SAT-based Model Checking for Debugging Embedded Software

Moonzoo Kim
Provable Software Lab, CS Dept, KAIST
Research Theme

• **SW reliability**
  – Quality attribute for minimizing malfunctions of systems to reduce damage to human life or valuable properties

• **Highly reliable SW technology is a key to the success of industrial products**
  – The portion of SW in embedded devices increases continuously
Software Development Cycle

- A practical end-to-end formal framework for software development

A SW Development Framework for SW with High Assurance

- Requirement analysis
- System design
- Design analysis
- Implementation
- Testing
- Monitoring

- Formal requirement Spec.
- Formal system modeling
- Model analysis/verification
- Model-assisted code generation
- Model-based testing
- Runtime monitoring and checking
Unified Formal Verification Framework

- Unified formal framework of the following three approaches can make synergy
  - **Model Verification**
    - Targets a system model
    - Req. spec is limited
    - Complete coverage
  - **Code verification**
    - Targets a real code
    - Extracts an abstract system model from a real code
    - Req. spec is limited
  - **Runtime Verification**
    - Targets a real code
    - Verifies correctness of current execution run
    - Req. spec can be very expressive
Research Approach

• Practical formal methods that can be applied to software intensive systems to enhance reliability
Overview of the Case Study

• In 2007, Samsung requested to debug the device driver for the OneNAND™ flash memory
• We reviewed the requirement specifications, the design documents, and C code to identify code-level properties to check.
• Then, we applied several model checkers including CBMC (C Bounded Model Checker) to check the properties
  – Found several bugs
  – Provided high confidence in multi-sector read operation through exhaustive exploration
Overview of the OneNAND® Flash Memory

- Characteristics of OneNAND® flash
  - Each memory cell can be written limited number of times only
    - Logical-to-physical sector mapping
    - Bad block management
    - Wear-leveling
  - XIP by emulating NOR interface through demand-paging scheme
    - Multiple processes access the device concurrently
    - Urgent read operation should have a higher priority
    - Synchronization among processes is crucial
  - Performance enhancement
    - Multi-sector read/write
    - Asynchronous operations
    - Deferred operation result check

Source: Software Center of Samsung Electronics '06
Logical to Physical Sector Mapping

1:N mapping from a LUN to PUNs

- In flash memory, logical data are distributed over physical sectors.
Overview of SAT-based Bounded Model Checking

- **C Program**
- **Requirement Properties in C assertion** $\phi$
- **Model Checker**
- **Abstract Model** $M$
- **SAT Solver**
  - $\neg \phi \land P$
- **Translation to the SAT formula**
- **Counter example**
  - Unsatisfied
  - Okay $P \models \phi$
  - Satisfiable
  - Counter example
- **Okay** $M \models \phi$
C Program to SAT Translation (1/2)

- **Unwinding a loop**

<table>
<thead>
<tr>
<th>Original code</th>
<th>Unwinding the loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=0; while (x &lt; 2) {</td>
<td>x=0;</td>
</tr>
<tr>
<td>y=y+x;</td>
<td>if (x &lt; 2) {</td>
</tr>
<tr>
<td>x++;</td>
<td>y=y+x;</td>
</tr>
<tr>
<td>}</td>
<td>x++;</td>
</tr>
<tr>
<td></td>
<td>} if (x &lt; 2) {</td>
</tr>
<tr>
<td></td>
<td>y=y+x;</td>
</tr>
<tr>
<td></td>
<td>x++;</td>
</tr>
<tr>
<td></td>
<td>} assert ( ! (x &lt; 2))</td>
</tr>
</tbody>
</table>

- **From C code to SAT formula**

<table>
<thead>
<tr>
<th>Original code</th>
<th>Convert to static single assignment (SSA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=x+y; if (x!=1) x=2;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>x++;</td>
</tr>
<tr>
<td>assert (x&lt;=3);</td>
<td></td>
</tr>
<tr>
<td>x=x+y; if (x!=1) x=2;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>x++;</td>
</tr>
<tr>
<td>assert (x&lt;=3);</td>
<td></td>
</tr>
</tbody>
</table>

- **Generate constraints**

  \[
P \equiv x_1 = x_0 + y_0 \land x_2 = 2 \land x_3 = x_1 + 1 \land ( (x_1 = 1 \land x_4 = x_2) \lor (x_1 = 1 \land x_4 = x_3) )
  \]

  \[
  \phi \equiv x_4 \leq 3
  \]

  Check if \( P \land \neg \phi \) is satisfiable, if it is then the assertion is violated
C Program to SAT Translation (2/2)

