Software Model Checking via Automatic Test Generation

Patrice Godefroid
Microsoft Research

"Model Checking"

- Model Checking (MC) = systematic state-space exploration = exhaustive testing
- "Model Checking" = "check whether the system satisfies a temporal-logic formula"
  - Example: G(p→Fq) is an LTL formula
- Simple yet effective technique for finding bugs in high-level hardware and software designs (examples: FormalCheck for Hardware, SPIN for Software, etc.)
- Once thoroughly checked, models can be compiled and used as the core of the implementation (examples: SDL, VFSM, etc.)

Model Checking of Software

- How to apply model checking to analyze software?
  - “Real” programming languages (e.g., C, C++, Java),
  - “Real” size (e.g., 100,000’s lines of code).
- Two main approaches to software model checking:

<table>
<thead>
<tr>
<th>Modeling languages</th>
<th>state-space exploration</th>
<th>Model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstraction</td>
<td>(SLAM, Bandera, ForVer, BLAST, CBMC,...)</td>
<td>adaptation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>state-space exploration</th>
<th>Systematic testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>VeriSoft, JPF, CMC, Buger, CHESS,...</td>
<td></td>
</tr>
<tr>
<td>Data input:</td>
<td>DART, EXE, BAG,...</td>
<td></td>
</tr>
</tbody>
</table>

Overview: Software MC via Systematic Testing

- Lecture 1: SMC via Systematic Testing - Concurrency
- Lecture 2: SMC via Systematic Testing - Data Inputs
- Disclaimer:
  - emphasis on what influenced the speaker, not an exhaustive survey

Dynamic Approach: Systematic Testing (VeriSoft)

- State Space = “product of (OS) processes” (Dynamic Semantics)
  - Processes communicate by executing operations on com. objects.
  - Operations on com. objects are visible, other operations are invisible.
  - Only executions of visible operations may be blocking.
  - The system is in a global state when the next operation of each process is visible.
  - State Space = set of global states + transitions between these.

THEOREM: Deadlocks and assertion violations are preserved in the "state space" as defined above.
VeriSoft

- Controls and observes the execution of concurrent processes of the system under test by intercepting system calls (communication, assertion violations, etc.).
- Systematically drives the system along all the paths (=scenarios) in its state space (=automatically generate, execute and evaluate many scenarios).
- From a given initial state, one can always guarantee a complete coverage of the state space up to some depth.
- Note: analyzes “closed systems”; requires test driver(s) possibly using “VS_toss(n)”.

VeriSoft State-Space Search

- Automatically searches for:
  - deadlocks,
  - assertion violations,
  - divergences (a process does not communicate with the rest of the system during more than x seconds),
  - livelocks (a process is blocked during x successive transitions).
- A scenario (=path in state space) is reported for each error found.
- Scenarios can be replayed interactively using the VeriSoft simulator (driving existing debuggers).

The VeriSoft Simulator

Originality of VeriSoft

- VeriSoft is the first systematic state-space exploration tool for concurrent systems composed of processes executing arbitrary code (e.g., C, C++, …) [POPL97].
- VeriSoft looks simple! Why wasn’t this done before?
- Previously existing state-space exploration tools:
  - restricted to the analysis of models of software systems;
  - each state is represented by a unique identifier;
  - visited states are saved in memory (hash-table, BDD,…).
- With programming languages, states are much more complex!
- Computing and storing a unique identifier for every state is unrealistic!

“State-Less” Search

- Don’t store visited states in memory: still terminates when state space is finite and acyclic… but terribly inefficient!
- Example: dining philosophers (toy example)
  - For 4 philosophers, a state-less search explores 386,816 transitions, instead of 708: every transition is executed on average 546 times!

Partial-Order Reduction in Model Checking

- A state-less search in the state space of a concurrent system can be much more efficient when using “partial-order methods”.
- POR algorithms dynamically prune the state space of a concurrent system by eliminating unnecessary interleavings while preserving specific correctness properties (deadlocks, assertion violations,…).
- Two main core POR techniques:
  - Persistent/stubborn sets (Valmari, Godefroid,…)
  - Sleep sets (Godefroid,…)
[ Note: checking more elaborate properties require other extensions
  - Ex: ample sets (Peled) are persistent sets satisfying additional conditions sufficient for LTL model checking
  - Not used here as VeriSoft only checks reachability properties ]
Persistent/Stubborn Sets

• Intuitively, a set \( T \) of enabled transitions in \( s \) are persistent in \( s \) if whatever one does from \( s \) while remaining outside of \( T \) does not interact with \( T \).

