Formal Verification Projects at Microsoft Research

Nikolaj Bjørner
Microsoft Research
Projects with and by: Leonardo de Moura, Nikolai Tillmann, Peli de Halleux, Mike Barnett, Rustan Leino, Ernie Cohen, Wolfram Schulte, Michael Levin, Patrice Godefroid
Outline

• Research in Formal Verification
• Selected projects
  – Pex, SAGE
  – Spec#, Boogie, VCC, Hypervisor
  – Z3
A few Projects

• **Static Driver Verifier** (Ball, Rajamani ...)
  – Symbolic model checking for Windows drivers.
  – Well established. Shipped with the Windows driver verification tools.

• **Software Model-checking** (Muthuvaththi, Nori, Qadeer, ...)
  – Yogi: concrete execution + symbolic partition refinement.
  – Chess: multi-threaded programs

• **Test Generation** (Godefroid, Halleux, Lahiri, Levin, Tillmann, Zhang ...)
  – Use decision procedures to generate test inputs.

• **Program Verification** (Barnett, Cohen, Leino, Schulte, ...)
  – Spec#/Boogie: programming with contracts.
  – The Viridian Hypervisor: Comprehensive functional correctness.

• **Termination, Separation, Heaps** (Berdine, Cook, Lahiri, Qadeer, ...)

• **Model Based Testing** (Grieskamp, Veanes, ...)
  – SpecExplorer, MUnit: Use models to extract test cases.

• **Decision Procedures** (Bordeaux, Hamadi, de Moura, Zhang, N.B., ...)
  – The plumbers
A few proof-carrying researchers
Unit Tests
Automatic Generation of Unit Tests

- **Test** (correctness + usability) is 95% of the deal:
  - Dev/Test is 1-1 in products.
  - Developers are responsible for unit tests.

- Basic memory bugs are very costly:
  - Especially to customers.

- **Tools:**
  - Annotations and static analysis (SAL, ESP)
  - File Fuzzing
  - Unit test case generation.
    - Use program analysis tools, automated theorem proving
Security is Critical

• Security bugs can be very expensive:
  – Cost of each MS Security Bulletin: $600K to $Millions [MS Treasury Group]
  – Cost due to worms (Slammer, CodeRed, Blaster, etc.): $Billions
  – The real victim is the customer.

• Most security exploits are initiated via files or packets
  – Ex: Internet Explorer parses dozens of file formats

• Security testing: hunting for million-dollar bugs
  – Write A/V (always exploitable), Read A/V (sometimes exploitable), NULL-pointer dereference, division-by-zero (harder to exploit but still DOS attacks), etc.
Hunting for Security Bugs

- Two main techniques used by “black hats”:
  - Code inspection (of binaries) and Blackbox fuzz testing

- **Blackbox** fuzz testing:
  - A form of blackbox random testing
  - Randomly *fuzz* (=modify) a well-formed input
  - Grammar-based fuzzing: rules that encode how to fuzz

- **Heavily** used in security testing
  - At MS: various internal tools
  - Conceptually simple yet effective in practice...
    - Has been instrumental in weeding out 1000’s of bugs during development and test
Challenge: Automatic Code-Driven Test Generation

**Given** a sequential program with a set of input parameters,

**Generate** a set of inputs that maximizes code coverage

Solve: $z = x + y \land z > x - y$  
Solve: $z = x + y \land z \leq x - y$

<table>
<thead>
<tr>
<th>Input x, y</th>
<th>x = 1, y = 2</th>
<th>x = -2, y = -3</th>
</tr>
</thead>
<tbody>
<tr>
<td>z := x + y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z &gt; x-y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Method: Dynamic Test Generation

Run program with random inputs.

Gather constraints on inputs.

Use constraint solver to generate new inputs.

Combination with randomization: DART
Godefroid-Klarlund-Sen-05,...

Input $x, y$

$z := x + y$

$z == \text{hash}(x)$

Return $z$

error();

Can’t statically generate values for $x$ and $y$ that satisfy “$z==\text{hash}(x)$” (or “$x = \text{complex_os_library}(y)$”)

But constraint solver finds it 😊
DARTish projects at Microsoft

- **SAGE** (CSE) implements DART for x86 binaries and merges it with “fuzz” testing for finding security bugs
- **PEX** (MSR-Redmond FSE group) implements DART for .NET binaries in conjunction with “parameterized-unit tests” for unit testing of .NET programs
- **YOGI** (MSR India) implements DART to check the feasibility of program paths generated statically using a SLAM-like tool
- **Vigilante** (MSR Cambridge) partially implements DART to dynamically generate worm filters
Initial Experiences with SAGE