• Example of arithmetic encoding into pure propositional formula

Assume that \( x, y, z \) are three bits positive integers represented by propositions \( x_0x_1x_2, y_0y_1y_2, z_0z_1z_2 \)

\[
C \equiv z = x + y \equiv (z_0 \leftrightarrow (x_0 \oplus y_0) \oplus (x_1 \land y_1) \lor (((x_1 \oplus y_1) \land (x_2 \land y_2))) \\
\land (z_1 \leftrightarrow (x_1 \oplus y_1) \oplus (x_2 \land y_2)) \\
\land (z_2 \leftrightarrow (x_2 \oplus y_2))
\]

Half adder circuit diagram

Full adder circuit diagram
Inputs: \{A, B, CarryIn\} → Outputs: \{Sum, CarryOut\}
C Bounded Model Checker (CBMC)

- Handles function calls using **inlining**
- Unwinds the loops a **fixed number of times** (bounded MC)
  - A user has to know a **upper bound** of each loop
    - Loops often have clear upper bounds
    - We can still get debugging result without upper bounds
- Specifies **constraints** to describe an **environment** of the target program, which can model non-deterministic user inputs, or multiple scenarios
  - Ex. __CPROVER assume(0<=nDev && nDev<=7)
  - Ex. __CPROVER_assume( SHDC.nPhySctsPerUnit == SHPC.nBlksPerUnit * SHVC.nPgsPerBlk * SHVC.nSctsPerPg)
- Checks properties by assertions
Overview of the Case Study

• The goal of the project
  – To check whether USP conforms to the given high-level requirements
    • we needed to identify the code-level properties to check from the given high-level requirements
• A top-down approach to identify the code level properties from high-level requirements
  – USP has a set of elaborated design documents
    • Software requirement specification (SRS)
    • Architecture design specification (ADS)
    • Detailed design specification (DDS)
      – DPM, STL, BML, and LLD
Three High-level Requirements in SRS

- SRS specifies 13 functional requirements, 3 of which have “very high” priorities
  - Support prioritized read operation
    - To minimize the fault latency, USP should serve a read request from DPM prior to generic requests from a file system.
    - This prioritized read request can preempt a generic I/O operation and the preempted operation can be resumed later.
  - Concurrency handling
    - BML and LLD should avoid a race condition or deadlock through synchronization mechanisms such as semaphores and locks.
  - Manage sectors
    - STL provides logical-to-physical mapping, i.e. multiple logical sectors written over the distributed physical sectors should be read back correctly.
Top-down Approach to Identify Code-level Property

- Total 43 code-level properties are identified

**Legend**
- Spec. in the design docs
- User defined property to check

**Sequence Diagram**
A sequence diagram of page fault handling while a device is being programmed in LLD DDS
Results of Unit Testings

- Prioritized read operation
  - Detected a bug of not saving the status of suspended erase operation

- Concurrency handling
  - Confirmed that the BML semaphore was used correctly
  - Detected a bug of ignoring BML semaphore exceptions

- Multi-sector read operation (MSR)
  - Provided high assurance on the correctness of MSR, since no violation was detected even after exhaustive analysis (at least with a small number of physical units (~10))
A Bug in PriRead()

374: VOID PriRead(Read(UINT32 nDev, UINT32 nPbn, UINT32 nPgOffset) {
  ...
416:    if ((bEraseCmd==FALSE32) && (pstInfo->bNeedToSave==TRUE32)) {
417:        pstInfo->nSavedStatus = GET_ONLD_CTRL_STAT(pstReg, ALL_STATE);
418:        pstInfo->bNeedToSave = FALSE32;
419:          saved=1;  // added for verification purpose  }
  ...
424:    assert(!(pstInfo->bNeedToSave) || saved);

• We added a flag saved to denote whether the status of the preempted operation is saved
• CBMC detected the given assertion was violated when an erase operation was preempted
  – It takes 8 seconds and 325 Mb on the 3Ghz Xeon machine
  – CBMC 2.6 with MiniSAT 1.1.4

01:....
02:State 14 file LLD.c line 408 function PriRead thread 0
03: LLD::PriRead::1::bEraseCmd=1
04:State 15 file LLD.c line 412 function PriRead thread 0
05: LLD::PriRead::1::1::2::nWaitingTimeOut=...
06:State 17 file LLD.c line 412 function PriRead thread 0
07: LLD::PriRead::1::1::2::nWaitingTimeOut=...
08:....
09:Violated property:
10: file LLD.c line 424 function PriRead
11: assertion !(_Bool)pstInfo->bNeedToSave || (_Bool)saved
12:VERIFICATION FAILED
BML Semaphore Usage

• The standard requirements for a binary semaphore
  – Semaphore acquire should be followed by a semaphore release
  – Every function should return with a semaphore released
    • unless the semaphore operation creates an exception error.

• There exist 14 BML functions that use the BML semaphore.
  – We inserted an \texttt{smp} to indicate the status of the semaphore
  – and simple codes to decrease/increase \texttt{smp} at the corresponding semaphore operation.

