Practical Verification of OSEK/VDX Operating System

Toshiaki Aoki
School of Information Science/
Research Center for Software Verification
JAIST
Motivation

- Recently, the safety and reliability of automotive systems are becoming a big concern of our society.
  - Functional safety standards (IEC61508, ISO26262)
  - Unintended acceleration problem of Toyota's cars (suspect on electronic throttle control systems).

- Automotive companies are interested in formal methods to improve the safety and reliability of automotive systems.
  - Conforming to the functional safety standards.
  - Effectively and efficiently increasing the safety and reliability.

- We are working on practically applying formal methods to automotive systems.
  - Mainly focus on automotive operating systems conforming to OSEK/VDX (shortly, OSEK OS).
  - The operating systems are considered as important components in the evaluation of the safety of the automotive systems.
Objective

- Providing a high quality OS by applying formal methods.
  - Target: RX-OSEK850
    - developed by Renesas Micro Systems (RMS)
    - already used in current series of cars.

- Joint research project.
  - JAIST and DENSO since 2006.
    - DENSO is a user of RX-OSEK850.
  - RMS joined in 2009.
    - RMS is developing RX-OSEK850.
Approach

• Firstly, we focus on design verification and testing.
  • Design verification: the design of RX-OSEK850 is verified by a model checking tool Spin.
  • Testing: we automatically generate test cases from the verified design.
In the design verification, we verify tasks and ISRs performed according to their specification.

We model a scheduler of RX-OSEK850.

The design model is constructed by JAIST, DENSO and Renesas.

- The design of RX-OSEK850 is described and checked by JAIST.
- The design and checking results are reviewed by DENSO and Renesas.
Design of RX-OSEK850

- We focus on the scheduler of RX-OSEK850.
  - The most of functionalities provided by OSEK OS is to schedule tasks and ISRs.
  - The scheduler computes a task or ISR to be dispatched based on some data such as task queue, TCB and tables.
- It is easy to describe the scheduler in Promela.
  - Those data and computation can be straightforwardly described in Promela.
Design Verification

• OSEK OS is an open system.
  • OSEK OS does scheduling of the tasks if it gets stimulus such as system call invocations.
  • OSEK OS does not do anything if it does not get any stimulus.

• We need the description of the outside of OSEK OS to verify the design model.
  • The outside consists of multi-task applications which invoke the system calls and hardware which causes the interrupts.

• The outside of the verification target is called an environment.
Environment

- The description of an environment consists of
  - invocations of the system calls of OSEK OS.
    - represented as state transition models with the system calls.
  - expected results of those invocations.
    - represented as assertions assigned to those states.
- Model checking the design model in combination with the description of the environment by Spin.

RTOS design Environment (2 Tasks, Task 1<Task 2)

invokes

info
Environment Modeling

- Various environments can be considered.
  - The number of tasks, the variation of priorities, number of the resources, invocation relations of the system calls, and so on.
  - It is impossible to make all the descriptions of the environments manually.
  - If all the environments are realized by one Promela description, that may cause state explosion problem.
- We model the variations of the environments, then automatically generate the Promela descriptions of the environments.
Environment Modeling

- We model possible structures of the environments using class diagram.
  - The structural variations of the environments are represented as multiplicities of associations among classes.
  - Constraints on attribute values and multiplicities are described in OCL (Object Constraint Language).
- Collective behavior of environments are described by Statechart with some extensions.

```
inv
self.res->forall(r|pr=<r.pr)

Task
pr : {1..P}
states {Sus,Rdy,Run}
tsk:(1..M)
res:(0..N)

Resource
pr : {1..Q}
states {Fre,Occ(n:1..M)}

inv
GetState(self.id)==Rdy

Sus
inv
GetState(self.id)==Sus

Run
inv
GetState(self.id)==Run
```
We propose a method to automatically generate environments from the environment model. A Promela description of an environment is generated for each of the variations represented by the environment model.

Steps of the generation.
1. Generate all the object graphs in a given bound from the class model of the environment model using a SMT solver.
2. Make the state transition models of each of the object graphs based on the statechart model of the environment model.
3. Translate the state transition models into Promela descriptions.
Design Verification

- We constructed multiple environment models for multi-viewpoint analysis.
  - Separation of concerns of the verification.
    - task lifecycles, event synchronization, multiple activation, and so on.
  - Making the environment model simple and confident.
    - It should be more confident than the design model in an engineering sense.

- Environment generation.
  - Total:786, Total Time: 81.4s, 0.1s per an environment

- Model checking.
  - Total Time: 8819.3s, 11s for each of them.
  - Almost of all of the time is for compiling pan.c.

