• Start from initial state
• Call up to k operations

Is the goal operation used in this sequence?
Other Scenarios?

- Operation is executable:
  - first state satisfies pre-condition
  - second state satisfies all post-conditions

- But: Antecedent is not considered in two postconditions!
How Does This Help?

• Model as blueprint for implementation
  – dead constraints realized as dead code

• Operation sequences as test cases
  – dead constraints lead to irrelevant tests
Coverage Metrics

- Operation coverage
  - How many operations have been called?

- Subexpression coverage
  - How many subexpressions have evaluated to true?
Algorithm

• Given:
  – UML class diagram with OCL
  – Initial set of operation sequences
  – Initial state

• Goal: New operation sequences which
  – ...contain operations not covered
  – ...trigger subexpression which do not hold yet
Increase Operation Coverage

- Find all not-covered operations
- Create for each an operation sequence which contains the operation
- Check which other operations are now covered by that sequence
Increase Subexpression Coverage

- Find all not-covered subexpressions

- Add each subexpression as post-condition to a copy of the original operation

- Create an operation sequence with this operation
Example: Operation Coverage

- Not covered operations: `read()`
  `write(content)`
- Create new operation sequences
Example: Operation Coverage

- Covers both `read()` and `write(content)`
- Leads to full operation coverage
Example: Subexpression Coverage

- Only one post-condition is triggered
- Create new operations with missing subexpressions as post-conditions
- Check each for executability
Example: Subexpression Coverage

- One new operation for each subexpression
- Contains all other pre- and post-conditions
Example: Subexpression Coverage

- Find a sequence containing `process_1()`
- Sequence is found

Expression is covered
Example: Subexpression Coverage

- Find a sequence containing process_1()
- Sequence is not found
  Expression is not covered
Implementation

• Implemented as a plugin for USE

• USE:
  – UML specification environment
  – supports UML and OCL
  – tool for visualization, evaluation of OCL constraints, consistency checks, ...
USE-Plugin

- Report of initial coverage

- Three categories:
  - Covered
  - Not covered
  - Maybe covered
USE-Plugin

- Report of maximal coverage
- Three categories:
  - Covered
  - Not covered
  - Partially covered
Evaluation

• Applied to several models (provided by USE and self-written)

• Focus: increase of coverage
  – Comparison between initial and maximal coverage
Evaluation

![Bar chart showing Initial Coverage and Maximal Coverage for different categories: CPU, Traffic, Memory, Car, Life.](chart.png)
Evaluation

- Increase of coverage for all models but one

- CPU: choice of initial state influences initial and maximal coverage

- TrafficLight, Memory: no full coverage possible
Conclusion

• Coverage metrics for formal specifications
  – Operation coverage
  – Subexpression coverage

• Avoid dead code and irrelevant test cases

• First steps for completeness-driven design on the FSL
Coverage at the Formal Specification Level

Rolf Drechsler, Julia Seiter, Mathias Soeken

Institute of Computer Science
University of Bremen, Germany
{drechsle,jseiter,msoeken}@informatik.uni-bremen.de

Cyber-Physical Systems
DFKI Bremen, Germany
{Rolf.Drechsler,Mathias.Soeken}@dfki.de