Model Checking for Practical C software Analysis – Experience Reports

Moonzoo Kim
Provable Software Lab. CS Dept. KAIST
Http://pswlab.kaist.ac.kr
Prelude
Model Checking for Practical C Software Analysis – Experience Reports

Professional Cook

Symbolic M.C.
Explicit M.C.
Exhaustive Testing
Bounded M.C.
Concolic Testing
Randomized Testing

Knife Master

SI software
Embedded software
Financial software
Web software
Mathematical software
End-user software

Moonzoo Kim et al.
Provable SW Lab. CS Dept.
• The series of empirical studies on verification of Samsung OneNAND™ flash memory FTL through various off-the-shelf techniques
  – Symbolic MC, Explicit MC, SAT-based MC, Exhaustive testing, randomized testing and concolic testing
Contents

• Overview on Multi-sector Read Operation (MSR)
  – Flash Translation Layer (FTL) scheme
  – MSR algorithm

• Model Checking MSR
  – Reports on the following three aspects
    • Target system modeling
    • Environment modeling
    • Performance analysis on the verification

• Three different types of model checkers are used
  – BDD based symbolic model checking (NuSMV)
  – Explicit model checking (Spin)
  – C-bounded model checking (CBMC)

• Exhaustive testing and concolic testing is applied as well
PART I: MSR Overview

- FTL basics
- Example of logical data distribution on physical unit
- Exponential increase of possible distributions
- MSR structure
• In flash memory, logical data are distributed over physical sectors.
Examples of Possible Data Distribution

- **Assumptions**
  - there are 5 physical units
  - each unit has 4 sectors
  - each sector is 1 byte long
Exponential Increase of Distribution Cases

\[ \sum_{i=1}^{n-1} \left( \binom{4}{i} 4! \right) \times \left( \binom{(n-i)}{3} (n-4)! \right) \]
01: curLU = LU0;
02: while (numScts > 0) {
    Loop1: iterates over LUs
    03:     readScts = # of sectors to read in the current LU
    04:     while (readScts > 0) {
        Loop2: iterates until the current LU is read completely
        05:         curPU = LU->firstPU;
        06:         while (curPU != NULL) {
            Loop3: iterates over PUs linked to the current LU
            07:             while (...) {
                Loop4: identify consecutive PS’s in the current PU
                08:                 conScts = # of consecutive PS’s to read in curPU
                09:                 offset = the starting offset of these consecutive PS’s in curPU
                10:                     }
                11:                     BML_READ(curPU, offset, conScts);
                12:                     readScts = readScts - conScts;
                13:                     curPU = curPU->next;
                14:                     }
                15:             }
            16:     }
        17: }
    18: }
19: curLU = curLU->next;
PART II: Model Checking Exp.

• Verification of MSR by using NuSMV, Spin, and CBMC
  – NuSMV: BDD-based symbolic model checker
  – Spin: Explicit model checker
  – CBMC: C-bounded model checker

• The requirement property is to check
  – after_MSR -> (\forall i. \text{logical\_sectors}[i] == \text{buf}[i])

• We compared these three model checkers empirically
Verification by NuSMV

NuSMV was the first choice as a verification tool, since:

1. BDD-based symbolic model checkers have been known to handle large state spaces
2. MSR operates with a semi-random environment (i.e. all possible configurations of PUs and SAMs analyzed)
3. Data structure of MSR can be abstracted in a simple array form with assignments and equality checking operations only
4. MSR is a single-threaded software
Target Model Creation in NuSMV

- We had to introduce control points variables, since
  - C is control-flow based
  - NuSMV modeling language is dataflow-based
- Linked list is replaced by an array operation.
  - Array index variables should be statically expanded, since NuSMV does not support index variables
- As a result, the final NuSMV model is more than 1000 lines long

<table>
<thead>
<tr>
<th>A fragment of C</th>
<th>Conversion to parallel statements based on control and data dependency</th>
<th>Corresponding NuSMV code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: x=x-1;</td>
<td>0: DP1=0; DP2=0; DP3=0;</td>
<td>init(DP1):=0; init(DP2):=0; init(DP3):=0;</td>
</tr>
<tr>
<td>2: while(x&gt;=0){</td>
<td>1: if (!DP1) { x=x-1; DP1 =1;}</td>
<td>next(DP1):= 1;</td>
</tr>
<tr>
<td>3: y = x;</td>
<td>2: if ((DP1</td>
<td></td>
</tr>
<tr>
<td>4: x --;}</td>
<td>y = x; DP2=1; DP3=0;</td>
<td>DP2</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>0;</td>
</tr>
<tr>
<td></td>
<td>3: if (DP2) {</td>
<td>1 : DP2;</td>
</tr>
<tr>
<td></td>
<td>x--; DP3=1; DP2=0;</td>
<td>esac;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>next(DP3):= case (DP1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DP2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 : 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>esac;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next(x):= case !DP1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 : x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>esac;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next(y):= case (DP1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 : y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>esac;</td>
</tr>
</tbody>
</table>
Modeling in NuSMV (2/2)

- **Environment model creation**
  
  - The environment of MSR (i.e., PUs and SAMs configurations) can be described by *invariant rules*. Some of them are
    
    1. One PU is mapped to at most one LU
    2. *Valid correspondence between SAMs and PUs*:
       
       If the $i$th LS is written in the $k$th sector of the $j$th PU, then the $i$th offset of the $j$th SAM is valid and indicates the $k$th PS. 
       