• Example:
  \( (q_1 \text{ is empty in } s) \)
  \[ P_1: \text{Send}(q_1, m_1) \]
  \( (P_1: \text{Send}(q_1, m_1)) \) is persistent in \( s \)

  \[ \text{The most advanced algorithms for (statically) computing persistent sets are based on “stubborn sets” [Valmari]} \]

• Limitation: need info on (static) system structure.
  • VeriSoft only exploits info on next transitions and “system_file.VS”.

Sleep Sets

• Sleep Sets exploit local independence (commutativity) among enabled transitions. One sleep set is associated with each state.

• Example:
  \[ P_1: \text{Send}(q_1, m_1) \]
  \[ P_2: \text{Send}(q_1, m_2) \]
  \[ \text{Transitions in Sleep are not explored?} \]

  \[ (q_1 \text{ is empty in } s) \]
  \[ P_1: \text{Send}(q_1, m_1) \]
  \[ P_2: \text{Send}(q_1, m_2) \]

  \[ \text{Limitation: alone, no state reduction.} \]
  • Sleep sets are easy to implement in VeriSoft since they only require information on next transitions.

An Efficient State-Less Search

• With POR algorithms, the pruned state space looks like a tree!

• Thus, no need to store intermediate states!

• Without POR algorithms, a state-less search in the state space of a concurrent system is untractable.

VeriSoft - Summary

• Two key features distinguish VeriSoft from other model checkers
  − Does not require the use of any specific modeling/programming language.
  − Performs a state-less search.

• Use of partial-order reduction is key in presence of concurrency.

• In practice, the search is typically incomplete.

• From a given initial state, VeriSoft can always guarantee a complete coverage of the state space up to some depth.

Users and Applications

• Development of research prototype started in 1996.

• VeriSoft 2.0 available outside Lucent since January 1999:
  − 100’s of licenses in 25+ countries, in industry and academia
  − Free download at http://www.bell-labs.com/projects/verisoft

• Examples of applications in Lucent:
  − 4ESS HBM unit testing and debugging (telephone switch maintenance)
  − WaveStar 40G R4 integration testing (optical network management)
  − 7R/E PTS Feature Server unit and integration testing (voice/data signaling)
  − CDMA Cell-Site Call Processing Library testing (wireless call processing)

Application: 4ESS HBM [ISSTA98]

• 4ESS switches control millions of calls every day.

• Heart Beat Monitor (HBM) determines the status of elements connected to 4ESS switch by monitoring propagation delays of messages to/from these elements.

• November 1996: “field incident”; June 1997: 2nd field incident…

• HBM code = 100s of lines of EPL (assembly) code, 7/3 years old

• Hoe does this code work exactly???
Application: 4ESS HBM (continued)

- Translate EPL code to C code (using existing partial translator)
- Build test harness for HBM C code, model its environment (using “VS_toss(n)”), add “VS_assert(0)” where HBM code hits NTH (took only a few hours!)
- Check properties (reverse eng./testing)
- Discovered several flaws in software and its documentation... [ISSTA98]

Example of scenario found:

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario A</td>
<td>Passed</td>
</tr>
<tr>
<td>Scenario B</td>
<td>Failed</td>
</tr>
</tbody>
</table>

Discussion: Strengths of VeriSoft

- Used properly, very effective at finding bugs
  - can quickly reveal behaviors virtually impossible to detect using conventional testing techniques (due to lack of controllability and observability)
  - compared with conventional model checkers, no need to model the application!
    - Eliminates this time-consuming and error-prone step
    - VeriSoft is WYSIWYG: great for reverse-engineering
- Versatile: language independence is a key strength in practice
- Scalable: applicable to very large systems, although incomplete
  - the amount of nondeterminism visible to VeriSoft can be reduced at the cost of completeness and reproducibility (not limited by code size)

