Optimizing the Automated Programming Stack

James Bornholt
University of Washington
Software is everywhere
Bugs are everywhere

Facebook And Instagram Go Down In Second Snag This Week

Janet Burns
Consumer Tech
I cover AI, cybersecurity, culture, drugs, and more.

Global Facebook users have reported outages this morning on the web’s largest social media platform, as well as on its sister platform Instagram.

On Tuesday, social media users took to Twitter and other sites to report frequent log-in and site loading issues on Facebook, which the company says it is currently working to fix.
Bugs are everywhere

Facebook And Instagram Go Down In Second Snag This Week

Global Facebook users woke up to frequent log-in and frequent log-out, and now Facebook says it is currently working to resolve the issue.

Facebook Security Breach Exposes Accounts of 50 Million Users

SAN FRANCISCO (AP) -- Facebook says it is currently working to resolve a security breach that exposed the private information of 50 million users.

The breach, which affects data from the company's 14-year history, was a result of a bug in the company's 14-year-old system, the company said.

ext4 and data loss

Here's How IBM Crashed Australia's First Online Census

(SYDNEY) – International Business Machines (IBM, +0.88%) failed in its handling of the A$10 million ($7.4 million) IT contract for Australia’s first predominantly online census, Australian Prime Minister Malcolm Turnbull said on Friday.
Automated programming tools
Automated programming tools

Verifier

Specified

Language

✅

Program

Specification

✅

Synthesizer

Specified

Language

✅

+ test case

+ program
Automated programming successes

- Verified SQL optimizers
  [Chu et al, VLDB’18]

- Synthesized network configs
  [McClurg et al, PLDI’15]

- Synthesized crypto primitives
  [Erbsen et al, Oakland’19]

- Verified operating systems
  [Nelson et al, SOSP’17]

- Synthesized biology experiments
  [Köksal et al, POPL’13]

- Synthesized memory models
  [Bornholt et al, PLDI’17]

- Synthesized educational models
  [Butler et al, VMCAI’18]
Challenges in automated programming

Intractability
Most problems in automated programming are intractable (many undecidable).

Specification
Automated programming requires a specification, which is often difficult to construct and audit.
Challenges in automated programming

**Intractability**
Most problems in automated programming are intractable (many undecidable).

**Specification**
Automated programming requires a specification, which is often difficult to construct and audit.

**Domain specialization**
Specialization reduces the size of the search space, eliminating irrelevant programs/behaviors.

Specialization allows for concise and expressive specifications that capture programmer intent.
Automated programming stack

Domain-specific tools
Automated programming stack

Domain-specific tools

SAT/SMT solving
improvements in scale and expressiveness
Automated programming stack

Domain-specific tools

Symbolic evaluation algorithms to translate programs to SAT/SMT

SAT/SMT solving improvements in scale and expressiveness
Automated programming stack

Domain-specific tools

- Solver-aided languages
  front-end abstractions for verification/synthesis

- Symbolic evaluation
  algorithms to translate programs to SAT/SMT

- SAT/SMT solving
  improvements in scale and expressiveness
Automated programming stack

Domain-specific tools

Solver-aided languages
front-end abstractions for verification/synthesis

Symbolic evaluation
algorithms to translate programs to SAT/SMT

SAT/SMT solving
improvements in scale and expressiveness
New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.
New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.

MemSynth [PLDI’17]: an automated tool for synthesizing memory consistency models

Ferrite [ASPLOS’16]: a tool for synthesizing crash-safe file system code
New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.

- **MemSynth [PLDI’17]**: an automated tool for synthesizing memory consistency models
- **Ferrite [ASPLOS’16]**: a tool for synthesizing crash-safe file system code
- **Metasketches [POPL’16]**: a strategy abstraction for synthesis problems
- **SymPro [OOPSLA’18]**: a technique for systematically building scalable tools
New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.

MemSynth [PLDI’17]: an automated tool for synthesizing memory consistency models

Ferrite [ASPLOS’16]: a tool for synthesizing crash-safe file system code

Metasketches [POPL’16]: a strategy abstraction for synthesis problems

SymPro [OOPSLA’18]: a technique for systematically building scalable tools
Automated tools are *worth building*
The case of memory models [PLDI’17]

Building them can be *made systematic*
Symbolic profiling [OOPSLA’18]

The future is *more automation*
Automating the automated programming stack
Automated tools are worth building
The case of memory models [PLDI’17]

Building them can be made systematic
Symbolic profiling [OOPSLA’18]

The future is more automation
Automating the automated programming stack
Memory models define the memory ordering behavior of multiprocessors