       Ex: $3^{rd}$ LS (‘C’) is in the $3^{rd}$ sector of the $2^{nd}$ PU, then $SAM1[2] == 2$
       
       $i=3$  $k=3$  $j=2$

    3. *For one LS, there exists only one PS that contains the value of the LS*:
       
       The PS number of the $i$th LS must be written in only one of the $(i \mod 4)$th offsets of the SAM tables for the PUs mapped to the corresponding LU.

\[
\forall i,j,k \ (LS[i] = PU[j].sect[k] \rightarrow (SAM[j].valid[i \mod m] = true \\
& SAM[j].offset[i \mod m] = k \\
& \forall p. (SAM[p].valid[i \mod m] = false) \\
& \text{where } p \neq j \text{ and } PU[p] \text{ is mapped to } [(\frac{i}{m} \text{th LU})])
\]
Verification Performance of NuSMV

- Verification was performed on the machine equipped with Xeon5160 (3Ghz, 32Gbyte Memory), 64 bit Fedora Linux 7, NuSMV 2.4.3
- The requirement property was proved correct for all the experiments (i.e., MSR is correct in this small model)
  - For 7 sectors long data that are distributed over 7 PUs consumes more than 11 hours while consuming 550 mb memory
Performance Analysis

- The MSR model (5 LS’s and 5 PUs) has 365 BDD variables for its symbolic representation
  - At least 240 BDD variables are required for PUs and SAMs
    - 5 (# of PUs) x 4 (sectors/PU) x 2 (current/next) x 3 (bits)
- The same MSR model generated 1.2 million BDD nodes.
- Dynamic reordering takes more than 90% of total verification time
  - Time is the bottleneck in this NuSMV verification task
Modeling by Spin

- **A target model**
  - Translated from the MSR C code through Modex which is an automated C-to-Promela translator with embedded C statements
    - Modex translates MSR into the same 4 level-nested loop control structure
- **An environment model**
  - PUs and SAMs, which takes most of memory, are tracked, but not stored in the state vector through a **data abstraction technique**
    - `c_track` keyword and **Unmatched** parameter
    - Based on the observation that SAMs and PUs are sparse
    - Only a unique **signature** of the current state of PUs and SAMs is stored succinctly
      - `<(0,1),(1,1),(1,2),(2,3),(3,0),(4,1)>` is the signature of the following PUs and SAMs configuration

\[
\begin{array}{c|c|c|c}
\text{Sector 0} & \text{SAM0} & \text{PU0} \\
\hline
1 & 0 & \text{E} \\
2 & 1 & \text{A} \\
3 & \text{B} & \text{F} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{Sector 1} & \text{SAM1} & \text{PU1} \\
\hline
1 & 1 & \text{B} \\
2 & \text{C} & \text{F} \\
3 & \text{D} & \text{E} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{Sector 2} & \text{SAM2} & \text{PU2} \\
\hline
1 & \text{A} & \text{F} \\
2 & \text{C} & \text{E} \\
3 & \text{D} & \text{A} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{Sector 3} & \text{SAM3} & \text{PU3} \\
\hline
1 & \text{B} & \text{E} \\
2 & \text{D} & \text{F} \\
3 & \text{A} & \text{C} \\
\end{array}
\]
• The requirement property was satisfied
• The data abstraction technique shows significant performance improvement upto 78% of memory reduction and 35% time reduction (for 5 logical sectors data)
CBMC does not require an explicit target model creation.

An environment for MSR was specified using *assume statements* and the environment model was similar to the environment model in NuSMV.

For the loop bounds, we can get valid upper bounds from the loop structure and the environment setting:

- The outermost loop: \( L \) times (\( L \) is a # of LUs)
- The 2\(^{nd} \) outermost loop: 4 times (one LU contains 4 LS’s)
- The 3\(^{rd} \) outermost loop: \( M \) times (\( M \) is a # of PUs)
- The innermost loop: 4 times (one PU contains 4 PS’s)
Verification Performance of CBMC

- Exponential increase in both time and memory. However, the slope is much lower than those of NuSMV and Spin, which makes CBMC perform better for large problems.
- A problem of 10 PUs and 8 LS’s has $8.6 \times 10^5$ variables and $2.9 \times 10^6$ clauses.
Performance Comparison

Time complexity $LS = 6$

Space complexity $LS = 6$

A number of physical units

Seconds

Megabytes
Comparison of Model Checking Techniques

- Application of Model Checking to Industrial SW Project
  - Current off-the-shelf model checkers showed their effectiveness to debug a part of industrial software, if a target portion is carefully selected.
  - Although model checker worked on a small scale problem, it still contributes due to its exhaustive exploration which is complementary to the testing result.

