Model-Based Automatic Test Generation for Event-Driven Embedded Systems Using Model Checkers

Zoltan Micskei and Istvan Majzik
{micskeiz, majzik}@mit.bme.hu

DepCos-RELCOMEX ‘06

2006. 05. 27.
Testing concepts

- **Goal:** improving the quality of the system
- **Test case:** input events and *expected output actions* representing a typical paths
- **Test suite:** set of test cases
- **Test requirement:** a specific sub goal for testing, e.g. call a function
- **Coverage criterion:** determines a set of test requirements, e.g. cover all statements
Problems in testing

- Test cases are written manually
  - Time and resource consuming
- If the specification/code changes all the test cases should be modified
- Detailed **behavioural model** can help
  - Parts of the testing tasks can be automated
Testing framework

System model
Criterion
Implementation

Test Generation
Abstract Test cases

Test Transformation
Concrete Test cases

Test Execution
Test results, coverage
Model-Based Automatic Test Generation for Event-Driven Embedded Systems Using Model Checkers

Testing framework

- System model
- Criterion
- Test Generation
- Abstract Test cases
- Test Transformation
- Concrete Test cases
- Test Execution
- Test results, coverage

Implementation

- Teszteles
- formula
Test generation using a model checker

- Engineering model
- Mathematical model
- Model checker
- Test cases
- Testing criterion
- TL formulae
Test generation using a model checker

System model is transformed into an input language of a model checker tool

- Engineering model
- Mathematical model
- TL formulae
- Test cases
- Testing criterion
- Model
Test generation using a model checker

Coverage criteria are expressed using temporal logic formulae, e.g. all state should be covered.
Test generation using a model checker

To get traces that satisfy these formulae, the negated formulae are checked in a model checker.
Test generation using a model checker

The counter-examples are in fact a test suite satisfying the original testing criterion.
Implemented test generator

- UML Statechart
- Promela code
- SPIN model checker
- XML test case
- Coverage criterion
- LTL formulae
Implementation details

- Model transformation
  - UML $\rightarrow$ *Extended Hierarchical Automata* $\rightarrow$ Promela

- Test generation input
  - **Model**: UML Statechart representing the behaviour
  - **Criterion**: coverage criterion on the model
    - *Currently*: All or selected states/transition covered, custom temporal logic formulae

- Output
  - **Events** collected from the SPIN detailed trace
Efficiency of test generation

- Classical model checking: exhaustive verification of the full state space
- Test generation: finding a minimal length counter-example quickly
  → special configuration is necessary.
- Measurements:
  - Effects of SPIN’s 10 parameters on runtime
  - Goal: minimize time needed for test generation while test suite is minimal in length
Analyzed SPIN parameters

- **-dBFS**: Breadth First Search
- **-m**: depth limit in Depth First Search
- **-i** and **–l**: minimal counter-example iteratively
- **-w**: size of the hash table to store states

Other parameters (e.g. **–dNOFAIR**, **-dSAFETY**) did not have significant effect
Experiment 1: Mobile model

- Statechart describing behaviour of a mobile phone
- 10 states, 21 transitions
### Detailed results

<table>
<thead>
<tr>
<th>Options</th>
<th>Duration</th>
<th>Sum length</th>
<th>Shortest test</th>
</tr>
</thead>
<tbody>
<tr>
<td>-i</td>
<td>22m 32.46s</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>-dBFS</td>
<td>11m 48.83s</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>-i -m1000</td>
<td>4m 47.23s</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>-I</td>
<td>2m 48.78s</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>default</td>
<td>2m 04.86s</td>
<td>385</td>
<td>94</td>
</tr>
<tr>
<td>-I -m1000</td>
<td>1m 46.64s</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>-m1000</td>
<td>1m 25.48s</td>
<td>97</td>
<td>16</td>
</tr>
<tr>
<td>-m200 -w24</td>
<td>46.7s</td>
<td>17</td>
<td>3</td>
</tr>
</tbody>
</table>
## Detailed results

<table>
<thead>
<tr>
<th>Options</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>-i</td>
<td>22m 32.46s</td>
</tr>
<tr>
<td>-dBFS</td>
<td>11m 48.83s</td>
</tr>
<tr>
<td>-i -m1000</td>
<td>4m 47.23s</td>
</tr>
<tr>
<td>-l</td>
<td>2m 48.78s</td>
</tr>
<tr>
<td>default</td>
<td>2m 04.86s</td>
</tr>
<tr>
<td>-l -m1000</td>
<td>1m 46.04s</td>
</tr>
<tr>
<td>-m1000</td>
<td>1m 25.48s</td>
</tr>
<tr>
<td>-m200 -w24</td>
<td>46.7s</td>
</tr>
</tbody>
</table>

Iterative search: short test cases but long duration

BFS: good results, but ran out of memory in more complex models

Limiting the depth resulted always in shorter execution

Limiting the depth and setting the hash table was the optimal setting
Real-life case study

- Industrial partner’s synchronization protocol
- 5 objects, 31 states, 174 transitions
- $2 \times 10^8$ states visited
- Further techniques needed:
  - State space compression – bit-state hashing
  - Reduction of the model in a conservative manner
  - Leaving out requirements already covered
- Finding minimal length/size test suite is NP-complete
Synchronisation protocol - results

Synchronisation protocol - all state covered

<table>
<thead>
<tr>
<th>Execution time [min]</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>150</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>50</td>
</tr>
</tbody>
</table>

- Default (depth limit and hash-table)
- Model reduction (channels)
- Leaving out already covered (65%)
Testing framework
Testing implementations

- Use the **abstract test cases** to test conformance of an implementation
- Definition of a **test interface** (mapping)
- Conducted experiments
  - Mobile model, with two implementation:
    - manually coded using nested switch method
      - Statechart Java code generator
  - Two test execution framework:
    - JUnit, Rational Robot
Code coverage results

Test cases contained only valid events. Testing of implicit transitions are needed.
Conclusion

- Tool chain for **automatic test generation** of event-driven systems
- Multiple **coverage criteria** on the model
- Optimizing model checker for test generation
- Applying tests to implementation
- **Code coverage** on implementation
- Real-time systems: generating also timing information