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( X = 1 )</td>
<td>3 ( Y = 1 )</td>
</tr>
<tr>
<td>2 If ( Y == 0 ): \begin{align*} &amp; \text{print } \text{“hello”} \ &amp; \text{print } \text{“goodbye”} \end{align*}</td>
<td></td>
</tr>
</tbody>
</table>
Memory models define the memory ordering behavior of multiprocessors

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X = 1 )</td>
<td>( Y = 1 )</td>
</tr>
<tr>
<td>( \text{if } Y == 0: ) | ( \text{if } X == 0: )</td>
<td></td>
</tr>
<tr>
<td>( \text{print &quot;hello&quot;} )</td>
<td>( \text{print &quot;goodbye&quot;} )</td>
</tr>
</tbody>
</table>

All variables initialized to 0

Can this print... hello?
Memory models define the memory ordering behavior of multiprocessors

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. X = 1</td>
<td>3. Y = 1</td>
</tr>
<tr>
<td>2. if Y == 0: print &quot;hello&quot;</td>
<td>4. if X == 0: print &quot;goodbye&quot;</td>
</tr>
</tbody>
</table>

All variables initialized to 0

Can this print... hello?

1 2 3 4
Memory models define the memory ordering behavior of multiprocessors

Thread 1

1  X = 1
2  if Y == 0:
   print "hello"

Thread 2

3  Y = 1
4  if X == 0:
   print "goodbye"

All variables initialized to 0

Can this print... hello? goodbye?
Memory models define the memory ordering behavior of multiprocessors

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $X = 1$</td>
<td>3 $Y = 1$</td>
</tr>
<tr>
<td>2 if $Y == 0$:</td>
<td>4 if $X == 0$:</td>
</tr>
<tr>
<td>print “hello”</td>
<td>print “goodbye”</td>
</tr>
</tbody>
</table>

All variables initialized to 0

Can this print... hello? 1 2 3 4
goodbye? 3 4 1 2
Memory models define the memory ordering behavior of multiprocessors

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $X = 1$</td>
<td>3. $Y = 1$</td>
</tr>
<tr>
<td>2. if $Y == 0$:</td>
<td>4. if $X == 0$:</td>
</tr>
<tr>
<td>print “hello”</td>
<td>print “goodbye”</td>
</tr>
</tbody>
</table>

All variables initialized to 0

Can this print... hello? goodbye? nothing?
Memory models define the memory ordering behavior of multiprocessors

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X = 1 )</td>
<td>( Y = 1 )</td>
</tr>
<tr>
<td>( \text{if } Y == 0:\ \text{print } \text{“hello”} )</td>
<td>( \text{if } X == 0:\ \text{print } \text{“goodbye”} )</td>
</tr>
</tbody>
</table>

All variables initialized to 0

Can this print...  hello?  goodbye?  nothing?

1 2 3 4
3 4 1 2
1 3 2 4
Memory models define the memory ordering behavior of multiprocessors

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  (X = 1)</td>
<td>3  (Y = 1)</td>
</tr>
</tbody>
</table>
| 2  \(\text{if } Y == 0: \)
  \hspace{1cm} \text{print} “hello” | 4  \(\text{if } X == 0: \)
  \hspace{1cm} \text{print} “goodbye” |

All variables initialized to 0

Can this print... hello? goodbye? nothing? both?

\[\begin{array}{c}
\text{1 2 3 4} \\
\text{3 4 1 2} \\
\text{1 3 2 4}
\end{array}\]
Memory models define the memory ordering behavior of multiprocessors

Thread 1

1. \( X = 1 \)
2. \( \text{if } Y == 0: \)
   \( \text{print } \"hello\" \)

Thread 2

3. \( Y = 1 \)
4. \( \text{if } X == 0: \)
   \( \text{print } \"goodbye\" \)

All variables initialized to 0

Can this print...

- hello?
- goodbye?
- nothing?
- both?

No! (sequential consistency)
Memory models define the memory ordering behavior of multiprocessors

Thread 1

1. \( X = 1 \)
2. \( \text{if } Y == 0: \)
   \( \text{print} \) “hello”

Thread 2

3. \( Y = 1 \)
4. \( \text{if } X == 0: \)
   \( \text{print} \) “goodbye”

All variables initialized to 0

Can this print... hello? goodbye? nothing? both?

1 2 3 4
3 4 1 2
1 3 2 4

No! (sequential consistency)

Yeah! We wanna go fast!
Memory models define the memory ordering behavior of multiprocessors

- Correctness of my compiler...
- Rules to verify against...
- Possible low-level behaviors...