- Comparison among the Three Model Checkers

<table>
<thead>
<tr>
<th>Model</th>
<th>Modeling Difficulty</th>
<th>Memory Usage</th>
<th>Verification Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>NuSMV</td>
<td>Most difficult</td>
<td>Good</td>
<td>Slow</td>
</tr>
<tr>
<td>Spin</td>
<td>Medium difficult</td>
<td>Poor</td>
<td>Fast</td>
</tr>
<tr>
<td>CBMC</td>
<td>Easiest</td>
<td>Best</td>
<td>Fastest</td>
</tr>
</tbody>
</table>
Part III: Experiments on Testing MSR
Exhaustive Testing

- Exhaustive testing on a small flash
  - We developed a testing environment and an abstracted version of MSR(), called S_MSR()
  - The reuse of formal environment models reduces the testbed setup time
  - Exhaustive testing is roughly **6 times faster** than CBMC
Randomized Testing

• Randomized testing on a large flash
  – Model checking and exhaustive testing cannot handle a large flash
    • We cannot find a bug on a large flash
  – We performed randomized testing on $10^{11}$ randomly chosen cases with 6 sectors long data distributed over 1000 PUs
    • This takes 8 hours 20 minutes
  – This test cover only $\frac{1}{3.9 \times 10^{11}}$ cases among all possible cases
  – 1GB flash has more than $2^{19}$ (a half million) physical units
    • $2^{19}$ units * 4 sector/unit * 512 bytes/sector

=> Randomized testing cannot provide sufficient coverage ever
Concolic (CONCrete + symbOLIC) Testing

- Automated Scalable Unit Testing of real-world C Programs
  - Execute unit under test on automatically generated test inputs so that all possible execution paths are explored
    - Explicit path model checking
- In a nutshell
  - Use concrete execution over a concrete input to guide symbolic execution
    - A symbolic path formula is obtained at the end of an execution
  - One branch condition of the path formula is negated to generate the next execution path
  - The next execution path formula is solved by SMT solver to generate concrete input values, and so on
  - No false positives or scalability issue like in model checking
We have to specify symbolic variables and put constraints on them.

- If assigned input value does not satisfy the constraints (i.e., invalid test case generated), a current iteration terminates immediately without testing MSR (goto out);
Result with Constraint-based Model (1/2)

(a) Total number of test cases generated

(b) Ratio of valid test cases/all test cases
Result with Constraint-based Model (2/2)

(a) Total analysis time

(b) Time ratio of analysis steps

Model Checking for Practical C software Analysis – Experience Reports

Moonzoo Kim et al
Provable SW Lab. CS Dept.
Explicit Environment Model

- Explicit environment model creates invalid test cases much less than the constraint-based model.
- CREST has a limitation on array index variable.

```c
01: for (i=0; i< NUM_LS; i++) {
02:     unsigned char idxPU, idxSect;
03:     CREST_unsigned_char(idxPU);
04:     CREST_unsigned_char(idxSect);
05: ...
06: // The switch statements encode the following two statements:
07: // PU[idxPu].sect[idxSect]= LS[i];
08: // SAM[idxPu].sect[i]= idxSect;
09: switch(idxPU){
10:   case 0: switch(idxSect) {
11:       case 0: PU[0].sect[0] = LS[i];
12:           SAM[0].offset[i] = idxSect; break;
14:           SAM[0].offset[i] = idxSect; break;
15:     ... }
16:   break;
17: case 1: switch(idxSect) {
```

Moonzoo Kim et al
Provable SW Lab. CS Dept.
Result with Explicit Model (1/2)

(a) Total number of test cases generated

- 4 PUs w/ 5 LSes
- 4 PUs w/ 6 LSes
- 5 PUs w/ 5 LSes
- 5 PUs w/ 6 LSes

(b) Ratio of valid test cases/all test cases

- 4 PUs w/ 5 LSes
- 4 PUs w/ 6 LSes
- 5 PUs w/ 5 LSes
- 5 PUs w/ 6 LSes
Result with Explicit Model (2/2)

(a) Total analysis time

(b) Time ratio of analysis steps
Overall Observations

- There are multiple **useful off-the-shelf analysis tool** to improve the reliability of target C programs in practice
  - Knowing characteristics of them and underlying mechanisms is essential for successful analysis
- **Systematic heuristics techniques for searching “XXX”** space are important
  - Good tradeoff between completeness and effectiveness
- **Abstract environment modeling** is very important for in-depth target system analysis
  - This area still largely relies on human expertise
Lessons Learned

• Necessity of Benchmarks for the purpose of SW analysis
  – To encourage comparative studies of various analysis methods

• Importance of target application selection
  – Several restrictions from industrial partner
  – Open source target application

• SMT techniques have large rooms for improvement
  – Pos: You can join the competition now !!!
  – Cons: You may better to use other analysis engine, at least in a few years