Discussion: Limitations of VeriSoft

- Requires test automation:
  - need to run and evaluate tests automatically (can be nontrivial)
  - if test automation is already available, getting started is easy
- Need be integrated in testing/execution environment
  - minimally, need to intercept VS_toss and VS_assert
  - intercepting/handling communication system calls can be tricky...
- Requires test drivers/environment models (like most MC)
- Specifying properties: the more, the better... (like MC)
  - Restricted to safety properties (ok in practice); use Purify!
- State explosion... (like MC)

Discussion: Conclusions

- VeriSoft (like model checking) is not a panacea.
  - Limited by the state-explosion problem,...
  - Requires some training and effort (to write test drivers, properties, etc.).
  - “Model Checking is a push-button technology” is a myth!
- Used properly, VeriSoft is very effective at finding bugs.
  - Concurrent/reactive/real-time systems are hard to design, develop and test.
  - Traditional testing is not adequate.
  - “Model checking” (systematic testing) can rather easily expose new bugs.
- These bugs would otherwise be found by the customer!
- So the real question is “How much ($) do you care about bugs?”

Software Model Checking Tools

<table>
<thead>
<tr>
<th>Year</th>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>VeriSoft (Bell Labs)</td>
<td>CBMC (CMU)</td>
</tr>
<tr>
<td>1995</td>
<td>JavaPathFinder (NASA)</td>
<td>Bandera (Kansas U.)</td>
</tr>
<tr>
<td>2000</td>
<td>CMC (Stanford)</td>
<td>BLAST (Bell Labs)</td>
</tr>
<tr>
<td>2005</td>
<td>And many other recent ones...</td>
<td></td>
</tr>
</tbody>
</table>
Static Partial-Order Reduction

• Use static analysis to predict locations red accesses after $S$
  – if static analysis proves that red thread only accesses $y$ and $z$
  – then $x := 1$ is a persistent transition from $S$

Example: static partial-order reduction

Global Vars

- lock $m$
- int $t1,t2$
- int $x=0$
- int $m=100$
- char[] $a$

Thread 1:
- lock($m$)
- $t1 := x++$
- unlock($m$)
- for( ;$t1<n$; $t1+=2$)
  - $a[t1] := 'b'$

Thread 2:
- lock($m$)
- $t2 := x++$
- unlock($m$)
- for( ;$t2<n$; $t2+=2$)
  - $a[t2] := 'r'$

New Idea: Dynamic Partial-Order Reduction

• Execute initial arbitrary execution trace to completion
• Examine transitions performed by each thread
  – identify and explore other interleavings that may behave differently
  – dynamic alias analysis is easy

Dynamic partial-order reduction algorithm

• Dynamic POR algorithm for
  – safety properties (deadlocks, assertion violations, etc.)
  – acyclic state spaces
• Dynamically computes a persistent set in each explored state
  – compatible and complementary with sleep sets
• Complexity: $O(m^2 r)$
  – $m =$ number of threads
  – $r =$ size of reduced state space
  – some assumptions on dependence relation
• See [POPL’05, with Cormac Flanagan]
**Filesystem Benchmark**

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Number of transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>100000</td>
<td>100000</td>
</tr>
<tr>
<td>1000000</td>
<td>1000000</td>
</tr>
<tr>
<td>10000000</td>
<td>10000000</td>
</tr>
</tbody>
</table>

**CHESS (MSR): Preemption Bounding**

- Focus on multi-threaded concurrent software (Win32, CLR)
- Focus on executions with small number of preemptions
  - Heuristic: most bugs can be found with a small number of preemptions
  - Many bugs found this way; CHESS is available on the web
    - Can deal with very large state spaces, complementary to (D)POR

**Other Related Work: Heuristics**

- Other heuristics for partially exploring large state spaces
  - Genetic algorithms (with property-specific fitness functions)
  - Heuristics based on concurrent dependencies
    
    
    ```
    x = 1;
    if (p != 0)
    {
      x = p + 1;
    }
    ```

    If dereference before initialization, BUG!
    Thus, ONE ordering constraint is sufficient for this bug
    - “Principled random searches” (e.g., see Cuzz from MSR)

**Conclusion**

- Software Model Checking via Systematic Testing
  - Lecture 1: Dealing with Concurrency
    
    ![Diagram](image)

    Modeling languages  
    Modeling languages  
    Programming languages  
    Programming languages  
    Systematic testing  
    Systematic testing  
    Data inputs  
    Data inputs