<table>
<thead>
<tr>
<th>No.</th>
<th>Environment Model</th>
<th>Generated Environments</th>
<th>model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>number</td>
<td>time(s)</td>
</tr>
<tr>
<td>1</td>
<td>TaskDiff</td>
<td>26</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>TaskEq</td>
<td>14</td>
<td>0.4</td>
</tr>
<tr>
<td>3</td>
<td>CtDiff</td>
<td>26</td>
<td>0.6</td>
</tr>
<tr>
<td>4</td>
<td>CtEq</td>
<td>14</td>
<td>0.5</td>
</tr>
<tr>
<td>5</td>
<td>MultDiff</td>
<td>26</td>
<td>0.7</td>
</tr>
<tr>
<td>6</td>
<td>MultEq</td>
<td>14</td>
<td>1.6</td>
</tr>
<tr>
<td>7</td>
<td>ResDiff</td>
<td>248</td>
<td>38.3</td>
</tr>
<tr>
<td>8</td>
<td>ResEq</td>
<td>98</td>
<td>15.4</td>
</tr>
<tr>
<td>9</td>
<td>EvDiff</td>
<td>26</td>
<td>1.2</td>
</tr>
<tr>
<td>10</td>
<td>EvEq</td>
<td>14</td>
<td>2.3</td>
</tr>
<tr>
<td>11</td>
<td>IsrDiff</td>
<td>182</td>
<td>10.3</td>
</tr>
<tr>
<td>12</td>
<td>IsrEq</td>
<td>98</td>
<td>9.6</td>
</tr>
</tbody>
</table>
Result (Design Verification)

- It takes around 6 months to construct, review and verify the design model.
  - Not full-time. All the members have their own works.
  - We have a meeting once a month for reviewing the design model and verification result.
- The design model was verified in the JAIST side.
  - We found many errors such as wrong conditions and wrong updates of data.
    - Those are not the errors of RX-OSEK850 but the design model.
    - We did not find the errors of RX-OSEK850 itself in the design verification.
- We recognized that the specification of OSEK/VDX is very ambiguous.
  - 5 misunderstandings of the specification are pointed out in the review.
  - Those misunderstandings come from the ambiguity of the specification.
    - We could not identify what behavior of OS is correct based only on the OSEK/VDX specification.
    - We have to refer to the specification and behavior of RX-OSEK850.
Approach

- Firstly, we focus on design verification and testing.
  - Design verification: the design of RX-OSEK850 is verified by a model checking tool Spin.
  - Testing: we automatically generate test cases from the verified design.
Testing

- We adopted testing to check RX-OSEK850 implementation.
  - RX-OSEK850 is already implemented.
  - Automata Theoretic Conformance Testing.
- Design Model = Test Oracle
  - We put the assumption that the design model is correct.
    - We made much effort to check the design model by model checking.
- We extract test cases and expected results from the design model.
  - Exhaustive test cases are automatically generated by search of Spin model checker.
  - Expected results are computed in the design model.

**State transition model (Promela)**

```
state transition model(Promela)

s1 -> s2
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
```

**Test tree**

```
test tree

s1 -> s2
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
```

**Test cases**

```
test cases

TC={ab,cd}
```
TCG/TPG

- Automatic test case generation by Spin.
  - Promela/Spin + embedded C codes + log analysis
  - TCG: generating test cases from the search log of Spin.
  - TPG: generating test programs from the test cases.
- We need a test model (an environment for testing).
  - Invocation of system services.
  - Specifying bounds of search.

![Diagram of TCG/TPG process]

Promela
- Test Model
- Design Model

Test Case Generator (TCG)
- Test Tree
- Scan Test Tree

Log analysis

Search Log
1: Down ..... 2: Down ...
107: Up ....

Test Program Generator (TPG)
- Transfoming to Programs
- Test Programs

if (....){
  ....
  }
  ....
The test model nondeterministically invokes system calls with some bounds.

- Pre-condition of the system calls.
- Configurations of system components such as task and ISRs.

We determined the minimum configuration to check the implementation by analyzing RX-OSEK 850.

- Tasks: 3, Resources: 2, ISR: 1, Event: 1
- It covers variations of preemptions of tasks and ISRs.

```
do ::precondition1 -> syscall1 ::precondition2 -> syscall2 .... od
```