25+ security bugs and counting...
(most missed by blackbox fuzzers)

• OS component X
  – 4 new bugs: “This was an area that we heavily fuzz tested in Vista”

• OS component Y
  – Arithmetic/stack overflow in y.dll

• Media format A
  – Arithmetic overflow; DOS crash in previously patched component

• Media format B & C
  – Hard-to-reproduce uninitialized-variable bug
Pex

- **Pex** monitors the execution of .NET application using the [CLR profiling API](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.clrprofilingapi).  
- **Pex** dynamically checks for violations of certain programming rules, e.g. whether some resources were not released.  
- **Pex** will suggest code snippets to the user, which will prevent the same failure from happening again.
using System;
using Microsoft.Pex.Framework;

namespace ChunkerDemo {
    public class Chunker {
    }
}
The Verifying Compiler

“A verifying compiler uses automated .. reasoning to check the correctness of the program that it compiles.

Correctness is specified by types, assertions, .. and other redundant annotations that accompany the program.”

[Hoare, 2004]
Spec# Approach for a Verifying Compiler

• **Source language:**
  – C# + goodies = Spec#

• **Specifications:**
  – method contracts,
  – invariants,
  – field and type annotations

• **Program logic:**
  – Dijkstra’s weakest preconditions

• **Automatic verification:**
  – type checking,
  – verification condition generation (VCG),
  – automatic theorem proving (ATP)
Spec# Approach for a Verifying Compiler

**Source language:**
C# + goodies = Spec#

**Specifications:**
- method contracts,
- invariants,
- field and type annotations

**Program logic:**
- Dijkstra’s weakest preconditions

**Automatic verification:**
- type checking,
- verification condition generation (VCG),
- automatic theorem proving (ATP)

```
class C{
    private int a, z;
    invariant a!=0;

    public void M()
        requires a!=0;
        {z = 100/a;}
}
```

wp(S; T, Q) = wp(S, wp(T, Q))

wp(assert x>1, Q) = x>1 ∧ Q
The Microsoft Hypervisor

- **Meta OS**: small layer of software between hardware and OS
- **Mini**: 60K lines of non-trivial concurrent systems C code
- **Critical**: must guarantee isolation
- **Trusted**: a grand verification challenge
<table>
<thead>
<tr>
<th>Layer</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot</td>
<td>Bm</td>
<td>Boot Manager</td>
</tr>
<tr>
<td>Dispatch</td>
<td>Dm</td>
<td>Dispatch Manager</td>
</tr>
<tr>
<td>Interface</td>
<td>Hc</td>
<td>Hypercall Manager</td>
</tr>
<tr>
<td>Virtual Processor</td>
<td>Vp</td>
<td>Virtual Processor Manager</td>
</tr>
<tr>
<td>Virtualization Abstraction</td>
<td>Val</td>
<td>Virtualization Abstraction Layer</td>
</tr>
<tr>
<td>Virtualization Base</td>
<td>Vm</td>
<td>Virtualization Manager</td>
</tr>
<tr>
<td>Scheduler</td>
<td>Th</td>
<td>Thread/Process Manager</td>
</tr>
<tr>
<td>Resource Management</td>
<td>Mm</td>
<td>Memory Manager</td>
</tr>
<tr>
<td>Kernel</td>
<td>Ke</td>
<td>Physical Processor Manager</td>
</tr>
<tr>
<td>Hardware Abstraction</td>
<td>Hal</td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>Runtime</td>
<td>Cpu</td>
<td>Current Processor</td>
</tr>
<tr>
<td>Kernel Base</td>
<td>Hk</td>
<td>Hypervisor Kernel</td>
</tr>
</tbody>
</table>

Hypervisor Architecture is well layered!
What is to be verified?

• Source code
  – C + x64 assembly

• Sample verifiable slices:
  – **Safety**: Basic memory safety
  – **Security**: OS Isolation
  – **Utility**: Hypervisor services guest OS with available resources.
Tool: A **Verifying C Compiler**

- annotated C program
  - VCC
    - BoogiePL program
      - Boogie
        - verification condition
          - Z3
          - Isabelle
          - Simplify
            - program 💖 spec?
Methodologies

• A C-ish memory model
  – Abstract heaps
  – Bit-level precision

• Embedded specs in C
  – Pure functions

• Framing
  – Reads/writes

• Better together: Framing + Pure functions
Outlook

• The verification project has very recently started
• It is a multi-man multi-year effort
Z3 – an SMT Solver on Steroids
What is Z3 for?

- x86
- .NET CLR
- SAGE, Pex, Yogi, Vigilante
- VCC
- Spec#
- Windows Drivers
- SLAM Static Driver Verifier
- Backend: Z3, Disolver, Zaphatho
Inside Z3

Input

Rewriting
Simplification

Congruence
Closure Core

SAT solver

Satellite theories

- Bit-Vectors
- Linear arithmetic
- Arrays
- Pointers
- Partial orders
- Tuples

Abstract matching machine
Z3 topics

• Based on SMT: SAT + Specialized constraint solvers
• Model-based Theory Combination
• Efficient E-matching for DPLL(QT)
Model based theory combination: Example

\[ x = f(y - 1), \ f(x) \neq f(y), \ 0 \leq x \leq 1, 0 \leq y \leq 1 \]

Purifying

\[ x = f(z), \ f(x) \neq f(y), \ 0 \leq x \leq 1, 0 \leq y \leq 1, \ z = y - 1 \]

How to combine theory solvers?
Model-based Theory Combination

Equalities

\[ x = f(z), \quad f(x) \neq f(y) \]

Arithmetic

\[ 0 \leq x \leq 1, \quad 0 \leq y \leq 1, \quad z = y - 1 \]

**Standard:**
propagate \( x \simeq y \) to \( \Gamma_2 \) whenever \( \mathcal{T}_1 \cup \Gamma_1 \models x \simeq y \),

**Model-based:**
Use a candidate model \( M_i \) for one of the theories \( \mathcal{T}_i \) and propagate all equalities implied by the candidate model, hedging that other theories will agree.

\[ \text{if } M_i \models \mathcal{T}_i \cup \Gamma_i \cup \{ u = v \} \text{ then propagate } u = v \]

If not, use backtracking to fix the model.

It is cheaper to enumerate equalities that are implied in a particular model than of all models.
## Model based theory combination: Example

<table>
<thead>
<tr>
<th>$T_\mathcal{E}$</th>
<th>$T_\mathcal{A}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Literal</strong></td>
<td><strong>Eq. Classes</strong></td>
</tr>
<tr>
<td>$x = f(z)$</td>
<td>${x, f(z)}$</td>
</tr>
<tr>
<td>$f(x) \neq f(y)$</td>
<td>${y}$</td>
</tr>
<tr>
<td></td>
<td>${z}$</td>
</tr>
<tr>
<td></td>
<td>${f(x)}$</td>
</tr>
<tr>
<td></td>
<td>${f(y)}$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume $x = y$
# Model based theory combination: Example

<table>
<thead>
<tr>
<th>Literals</th>
<th>Eq. Classes</th>
<th>Model</th>
<th>Literals</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = f(z)$</td>
<td>${x, y, f(z)}$</td>
<td>$x^E = *_1$</td>
<td>$0 \leq x \leq 1$</td>
<td>$x^A = 0$</td>
</tr>
<tr>
<td>$f(x) \neq f(y)$</td>
<td>${z}$</td>
<td>$y^E = *_1$</td>
<td>$0 \leq y \leq 1$</td>
<td>$y^A = 0$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>${f(x), f(y)}$</td>
<td>$z^E = *_2$</td>
<td>$z = y - 1$</td>
<td>$z^A = -1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$f^E = {*_1 \mapsto *_3, }$</td>
<td></td>
<td>$x = y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$*_2 \mapsto *_1, \text{ else } \mapsto *_4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unsatisfiable
### Model based theory combination: Example

<table>
<thead>
<tr>
<th>$\mathcal{T}_E$</th>
<th>$\mathcal{T}_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Literals</strong></td>
<td><strong>Eq. Classes</strong></td>
</tr>
<tr>
<td>$x = f(z)$</td>
<td>${x, f(z)}$</td>
</tr>
<tr>
<td>$f(x) \neq f(y)$</td>
<td>${y}$</td>
</tr>
<tr>
<td>$x \neq y$</td>
<td>${z}$</td>
</tr>
<tr>
<td></td>
<td>${f(x)}$</td>
</tr>
<tr>
<td></td>
<td>${f(y)}$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Backtrack, and assert $x \neq y$.*

$\mathcal{T}_A$ model need to be fixed.
### Model based theory combination: Example

<table>
<thead>
<tr>
<th>$T_{\varepsilon}$</th>
<th>$T_{\Delta}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Literals</strong></td>
<td><strong>Eq. Classes</strong></td>
</tr>
<tr>
<td>$x = f(z)$</td>
<td>${x, f(z)}$</td>
</tr>
<tr>
<td>$f(x) \neq f(y)$</td>
<td>${y}$</td>
</tr>
<tr>
<td>$z \neq y$</td>
<td>${z}$</td>
</tr>
<tr>
<td></td>
<td>${f(x)}$</td>
</tr>
<tr>
<td></td>
<td>${f(y)}$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume $x = z$
# Model based theory combination: Example

<table>
<thead>
<tr>
<th>(T_\mathcal{E})</th>
<th>(T_\mathcal{A})</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Literals</strong></td>
<td><strong>Eq. Classes</strong></td>
</tr>
<tr>
<td>(x = f(z))</td>
<td>({x, z, f(x), f(z)})</td>
</tr>
<tr>
<td>(f(x) \neq f(y))</td>
<td>({y})</td>
</tr>
<tr>
<td>(x \neq y)</td>
<td>({f(y)})</td>
</tr>
<tr>
<td>(x = z)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Satisfiable*
Efficient E-matching for SMT Solvers

Deductive setting:
E – asserted equalities, P – set of patterns, C – logical context

Assert equality: \[ E, P \parallel C \Rightarrow E \cup \{ s = t \}, P \parallel C \]
Add pattern: \[ E, P \parallel C \Rightarrow E, P \cup \{ p(x): \forall x. \varphi(p(x)) \} \parallel C \]
Instantiate: \[ E, P \parallel C \Rightarrow E, P \parallel C \cup \{ [\forall x. \varphi(p(x))] \Rightarrow \varphi(p(t)) \} \]
E-matching example

\[ \text{select(store(x,i,v),i)} : \forall x,i,v. \text{select(store(x,i,v),i)} = v \]

\[ 5 = \text{select}(b, 2), \]
\[ \{ \text{c = store(a, 2, 4)}, \} \models 5 = \text{select}(\text{store}(x,y,z),y) [ x \rightarrow a, \]
\[ y \rightarrow 2, \]
\[ z \rightarrow 4 \]
Incremental matching

\[ 5 = \text{select}(b,2) \]

\[ E_1 = \{ \{5, \text{select}(b,2)\}, \{b\} \} \]

\[ c = \text{store}(a,2,4) \]

\[ E_2 = E_1 \cup \{ \{c, \text{store}(a,2,4)\} \} \]

\[ b = c \]

\[ E_3 = \{ \{b, \text{store}(a,2,4)\}, \{5, \text{select}(b,2)\} \} \]

\[ E_3 \models 5 = \text{select}(b,2) = \text{select}(\text{store}(a,2,4),2) \]

Observation: pattern \text{select}(\text{store}(x,i,v),i) gets enabled when \textit{child} of \texttt{select} is merged with term labeled by \texttt{store}. 
Inverted path indices

Index all patterns with $f(\ldots g(\ldots)\ldots)$ sub-term, that may become enabled when

merge($n_1$, $n_2$) where

$\exists$ parent $p_1$ of $n_1$. Label($p_1$) = $f(\ldots n_1\ldots)$

$\exists$ sibling $m_2$ of $n_2$. Label($m_2$) = $g(\ldots)$

<table>
<thead>
<tr>
<th>Pattern id</th>
<th>pattern</th>
<th>Path to g under f</th>
<th>Inverted paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>$f(f(\ g(x) ,a),x)$</td>
<td>p1 $\rightarrow$ g: f.1.f.1</td>
<td>(f,g): f.1.f.1 $\rightarrow$ p1</td>
</tr>
<tr>
<td>p2</td>
<td>$h(c, f(\ g(y) , x))$</td>
<td>p2 $\rightarrow$ g: h.2.f.1</td>
<td>(f,g): f.1.h.2 $\rightarrow$ p2</td>
</tr>
<tr>
<td>p3</td>
<td>$f(f(\ g(x) ,b),y)$</td>
<td>p3 $\rightarrow$ g: f.1.f.1</td>
<td>(f,g): f.1.f.1 $\rightarrow$ p3</td>
</tr>
<tr>
<td>p4</td>
<td>$f(f(a, \ g(x) ), \ g(y) )$</td>
<td>p4 $\rightarrow$ g: f.1.f.2, f.2</td>
<td>(f,g): f.2.f.1 $\rightarrow$ p4, (f,g): f.2 $\rightarrow$ p4</td>
</tr>
</tbody>
</table>
Inverted path index
Summary

• Formal Verification is hot at MSR
• I gave the flavor of two selected projects
  – Automatic generation of unit tests
  – Pex, SAGE
• I gave a taste of Z3: plumbing behind the scenes
  – Theory combination techniques
  – Quantifier instantiation