Compiler writers

Verification tools

Kernel/library developers
Memory models define the memory ordering behavior of multiprocessors

...correctness of my compiler...
Compiler writers

...rules to verify against...
Verification tools

...possible low-level behaviors...
Kernel/library developers

Litmus tests and prose
Memory models define the memory ordering behavior of multiprocessors

- Correctness of my compiler
- Rules to verify against
- Possible low-level behaviors

Compiler writers

Verification tools

Kernel/library developers

Litmus tests and prose

Formal specifications
Memory models define the memory ordering behavior of multiprocessors

- Correctness of my compiler...
- Rules to verify against...
- Possible low-level behaviors...

Compiler writers

Verification tools

Kernel/library developers

Litmus tests and prose

Formal specifications

- x86 [Sewell et al, CACM’10]
- PowerPC [Alglave et al, CAV’10, etc]
- ARM [Flur et al, POPL’16]
MemSynth: automated programming for memory consistency models

Litmus tests

Formal specifications
MemSynth: automated programming for memory consistency models

Litmus tests

Synthesize specifications from litmus tests

Formal specifications
MemSynth: automated programming for memory consistency models

Litmus tests

Synthesize specifications from litmus tests

Detect ambiguities in synthesized models

Formal specifications
MemSynth: automated programming for memory consistency models

- Synthesize specifications from litmus tests
- Detect ambiguities in synthesized models

x86: 2 seconds  
PowerPC: 12 seconds

- x86: 4 ambiguities  
PowerPC: 9 ambiguities
MemSynth: automated programming for memory consistency models

1. Litmus tests as relations
2. Memory models as constraints
3. Synthesis via sketches

Litmus tests

Formal specifications
Litmus tests as relations

Thread 1

1. \( X = 1 \)
2. \( \text{if } Y == 0: \)
   \( \text{print "hello"} \)

Thread 2

3. \( Y = 1 \)
4. \( \text{if } X == 0: \)
   \( \text{print "goodbye"} \)

All variables initialized to 0
## Litmus tests as relations

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 X = 1</td>
<td>3 Y = 1</td>
</tr>
<tr>
<td>2 r0 = Y</td>
<td>4 r1 = X</td>
</tr>
</tbody>
</table>

All variables initialized to 0
Litmus tests as relations

Encode programs and behaviors as *relations* in *relational logic* (like Alloy)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( X = 1 )</td>
<td>3 ( Y = 1 )</td>
</tr>
<tr>
<td>2 ( r0 = Y )</td>
<td>4 ( r1 = X )</td>
</tr>
</tbody>
</table>

All variables initialized to 0
## Litmus tests as relations

### Thread 1

1. $X = 1$
2. $r0 = Y$

### Thread 2

3. $Y = 1$
4. $r1 = X$

All variables initialized to 0

Encode programs and behaviors as **relations** in **relational logic** (like Alloy)

**Program relations**

extracted from program text:

$$po = \{(1,2), (3,4)\}$$

**Program order:**

$(a,b) \in po$ if $b$ is after $a$ on the same thread
Litmus tests as relations

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ X = 1</td>
<td>✓ Y = 1</td>
</tr>
<tr>
<td>✓ r0 = Y</td>
<td>✓ r1 = X</td>
</tr>
</tbody>
</table>

All variables initialized to 0

Encode programs and behaviors as **relations** in **relational logic** (like Alloy)

**Program relations**
extracted from program text:

\[ po = \{(1,2), (3,4)\} \]

Program order:
\((a,b) \in po\) if \(b\) is after \(a\) on the same thread

**Execution relations**
describe dynamic behavior:

\[ rf = \{(2,3), (4,1)\} \]

Reads-from:
\((r,w) \in rf\) if \(r\) reads the value written by \(w\)
Litmus tests as relations

Program relations extracted from program text:

\[ po = \{ (1,2), (3,4) \} \]

Execution relations describe dynamic behavior:

\[ rf = \{ (2,3), (4,1) \} \]

Program order:
\( (a,b) \in po \) if \( b \) is after \( a \) on the same thread

Reads-from:
\( (r,w) \in rf \) if \( r \) reads the value written by \( w \)
Memory models as relational constraints

Program relations extracted from program text:

\[ po = \{(1, 2), (3, 4)\} \]

Execution relations describe dynamic behavior:

\[ rf = \{(2, 3), (4, 1)\} \]

A memory model constrains the allowed executions of a program.

Written as a predicate in relational logic.
Memory models as relational constraints

Program relations extracted from program text:

\[ \text{po} = \{(1, 2), (3, 4)\} \]

Program order

Execution relations describe dynamic behavior:

\[ \text{rf} = \{(2, 3), (4, 1)\} \]

Reads-from

A memory model constrains the allowed executions of a program

Written as a predicate in relational logic

\[ M(T, E) \triangleq \]

\[ \]
Memory models as relational constraints

Program relations extracted from program text:
\[\text{po} = \{(1, 2), (3, 4)\}\]

Program order

Execution relations describe dynamic behavior:
\[\text{rf} = \{(2, 3), (4, 1)\}\]

A memory model constrains the allowed executions of a program

Written as a predicate in relational logic

\[M(T, E) \triangleq \]

\[
&\& \&
\begin{align*}
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
&\& \& \\
\end{align*}
\]
Memory models as relational constraints

Program relations extracted from program text:

\[ po = \{(1,2), (3,4)\} \]

Program order

Execution relations describe dynamic behavior:

\[ rf = \{(2,3), (4,1)\} \]

Reads-from

A memory model constrains the allowed executions of a program

Written as a predicate in relational logic

\[
M(T, E) \triangleq (\text{no} \ (\land (\lor \ (po \ rf) \ iden))
\]
Memory models as relational constraints

Program relations extracted from program text:
\[ po = \{(1,2), (3,4)\} \]

Execution relations describe dynamic behavior:
\[ rf = \{(2,3), (4,1)\} \]

A memory model constrains the allowed executions of a program

Written as a predicate in relational logic

\[ M(T, E) \triangleq (\text{no} \ (\& \ (\^ (\ + \ po \ rf)) \ \text{idem})) \]

...by forbidding cycles involving \( rf \cup po \)

Constraining the possible values of \( rf \)...
Memory models as relational constraints

Program relations
extracted from program text:

\[ \text{po} = \{(1,2), (3,4)\} \]

Execution relations
describe dynamic behavior:

\[ \text{rf} = \{(2,3), (4,1)\} \]

A memory model constrains the allowed executions of a program

Written as a predicate in relational logic

\[ M(T,E) \triangleq (\text{no} \ (\& \ (\wedge \ (\uplus \ \text{po} \ \text{rf})) \ \text{idem})) \]

A memory model allows a test T if there exists an execution E that satisfies the predicate

...by forbidding cycles involving \( \text{rf} \cup \text{po} \)

Constraining the possible values of \( \text{rf} \)...
Synthesis from a memory model sketch

\[ M(T, E) \triangleq (\text{no} \: (\& \: (^\: (+ \: \text{po} \: \text{rf}) \: \text{iden})) ) \]
Synthesis from a memory model sketch

\[ M(T, E) \triangleq (\text{no } \& (\wedge (+ ?? ??)) \text{ iden}) \]
Synthesis from a memory model sketch

\[ M(T, E) \triangleq (\text{no} \ (\& \ (\wedge \ (+ \ ??? \ ???)) \ \text{idem})) \]

Expression holes for a synthesizer to complete:
- po
- rf
- po + rf
- po & rf
- po - rf
- ...

Here, the `no` and `iden` terms represent placeholders for the synthesizer to fill in with appropriate expressions.
Synthesis from a memory model sketch

\[ M(T, E) \triangleq (\text{no} \ (\& \ (^{+} \ ?? \ ??)) \ \text{idem}) \]

A sketch specifies things we know (e.g., want a happens-before ordering)...

Expression holes for a synthesizer to complete:
- po
- rf
- po + rf
- po & rf
- po - rf
- ...

M(T, E) ≜ (no (& (^ (+ ?? ??)) idem))
Synthesis from a memory model sketch

A sketch specifies things we know (e.g., want a happens-before ordering)...

...and defines the shape of the parts we don't know

\[ M(T, E) \triangleq (\text{no} \ (\& \ (\wedge \ (+ \ ??? \ ???)) \ \text{iden})) \]

Expression holes for a synthesizer to complete:

- po
- rf
- po + rf
- po & rf
- po - rf
- ...
Memory model frameworks

\[ M(T, E) \triangleq (\textbf{no} \ (\& \ (^\ (\ + \ ws \ rf \ \textbf{ppo} \ \textbf{grf}) \ \textbf{idem})) \)
Memory model frameworks

\[ M(T, E) \triangleq (\text{no} \land (^{+ \text{ws} \text{ rf} \text{ ppo} \text{ grf}}) \text{idem}) \]

Preserved program order: same-thread reorderings

Global reads-from: inter-thread reorderings

[Alglave et al, CAV’10]
Memory model frameworks

\[ M(T, E) \triangleq (\text{no} \ (\& \ (^{\land} \ (+ \ ws \ rf \ ppo \ grf)) \ \text{idem})) \]

Sequential consistency

\[ \text{po} \quad \text{rf} \]

Total store order (x86)

\[ \text{po} - (\text{Wr} \rightarrow \text{Rd}) \quad \text{rf} \& \ \text{SameThd} \]

Preserved program order: same-thread reorderings

Global reads-from: inter-thread reorderings

[Alglave et al, CAV’10]
Memory model frameworks

\[ M(T, E) \triangleq (\text{no} \ (\& \ (^{\text{no}} \text{ ws \ rf \ ??? \ ???}) \text{ iden})) \]

- **Preserved program order**: same-thread reorderings
- **Global reads-from**: inter-thread reorderings

**Sequential consistency**
- \( M(T, E) \)

**Total store order** (x86)
- \( M(T, E) \)

[Alglave et al, CAV’10]
Ocelot DSL for relational logic with holes

$M(T, E) \triangleq \text{no} \ (\land \ (\lor \ (\land \ (\land \ (\land \ (? \ ?? \ ?? \ ))) \ \text{idem}))$
The synthesis query

- Allowed litmus tests
- Forbidden litmus tests

Synth

Completed memory model $M$

Memory model sketch $\hat{M}$
The synthesis query

2 allowed tests
3 5

8 forbidden tests
1 2 4 6
7 8 9 10

Synth

Total store order

Memory model sketch $\hat{M}$
The synthesis query

Allowed litmus tests → Synth → Completed memory model $M$

Forbidden litmus tests → Memory model sketch $\hat{M}$
The synthesis query

Allowed litmus tests → \exists M. \forall T \in T^+. M \text{ allows } T

Forbidden litmus tests →

Memory model sketch \hat{M}

Memory model \( M \)
The synthesis query

Allowed litmus tests

Forbidden litmus tests

\exists M. \forall T \in T^+. M \text{ allows } T

\exists M. \forall T \in T^- . M \text{ forbids } T

Memory model sketch \(\hat{M}\)

Memory model \(M\)
The synthesis query

Standard exists-forall quantifier pattern for synthesis

Allowed litmus tests \( \exists M. \forall T \in T^+. M \text{ allows } T \)

Forbidden litmus tests \( \exists M. \forall T \in T^- . M \text{ forbids } T \)

Memory model sketch \( \hat{M} \)

Memory model \( M \)
The synthesis query

Standard exists-forall quantifier pattern for synthesis

Allowed litmus tests

\[ \exists M. \forall T \in T^+. \; M \text{ allows } T \]

Forbidden litmus tests

\[ \exists M. \forall T \in T^- . \; M \text{ forbids } T \]

Memory model sketch \( \hat{M} \)

Memory model \( M \)
The synthesis query

- **Allowed litmus tests**
  \[ \exists M. \forall T \in T^+. \exists E. M(T, E) \]

- **Forbidden litmus tests**
  \[ \exists M. \forall T \in T^- \forall E. \neg M(T, E) \]

- **Memory model sketch** \( \hat{M} \)

- **Memory model** \( M \)

**Standard exists-forall quantifier pattern for synthesis**

**M allows T:**
\[ \exists E. M(T, E) \]
The synthesis query

- Standard exists-forall quantifier pattern for synthesis

- Allowed litmus tests

- Forbidden litmus tests

- \[ \exists M. \forall T \in T^+. \exists E. M(T, E) \]

- \[ \exists M. \forall T \in T^-. \forall E. \neg M(T, E) \]

- \[ M \text{ allows } T: \exists E. M(T, E) \]

- Memory model \( M \)

- Memory model sketch \( \hat{M} \)

- Higher-order quantification over relations!
The synthesis query

**M allows T:**
\[ \exists E. M(T, E) \]

**Allowed litmus tests**
\[ \exists M. \forall T \in T^+. \exists E. M(T, E) \]

**Forbidden litmus tests**
\[ \exists M. \forall T \in T^- . \forall E. \neg M(T, E) \]

**Memory model sketch \( \hat{M} \)**

**Memory model \( M \)**
The synthesis query

Allowed litmus tests

∀ T ∈ T⁺. ∃ E. M(T, E)

M allows T:
∃ E. M(T, E)

Forbidden litmus tests

∀ T ∈ T⁻. ∀ E. ¬M(T, E)

Memory model sketch ̂M

Handled by a quantified boolean formula (QBF) solver

Memory model M
The synthesis query

\[ M \text{ allows } T: \exists E. M(T, E) \]

Allowed litmus tests

\[ \exists M. \forall T \in T^+. \exists E. M(T, E) \]

\[ \exists M. \forall T \in T^- . \forall E. \neg M(T, E) \]

Forbidden litmus tests

Memory model sketch \( \hat{M} \)

Handled by a quantified boolean formula (QBF) solver

Handled by incremental synthesis engine

Memory model \( M \)
Incremental synthesis

Allowed litmus tests

Forbidden litmus tests

Synth
Incremental synthesis

Allowed litmus tests

Forbidden litmus tests

Synth

T₁  T₂
Incremental synthesis

Handled by a quantified boolean formula (QBF) solver
Incremental synthesis

Handled by a quantified boolean formula (QBF) solver
Incremental synthesis

Handled by a quantified boolean formula (QBF) solver
Incremental synthesis

Allowed litmus tests

Forbidden litmus tests

Handed by a quantified boolean formula (QBF) solver
Incremental synthesis

Handled by a quantified boolean formula (QBF) solver

Handled by a quantified boolean formula (QBF) solver
Incremental synthesis

Handled by a quantified boolean formula (QBF) solver

Empirically, need very few iterations to converge

Completed memory model $M'$

Allowed litmus tests

Forbidden litmus tests

Handled by a quantified boolean formula (QBF) solver
Disambiguating synthesized models

**Key idea:** after synthesis, is there a *different* memory model that also explains the input tests?
Disambiguating synthesized models

Key idea: after synthesis, is there a different memory model that also explains the input tests?
Disambiguating synthesized models

**Key idea:** after synthesis, is there a different memory model that also explains the input tests?
Disambiguating synthesized models

Key idea: after synthesis, is there a different memory model that also explains the input tests?

- Allowed litmus tests
- Forbidden litmus tests
- Completed memory model $M$
- Memory model sketch $\hat{M}$
- Completed memory model $M_2$
- Litmus test $T$
Disambiguating synthesized models

Key idea: after synthesis, is there a different memory model that also explains the input tests?

- Allowed litmus tests
- Forbidden litmus tests
- Completed memory model $M$
- Memory model sketch $\hat{M}$
- Completed memory model $M_2$
- Litmus test $T$

Difference between $M$ and $M_2$ is not just syntactic: they disagree about test $T$
Synthesizing existing memory models

x86

PowerPC
Synthesizing existing memory models

- x86: 10 tests
- PowerPC: 768 tests

[Alglave et al, CAV’10]
Synthesizing existing memory models

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tests</th>
<th>Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>10</td>
<td>✓ 2 seconds</td>
</tr>
<tr>
<td>PowerPC</td>
<td>768</td>
<td>✓ 12 seconds</td>
</tr>
</tbody>
</table>

[Alglave et al, CAV’10]
Synthesizing existing memory models

Synthesis

x86  10 tests  ✓ 2 seconds

Not equivalent to TSO!

PowerPC  768 tests  ✓ 12 seconds

[Alglave et al, CAV’10]
Synthesizing existing memory models

Synthesis

x86
- 10 tests
- ✓ 2 seconds
- Not equivalent to TSO!

PowerPC
- 768 tests
- ✓ 12 seconds
- Not equivalent to published model!

[Alglave et al, CAV’10]
Synthesizing existing memory models

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tests</th>
<th>Synthesis</th>
<th>Ambiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>10</td>
<td>✓ 2 seconds</td>
<td>4 new tests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mfence, xchg</td>
</tr>
<tr>
<td>PowerPC</td>
<td>768</td>
<td>✓ 12 seconds</td>
<td>9 new tests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sync, lwsync</td>
</tr>
</tbody>
</table>

Not equivalent to TSO!

Not equivalent to published model!
MemSynth: automated programming for memory consistency models

- Synthesize specifications from litmus tests
- Detect ambiguities in synthesized models
- Formal specifications

- x86: 2 seconds
  PowerPC: 12 seconds
- x86: 4 ambiguities
  PowerPC: 9 ambiguities
Automated tools are worth building
The case of memory models [PLDI’17]

Building them can be made systematic
Symbolic profiling [OOPSLA’18]

The future is more automation
Automating the automated programming stack
Automated tools are **worth building**
The case of memory models [PLDI’17]

Building them can be **made systematic**
Symbolic profiling [OOPSLA’18]

The future is **more automation**
Automating the automated programming stack
Scaling a synthesis tool is hard work
Scaling a synthesis tool is hard work

12 seconds
Scaling a synthesis tool is hard work
Scaling a synthesis tool is hard work

Finding these optimization opportunities is the key to good performance and new functionality.
A symbolic profiler identifies optimization opportunities in an automated tool.
A symbolic profiler identifies optimization opportunities in an automated tool.

What makes scaling an automated programming tool hard?

How does symbolic profiling work?

How effective is symbolic profiling?
Symbolic evaluation executes *all paths* through a program

```scheme
(filter even? '(3 6 8 2))
```
Symbolic evaluation executes *all paths* through a program

\[
\text{(filter even? '(3 6 8 2))}
\]

\[
'()
\]
Symbolic evaluation executes all paths through a program

\[(\text{filter even? '(3 6 8 2 ))}\]
Symbolic evaluation executes *all paths* through a program

\[
(f \text{ilter } \text{even? } '(3 \ 6 \ 8 \ 2))
\]
Symbolic evaluation executes all paths through a program

$$(\text{filter even? (3 6 8 2))}$$
Symbolic evaluation executes all paths through a program

```
(filter even? '(3 6 8 2))
```

- `(even? 3)`
- `(even? 6)`
- `(even? 8)`
- `(even? 2)`
Symbolic evaluation executes *all paths* through a program

\[(\text{filter even? } '(3\ 6\ 8\ 2))\]

Does this expression always return only *even* numbers?
Symbolic evaluation executes all paths through a program

Does this expression always return only even numbers?

Values of list elements are unknown (e.g., verifying filter for all inputs)
Symbolic evaluation executes all paths through a program

\[(\text{filter even? } '(X_0 \ X_1 \ X_2 \ X_3))\]

Values of list elements are unknown (e.g., verifying filter for all inputs)

Does this expression always return only even numbers?
Symbolic evaluation executes all paths through a program

\[
\text{(length (filter even? '(x_0 x_1 x_2 x_3)))}
\]

Does this expression always return only even numbers?

Values of list elements are unknown (e.g., verifying filter for all inputs)
Symbolic evaluation executes all paths through a program

\[(\text{length } (\text{filter even? } '(X_0 \ X_1 \ X_2 \ X_3)))\]

Does this expression always return only even numbers?

Values of list elements are unknown (e.g., verifying filter for all inputs)
Symbolic evaluation techniques

Symbolic execution

Always fork into independent paths (more paths, but more concrete)

Bounded model checking

Merge after every fork (fewer paths, but less concrete)
Symbolic evaluation techniques

- **Crucible** [Galois, Inc.]
- **Jalangi** [Sen et al 2014]
- **Rosette** [Torlak & Bodik 2014]

Symbolic execution:
- Always fork into independent paths (more paths, but more concrete)

Bounded model checking:
- Merge after every fork (fewer paths, but less concrete)
Symbolic evaluation techniques

**Crucible** [Galois, Inc.]

**Jalangi** [Sen et al 2014]

**Rosette** [Torlak & Bodik 2014]

Symbolic execution

- Always fork into independent paths (more paths, but more concrete)

Controlling the trade-off between these strategies is key to good scalability

Bounded model checking

- Merge after every fork (fewer paths, but less concrete)
Two data structures to summarize symbolic evaluation

Symbolic evaluation graph
Reflects the evaluator’s strategy for all-paths execution of the program

Symbolic heap
Shape of all symbolic values created by the program
Two data structures to summarize symbolic evaluation

Symbolic evaluation graph
Reflects the evaluator’s strategy for all-paths execution of the program

Symbolic heap
Shape of all symbolic values created by the program

Any symbolic evaluation technique can be summarized by these two data structures
Analyzing symbolic data structures

<table>
<thead>
<tr>
<th>Function</th>
<th>Score</th>
<th>Time (ms)</th>
<th>Term Count</th>
<th>Unused Terms</th>
<th>Union Size</th>
<th>Merge Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>4.3</td>
<td>1249</td>
<td>137408</td>
<td>131164</td>
<td>4288</td>
<td>93664</td>
</tr>
<tr>
<td>take</td>
<td>2.8</td>
<td>4692</td>
<td>50312</td>
<td>49986</td>
<td>2209</td>
<td>49986</td>
</tr>
<tr>
<td>andmap</td>
<td>0.3</td>
<td>94</td>
<td>14180</td>
<td>14180</td>
<td>0</td>
<td>4097</td>
</tr>
<tr>
<td>the-profiled-thunk</td>
<td>0.1</td>
<td>511</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
For each procedure, measure metrics that summarize the evolution of the symbolic evaluation graph and symbolic heap.

Summarize metrics as a score to rank procedures in the program.