• CBMC concluded that all 14 BML functions satisfied the above two properties.
  – Consumes 10 seconds and 300 megabytes of memory on average to analyze each BML function
BML Semaphore Exception Handling (1/2)

- The BML semaphore operation might cause an exception depending on the hardware status.
- Once such BML semaphore exception occurs, that exception should be propagated to the topmost STL functions to reset the file system
  - We checked this property by the following assert statement inserted before the return statement of the topmost STL functions:
  - `assert(!(SMerr==1)| |nErr==STL CRITICAL ERR)"
CBMC analyzed a call graph of each of the topmost STL functions and detected that BML semaphore exception might not propagate due to bug at `_GetSInfo()`.

The bug was detected when loop bound was set 2 with ignoring loop unwinding assertion.
- Memory overflow occurred with the loop bound 3

For STL_Write(), this verification task consumed 616 megabytes of memory in 97 seconds
- Each call sequence is around 1000 lines long on average.
Multi-sector Read Operation (MSR) (1/2)

- MSR reads adjacent multiple physical sectors once in order to improve read speed
  - MSR is 157 lines long, but highly complex due to its 4 level loops
- We built a small test environment for MSR
  - The test environment contains only up to 10 physical units
  - The test environment should follow constraints, which are described by _CPROVER_assume(Boolean exp) statement
    - SAM tables and PUs should correspond each other
    - For each logical sector, at least one physical sector that has the same value exists
Modeling in NuSMV (2/2)

• The test environment should follow constraints, which are described by
  \_CPROVER\_assume(Boolean exp) statement
  – SAM tables and PUs should correspond each other
• 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\textsuperscript{rd} LS ('C') is in the 3\textsuperscript{rd} sector of the 2\textsuperscript{nd} PU, then \( \text{SAM1}[2] \equiv 2 \)
     \( i=3 \quad k=3 \quad j=2 \)
  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.

\[
\begin{array}{cccc}
\text{SAM0~SAM4} & \text{PU0~PU4} \\
\hline
\text{Sector 0} & 1 & 0 & \text{E} \\
\text{Sector 1} & 1 & 1 & \text{AB} \\
\text{Sector 2} & 2 & & \text{C} \\
\text{Sector 3} & 3 & & \text{D} \\
\end{array}
\]
Multi-sector Read Operation (MSR) (2/2)

- We checked MSR for data that was 5~8 sectors long and distributed over 5~10 PUs.
  - CBMC analyzed all possible scenarios/distributions in this environment
- CBMC detected no violation of the property (read buffer should contain correct data) in this series of experiments with small flash memory.
  - Each of the experiments consumed 280 to 700 megabytes of memory
- More details of this verification task, see “Formal Verification of a Flash Memory Device Driver -an Experience Report” published at Spin ’08
Performance Comparison

Time complexity \( LS = 6 \)

Space complexity \( LS = 6 \)

A number of physical units

Seconds

Megabytes

A number of physical units
Promising Research Topics (1/3)

• Requirement property derivation is a crucial starting activity in model checking, but often neglected
  – No systematic study yet, to my knowledge
    • Close relation to requirement engineering
• Environment modeling as well as target modeling is a crucial issue for industrial success of model checking
  – Garbage in, garbage out
  – No automation yet
  – No significant research activities yet
Promising Research Topics (2/3)

• Practical application of SAT-based model checking for program verification
  – Bit-level accuracy is a big advantage!!!
    • Less restriction and limitation compared to CEGAR approach
    • We can avoid many misleading results due to abstraction
  – SMT is a new challenger, but
    • SMT has overwhelming restrictions (e.g. linear arithmetic, requirement of loop invariants, etc)
    • Performance of SMT is not significantly better than that of SAT
      – Decision procedures in most SMT theories are based on SAT.
      – SAT solvers possess industrial strength through 50 years’s research
Promising Research Topics (3/3)

• Clear limitation of model checking
  – The result of model checking can guarantee the correctness of MSR only for a small environment

• Natural subsequent approach => Theorem proving
  – No automation aimed (at least by me ;)), but an intellectually challenging task
  – For a specific domain, such as MSR in flash memory device driver, one pattern of logical specification can be reused and may give back reward to the investment
Conclusion

• We successfully applied CBMC to detect hidden bugs in the device driver for Samsung’s OneNAND flash memory
  – Also, we established confidence in the correctness of the complex MSR

• Lessons learned
  – Software model checker as an effective unit testing tool
    • CBMC took modest amount of memory and time to detect bugs in USP
    • Exhaustive analysis can detect hidden bugs
  – Advantages of a SAT-based model checker
    • Analysis capability of whole ANSI-C
    • No abstract model required

• We believe that a SAT-based model checker can be utilized effectively as a unit testing tool to complement conventional testing