Invoking system calls

acquiring expected results
**Execution of Testing**

- Total number of test cases: 742,748

<table>
<thead>
<tr>
<th>Priorities</th>
<th>Priorities</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskA</td>
<td>1</td>
</tr>
<tr>
<td>TaskB</td>
<td>2</td>
</tr>
<tr>
<td>TaskC</td>
<td>3</td>
</tr>
<tr>
<td>ResourceA</td>
<td>1 2 3 2 1</td>
</tr>
<tr>
<td>ResourceB</td>
<td>2 3 3 3 2 1</td>
</tr>
<tr>
<td>#Test cases</td>
<td>12483 15077 26373 37127 25035 8495 26489 26489 66361 66361 66361 13301</td>
</tr>
<tr>
<td>pan exe. time</td>
<td>12.9 16.8 26.0 38.7 26.7 10.7 27.6 30.6 66.3 66.9 68.3 14.6</td>
</tr>
<tr>
<td>TCG exe. time</td>
<td>19.0 24.9 67.9 73.1 58.5 12.7 81.8 93.7 289.2 287.2 282.6 27.2</td>
</tr>
<tr>
<td>TPG exe. time</td>
<td>176.8 174.4 508.5 522.8 433.9 95.0 548.4 626.9 2267.4 2694.1 2692.2 232.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priorities</th>
<th>Priorities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task A</td>
<td>1</td>
</tr>
<tr>
<td>TaskB</td>
<td>1</td>
</tr>
<tr>
<td>TaskC</td>
<td>2</td>
</tr>
<tr>
<td>ResourceA</td>
<td>1 2 3 2 1</td>
</tr>
<tr>
<td>ResourceB</td>
<td>2 3 3 3 2 1</td>
</tr>
<tr>
<td>#Test cases</td>
<td>17151 22427 44723 60707 39457 10331 13011 20707 33117 56281 25459 9425</td>
</tr>
<tr>
<td>pan exe. time</td>
<td>19.0 22.7 45.0 60.5 40.2 11.5 13.7 21.9 33.4 55.6 25.5 9.9</td>
</tr>
<tr>
<td>TCG exe. time</td>
<td>35.6 44.7 117.6 179.7 99.4 16.8 21.8 38.8 78.3 161.8 55.1 14.1</td>
</tr>
<tr>
<td>TPG exe. time</td>
<td>290.7 320.2 799.8 1353.5 694.4 138.9 175.2 317.9 546.8 1486.5 442.8 125.3</td>
</tr>
</tbody>
</table>
Testing Results

- The execution of testing is fully automated by TCG and TPG.
- It can be done in parallel, but, the number of simulator licenses is limited to 3.
  - The licenses are occupied by engineers of Renesas daytime.
  - We executed testing in parallel nighttime and weekends.
- It takes around 3 months to complete the testing.
  - We could not major precise time to complete the testing due to our ad-hoc parallelization of testing.
  - Samples)
    - It takes 169.75 hours for 26,489 test cases (Total compile time: 42.5 hours, execution: 127.25 hours)
    - It takes 265 hours for 44,723 test cases (compile time: 80.25 hours, execution: 184.75)
  - The estimation of total time of testing based on those samples is around 4535 hours, that is, 189 days.
    - We succeeded in reducing the testing time by the parallelization.
- Surprisingly, all the test cases have been passed at a time!
  - All the execution sequences of the design model have been accepted by the implementation.
  - We did not need any modification of the design model during the testing.
Whole view of our approach
Whole view of our approach

OSEK/VDX Specification

RX-OSEK850 Specification

Design Verification

Environment Model

EnvGen

Environment (Promela)

Spin

Design Model (Promela)

Test Model (Promela)

Spin

TCG/TPG

Test Case Generation

Testing

RX(Binary)

Test Programs

Simulator
Discussion

- No error of RX-OSEK850 has been found.
  - RX-OSEK850 is developing more than 10 years including similar past operating systems.
  - It is already used in the current series of cars.
    - Some bugs have been found and fixed in developing those cars.
- We succeeded in making the design model whose quality is similar to RX-OSEK850 only for 6 months.
  - All the execution sequences of the design model are equivalent to those of the implementation.
    - We expected before the testing that if testing was failed, it should be due to the bugs of the design model.
    - All the test cases (742,748) have been passed with no backtrack to the design phase.
Ongoing Works

- Making a formal specification of OSEK/VDX in formal specification languages.
  - We found that many ambiguities are involved in the OSEK/VDX specification.
  - The specification of AUTOSAR OS is much worse than OSEK/VDX.
  - We should describe the specification formally using formal specification languages.
    - We are using Event-B and CafeOBJ.
- On the other hand, model checking is suitable for design of OS.
  - Timing analysis is important.
  - It is easy to describe the design in an imperative specification language like Promela.
- We are studying the formalization of RTOS documents and conformance verification between the specification and design
  - Our setting:
    - Specification is described in Event-B.
    - Design is described in Promela.

```
OSEK/VDX Specification in Event-B

Sched
when
  ts = RUN
  prio1 > prio2
  ...
then
  ct := new
  ....

Design in Promela

proctype Sched(){
  if
    :: prio1 > prio2 ->
    ...
  fi;
  enqTask(tt);
  ....
```

model checking with some bounds
Conclusion

- We are practically applying formal methods to the verification of an automotive operating system.
  - Target: RX-OSEK850
  - Combining model checking, testing and formal specification techniques.
- Currently, many modeling elements such (ex. #task, #queue) are bounded due to the limitation of model checking and testing.
  - Challenge: extending our approach to universally ensure properties using induction and theorem proving so that it can be practically applied.

References in English:


References in Japanese:
