Untold stories of Synthesis, Verification, and Runtime Checking

Eran Yahav
IBM T.J. Watson Research Center

Joint Work with Martin Vechev and Matt Arnold
The Bigger Picture

Synthesis

Paraglide

Static

SAFE

Software Reliability

Dynamic

QVM
Paraglide: Synthesis of Concurrent System-level Code

- Generate **efficient provably correct** components of concurrent systems from higher-level specs
  - Verification/checking integrated into the design process
  - Automatic exploration of implementation details

- Deriving concurrent GC algorithms [PLDI06]
- Synthesis of concurrent GC algorithms [PLDI07]
- Deriving linearizable concurrent data-structures [PLDI08]
- Machine assisted construction of concurrent algorithms [EC2-08]
- Inferring synchronization under limited observability [TACAS‘09]
- Experience model-checking linearizability [SPIN‘09]
- Abstraction-guided synthesis [POPL‘10]

Bacon, Rinetzky, Vechev, Yorsh
SAFE: Scalable and Flexible Error-detection and Verification

- Lightweight verification for (real) Java programs
- Typestate properties
  - e.g., “don’t read from a file after it has been closed”
- Typestate verification in presence of aliasing [ISSTA’06]
- Static Specification Mining [ISSTA’07]
- Modular Typestate [POPL’08]
- Verifying Dereference Safety via Expanding-Scope Analysis [ISSTA’08]
- Synthesis of resource management code [ISMM’08]
- ...

Chandra, Dor, Dillig, Fink, Geay, Loginov, Pistoia, Ramalingam, Shoham, Yorsh
QVM: The Quality Virtual Machine

- A “quality-aware” virtual machine
- Increase diagnosability of bugs in the field
- Controlled overhead – can be low enough for production
- Tap into the VM’s wealth of information

- Sample applications
  - Checking expressive user assertions (including heap assertions)
  - Dynamic typestate checking
  - Many more...

- QVM: An Efficient Runtime for Detecting Defects in Deployed Systems [OOPSLA08]
- Chameleon: Adaptive Selection of Collections [PLDI’09]
- Parallel Checking of Heap Assertions

Arnold, Shacham, Vechev, Yorsh
SAFE

- SAFE
  - **Scalable And Flexible Error-detection and verification**
- SAFE Mining
- ...

SAFE

- Scalable And Flexible Error-detection and verification
- SAFE Mining
- ...

SAFE
Motivation

- **Application Trend**: Increasing number of libraries and APIs
  - Non-trivial restrictions on permitted sequences of operations
- **Typestate**: Temporal safety properties
  - What sequence of operations are permitted on an object?
  - Encoded as DFA
  - *e.g.* “Don’t use a Socket unless it is connected”
Goal

- **Typestate Verification**: statically ensure that no execution of a Java program can transition to **err**
  - Sound\(^1\) (excluding concurrency)
  - Precise enough\(^2\) (reasonable number of false alarms)
  - Scalable\(^3\) (handle programs of realistic size)

---

1 In the real world, some other caveats apply
2 we’ll get back to that
3 relatively speaking
Challenges

class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) {
    l.connect();
}
talk(Socket s) {
    s.getOutputStream()).write("hello");
}

main() {
    Set<SocketHolder> set = new HashSet<SocketHolder>();
    while(...) {
        SocketHolder h = new SocketHolder();
        h.s = makeSocket();
        set.add(h)
    }
    for (Iterator<SocketHolder> it = set.iterator(); ...) {
        Socket g = it.next().s;
        open(g);
        talk(g);
    }
}
Our Approach

- Flow-sensitive, context-sensitive interprocedural dataflow analysis
  - Abstract domains combine typestate and pointer information
    - More precise than 2-stage approach
    - Concentrate expensive effort where it matters
  - Staging: Sequence of abstractions of varying cost/precision
    - Inexpensive early stages reduce work for later expensive stages
  - Techniques for inexpensive strong updates (Uniqueness, Focus)
    - Much cheaper than typical shape analysis
    - More precise than usual “scalable” analyses

- Results
  - Flow-sensitive functional IPA with sophisticated alias analysis on ~100KLOC in 10 mins.
  - Verify ~92% of potential points of failure (PPF) as safe
Analysis Overview

Program → Preliminary Pointer Analysis/Call Graph Construction → Composite Typestate Verifier → Possible failure points

Initial Verification Scope → Intraprocedural Verifier → Unique Verifier → AP Focus Verifier

Dataflow Analysis
- Sound, abstract representation of program state
- Flow-sensitive propagation of abstract state
- Context-sensitive: Tabulation Solver [Reps-Horwitz-Sagiv 95]
(Instrumented) Concrete Semantics

σ = { <o1, init>, <o2, closed>, <o3, init>, ... }
Abstract State

\[ \sigma = \{ \langle o_1, \text{init} \rangle, \langle o_2, \text{closed} \rangle, \langle o_3, \text{init} \rangle, \ldots \} \]

\[ \sigma^\# = \{ \langle \text{AS}_1, \text{init} \rangle, \langle \text{AS}_1, \text{closed} \rangle \} \]
Base Abstraction

```
open(Socket s) { s.connect(); }
talk(Socket s) { s.getOutputStream().write("hello"); }
dispose(Socket s) { s.close(); }
main() {
    Socket s = new Socket(); // S
    open(s);
    talk(s);
    dispose(s);
}
```
Unique Abstraction

Abstract State := { <Abstract Object, TypeState, UniqueBit> }

- “UniqueBit” ≈ “∃ exactly one concrete instance of abstract object”
- Allows strong updates

```java
main() {
    Socket s = new Socket(); // S
    open(s);
    talk(s);
    dispose(s);
}
```

```java
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
```
open(Socket s) { s.connect(); }
talk(Socket s) { s.getOutputStream().write("hello"); }
dispose(Socket s) { s.close(); }
main() {
  while (...)
  {
    Socket s = new Socket(); // S
    open(s); // S, init, U
    talk(s); // S, connected, U
    dispose(s); // S, closed, U
    ...
  }
}

Object liveness analysis to the rescue
  ▪ Preliminary live analysis oracle
  ▪ On-the-fly remove unreachable configurations

More than just singletons?

Unique Abstraction
**Access Path Must**

\[
\text{MustSet} := \text{set of symbolic access paths (x.f.g....) that must point to the object}
\]

\[
\text{MayBit} := \text{“must set is incomplete. Must fall back to may-alias oracle”}
\]

- Strong Updates allowed for \( e.op() \) when \( e \in \text{Must} \) or unique logic allows

**Access Path Focus**

\[
\text{MustNotSet} := \text{set of symbolic access paths that must not point to the object}
\]

**Focus** operation when interesting things happen

- **generate 2 tuples**, a **Must** information case and a **MustNot** information case

- Only track access paths to “interesting” objects

- Sound flow functions to *lose precision in MustSet, MustNotSet*
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket t) {  
  t.connect();  
}
talk(Socket s) {  
  s.getOutputStream().write("hello");  
}
dispose(Socket s) { h.s.close(); }
main() {  
  Set<SocketHolder> set = new HashSet<SocketHolder>();  
  while(...) {  
    SocketHolder h = new SocketHolder();  
    h.s = makeSocket();  
    set.add(h);  
  }  
  for (Iterator<SocketHolder> it = set.iterator(); ...) {  
    Socket g = it.next().s;  
    open(g);  
    talk(g);  
  }  
}
Implementation Details Matter

**Sparsification**
Separation (solve for each abstract object separately)
“Pruning”: discard branches of supergraph that cannot affect abstract semantics
- Reduces median supergraph size by 50X

**Preliminary Pointer Analysis/Call Graph Construction**

Details matter a lot
- if context-insensitive preliminary, stages time out, terrible precision

Current implementation:
- Subset-based, field-sensitive Andersen’s
- SSA local representation
- On-the-fly call graph construction
- Unlimited object sensitivity for
  - Collections
  - Containers of typestate objects (e.g. IOStreams)
- One-level call-string context for some library methods
- Heuristics for reflection (e.g. Livshits et al 2005)
Sources of False Positives

- Limitations of analysis
  - Aliasing
  - Path sensitivity
  - Return values

- Limitations of typestate abstraction
  - Application logic bypasses DFA, still OK
Running time

IBM Intellistation Zpro 2x3GHz Xeon
3.62 GB RAM/ Win XP
IBM J2RE 1.4.2 / -Xmx800M

APMust = 1677
APFocus = 4275
Back to the Fine Print

- **Soundness**
  - Just have to understand the assumptions

- **Precision**
  - 7% false alarms is very low
  - Until you have to go out and hunt them in real code
  - For a program with 1000 PPFs, looking at 70 points in the program
More Fine Print

- Scalable (code size)
  - Currently easily handles ~300,000 LOCs
  - Can probably go up to 1 million lines of code with better engineering
  - This is still not enough for real world software (libraries are the enemy)

- Scalable (specifications)
  - What are you checking?
  - Single object typestate is limited
  - Where are you going to get specs?
Challenges

- Millions of lines of code?
- What properties should we check?
- How can we get specifications?
SAFE Mining

- Client-side specification mining
  - based on flow-sensitive, context-sensitive abstract interpretation
  - combined domain abstracting both aliasing and event sequences
- Novel family of abstractions to represent unbounded event sequences
- Novel summarization algorithms
- Preliminary experimental results
java.security.Signature

Base/Past/Total

Base/Past/Exterior

APFocus/Past/Exterior
Ganymed Session

Base/Past/Exterior

APFocus/Past/Exterior

(all results here are actual images produced by the tool)
SAFE Mining

- Hit a scalability wall
- Worse than “just” checking the properties

- Solvable by engineering compromises
  - Timing out still gives useful results
  - Analyze only parts of the program
  - Ignore/prune libraries

- May work in practice...
Make writing specs more rewarding

- Enable developer to write runtime checks without having to worry about performance impact

- Expressive assertions
  - beyond what is (naturally/efficiently) expressible in the programming language

- Monitoring of global properties

- Collection of debug information when a property is violated
Challenges

- Predictable overhead
- Expressive assertions
- Helping find a fix
QVM: Quality-aware VM

- Leverage **available** system resources for software quality checks
- Adaptive overhead management
  - QVM monitors overhead it incurs, adapts analyses to meet user specified target overhead
  - **No free lunches** – allow “unknown” result
- Support a variety of analysis clients
  - Typestate properties
  - Heap probes
  - Java assertions
- **Improve Diagnosability**
  - Collect (sampled) debug information
Motivating Example: Azureus

Over 160 million downloads
QVM: GDI Resource Leaks

Example Leaking Code

class ListView extends ... {
    private Image imgView = null;
    // ...
    protected void handleResize(boolean bForce) {
        // ...
        if (imgView == null || bForce) {
            imgView = new Image(listCanvas.getDisplay(), clientArea);
            lastBounds = new Rectangle(0, 0, 0, 0);
            bNeedsRefresh = true;
        } else {
            // ...
        }
        // ...
    }
}
protected void handleResize(boolean bForce) {
    // ...
    if (imgView == null || bForce) {
        if (imgView != null && !imgView.isDisposed()) {
            assert (!QVM.isShared(imgView));
            imgView.dispose();
        }
        imgView = new Image(listCanvas.getDisplay(), clientArea);
        lastBounds = new Rectangle(0, 0, 0, 0);
        bNeedsRefresh = true;
    } else {
        // ...
    }
    // ...
}
QVM Architecture

- **Application**
  - **Clients**
    - typestate client
    - heap probes client
    - assertions client
  - **QVMI**
    - observed overhead
    - event filters
    - event callbacks
  - **OHM**
    - specified overhead
    - adjust sampling rates
  - **Execution Engine**
  - **VM Core**
  - **QVM**

- **Execution Engine**
  - event filters
  - event callbacks

- **Client**
  - typestate client
  - heap probes client
  - assertions client

- **QVMI**
  - QVM

- **OHM**
  - specified overhead
  - adjust sampling rates

- **QVM Architecture**
  - violations report
  - overhead
  - typestate specs
  - specified overhead
QVMI: The QVM Interface

- Key: filtering on the VM side
Overhead Manager (OHM)

- **Monitoring**: measure overhead incurred by clients
- **Sampling strategy**: separate sampling rates for different origins
- **Controller**: adjust strategy per origin based on measured overhead

![Diagram of Overhead Manager (OHM)]
Overhead Manager

Sampling Information

Average time per origin (cycles)
Object Centric Sampling

```java
assert (...)()

T t = new T()

assert(...)()
```
Clients

- Typestate Properties
  - With debug information (typestate histories)
  - Including times

- Heap Probes & Operations
  - Check ownership/sharing/shape properties
  - Runtime support for transfer of ownership

- Java Assertions

- Overhead of all clients managed by OHM
Typestate Properties

- undisposed
- disposed
- err

*Dispose* or *Release*

Object death
Typestate History
<table>
<thead>
<tr>
<th>Function</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image.createMask(ImageData;Z)I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image.init(Device;ImageData:)V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image.&lt;init&gt;(Device;InputStream:)V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image.isDisposed()Z</td>
<td>69/39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image.getData()ImageData;</td>
<td>4/39</td>
<td>4/39</td>
<td>65/39</td>
</tr>
<tr>
<td>Image.getBounds()Rectangle;</td>
<td>65/39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image.createMask()V</td>
<td>65/52</td>
<td>64/52</td>
<td>1/52</td>
</tr>
<tr>
<td>CLabel.getTotalSize(Image;...)Point;</td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>CLabel.computeSize(Image;...)Point;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC.drawImageMask(Image;...)V</td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>GC.drawImage(Image;...)V</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Heap Probes & Operations

- **Heap Probes**
  - Ownership and sharing properties
  - Reachability...

- **Heap Operations**
  - Runtime support for transfer of ownership

- Use components of a parallel GC to evaluate heap queries
SWT Example

canvas.addDisposeListener(new DisposeListener() {
    public void widgetDisposed(DisposeEvent argo) {
        if (img != null && !img.isDisposed()) {
            assert (QVM.isObjectOwned(img));
            img.dispose();
        }
    }
});
Experimental Evaluation
Overhead Manager: stabilization
Overhead Manager

<table>
<thead>
<tr>
<th>Overhead</th>
<th>javac</th>
<th>jess</th>
<th>jack</th>
<th>db</th>
<th>luindex</th>
<th>bloat</th>
<th>hsqldb</th>
<th>chart</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base overhead</td>
<td>970.7</td>
<td>60.1</td>
<td>114.5</td>
<td>261.7</td>
<td>142.4</td>
<td>206.1</td>
<td>189.7</td>
<td>41.2</td>
<td>272</td>
</tr>
<tr>
<td>5% Budget</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.7</td>
</tr>
<tr>
<td>10% Budget</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>20% Budget</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.7</td>
</tr>
<tr>
<td>Exhaustive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.7</td>
</tr>
</tbody>
</table>

Budgets: 5%, 10%, 20%
## Leak Detection Results

<table>
<thead>
<tr>
<th>Application</th>
<th>SWT Resources</th>
<th>IOStreams</th>
<th>High Frequency</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azureus</td>
<td>11</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Etrader</td>
<td>17</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Feednread</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Goim</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>IBMapp1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IBM app2</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jcommander</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>J uploader</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nomadpim</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rssowl</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tvbrowser</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tvla</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Virgoftp</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>60</strong></td>
<td><strong>22</strong></td>
<td><strong>7</strong></td>
<td><strong>14</strong></td>
</tr>
</tbody>
</table>
Sampling coverage (5% budget)

- **Global sampling**
- **Origin-centric sampling**

Bar chart showing the percent of allocation sites sampled for various projects, with blue bars representing global sampling and red bars representing origin-centric sampling. The chart compares these sampling methods across different projects and shows the average percent of sites sampled.
## Sampling coverage (typestate)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1%</th>
<th>2%</th>
<th>5%</th>
<th>10%</th>
<th>20%</th>
<th>50%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mpegudio</td>
<td>98</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>jess</td>
<td>63</td>
<td>76</td>
<td>85</td>
<td>87</td>
<td>95</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>jack</td>
<td>22</td>
<td>37</td>
<td>45</td>
<td>52</td>
<td>71</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>javac</td>
<td>0.4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>31</td>
<td>41</td>
<td>49</td>
</tr>
<tr>
<td>compress</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mtrt</td>
<td>39</td>
<td>46</td>
<td>66</td>
<td>83</td>
<td>90</td>
<td>93</td>
<td>94</td>
</tr>
<tr>
<td>antlr</td>
<td>13</td>
<td>19</td>
<td>34</td>
<td>68</td>
<td>67</td>
<td>92</td>
<td>98</td>
</tr>
<tr>
<td>eclipse</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>28</td>
<td>44</td>
<td>66</td>
<td>67</td>
</tr>
<tr>
<td>luindex</td>
<td>5</td>
<td>51</td>
<td>79</td>
<td>97</td>
<td>99</td>
<td>99</td>
<td>100</td>
</tr>
<tr>
<td>hsqldb</td>
<td>7</td>
<td>13</td>
<td>16</td>
<td>30</td>
<td>43</td>
<td>31</td>
<td>75</td>
</tr>
<tr>
<td>chart</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>fop</td>
<td>47</td>
<td>70</td>
<td>42</td>
<td>66</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>bloat</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>pmd</td>
<td>81</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
QVM Recap

- IBM’s production VM
- Adaptive overhead manager
- Analysis clients
  - Typestate
  - Heap probes
- Enables other analyses
  - Chameleon [PLDI’09]
  - Dynamic Shape Analysis
  - ...
- Diagnosability is key
An Ounce of Prevention...

- **Synthesis** of low-level code from higher-level specifications

- **Resource Leaks**
  - Can synthesis resource-management code to avoid leaks by construction [ISMM’08]

- **Concurrent Algorithms**
  - Paraglide [PLDI’06,07,08],[TACAS’09],[POPL’10]
Summary

- **SAFE**
  - Effective typestate verification (static)
  - Mining

- **QVM**
  - Adaptive overhead manager to enable dynamic checking of typestate properties
  - Debug information
  - Heap assertions, other expressive assertions

- **Paraglide**
  - Synthesis of synchronization in concurrent programs
The End
Invited Questions

1) How hard would it be to find these defects with standard testing?
2) Can I find these defects with static analysis?
3) Is QVM open-source?
4) Why muck with the VM?
5) Related Work?