<table>
<thead>
<tr>
<th>Function</th>
<th>Score</th>
<th>Time (ms)</th>
<th>Term Count</th>
<th>Unused Terms</th>
<th>Union Size</th>
<th>Merge Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter 1 call</td>
<td>4.3</td>
<td>1249</td>
<td>137408</td>
<td>131164</td>
<td>4288</td>
<td>93664</td>
</tr>
<tr>
<td>take 1 call</td>
<td>2.8</td>
<td>4692</td>
<td>50312</td>
<td>49986</td>
<td>2209</td>
<td>49986</td>
</tr>
<tr>
<td>andmap 1 call</td>
<td>0.3</td>
<td>94</td>
<td>14180</td>
<td>14180</td>
<td>0</td>
<td>4097</td>
</tr>
<tr>
<td>the-profiled-thunk</td>
<td>0.1</td>
<td>511</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Analyzing symbolic data structures

For each procedure, measure metrics that summarize the evolution of the symbolic evaluation graph and symbolic heap.

- The most likely bottleneck is not the slowest procedure.

Summarize metrics as a score to rank procedures in the program.

<table>
<thead>
<tr>
<th>Function</th>
<th>Score</th>
<th>Time (ms)</th>
<th>Term Count</th>
<th>Unused Terms</th>
<th>Union Size</th>
<th>Merge Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>4.3</td>
<td>1249</td>
<td>137408</td>
<td>131164</td>
<td>4288</td>
<td>93664</td>
</tr>
<tr>
<td>take</td>
<td>2.8</td>
<td>4692</td>
<td>50312</td>
<td>49986</td>
<td>2209</td>
<td>49986</td>
</tr>
<tr>
<td>andmap</td>
<td>0.3</td>
<td>94</td>
<td>14180</td>
<td>14180</td>
<td>0</td>
<td>4097</td>
</tr>
<tr>
<td>the-profiled-thunk</td>
<td>0.1</td>
<td>511</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Three symbolic profilers

We developed two implementations:

- The **Rosette** solver-aided language (Racket)
- The **Jalangi** dynamic analysis framework (JavaScript)

Since publication, based on our work:

- The **Crucible** symbolic simulation library (C, Java, ...) by Galois
Three symbolic profilers

We developed two implementations:

• The **Rosette** solver-aided language (Racket)
• The **Jalangi** dynamic analysis framework (JavaScript)

Since publication, based on our work:

• The **Crucible** symbolic simulation library (C, Java, ...) by Galois
Symbolic profiling in practice

Case studies: fixed 8 performance issues in 15 Rosette tools

Refinement type checker for Ruby [VMCAI’18] 6× speedup
Cryptographic protocol verifier [FM’18] 29× speedup
SQL query verifier [CIDR’17] 75× speedup
Safety-critical radiotherapy system verifier [CAV’16] 290× speedup
Symbolic profiling in practice

Case studies: fixed 8 performance issues in 15 Rosette tools

- Refinement type checker for Ruby [VMCAI'18] 6× speedup
- Cryptographic protocol verifier [FM'18] Used in production at the UW Medical Center 29× speedup
- SQL query verifier [CIDR'17] 75× speedup
- Safety-critical radiotherapy system verifier [CAV'16] 290× speedup
Symbolic profiling in practice

**Case studies:** fixed 8 performance issues in 15 Rosette tools

- Refinement type checker for Ruby [VMCAI’18] 6× speedup
- Cryptographic protocol verifier [FM’18] 29× speedup
- SQL query verifier [CIDR’17] 75× speedup
- Safety-critical radiotherapy system verifier [CAV’16] 290× speedup

**User study:** 8 Rosette users tasked with finding known performance issues in 4 programs

Users solved every task more quickly when they had access to symbolic profiling

- 6 failures without symbolic profiling vs. none with it
A symbolic profiler identifies optimization opportunities in an automated tool.
Automated tools are worth building
The case of memory models [PLDI'17]

Building them can be made systematic
Symbolic profiling [OOPSLA'18]

The future is more automation
Automating the automated programming stack
Automated tools are worth building
The case of memory models [PLDI’17]

Building them can be made systematic
Symbolic profiling [OOPSLA’18]

The future is more automation
Automating the automated programming stack
Automated programming abstractions

- **File systems**
  - [ASPLOS'16, OSDI'16]

- **Operating systems**
  - [SOSP'17, OSDI'18]

- **Memory models**
  - [PLDI'17]

- **Solver-aided languages**
  - front-end abstractions for verification/synthesis
  - [POPL'16]

- **Symbolic evaluation**
  - algorithms to translate programs to SAT/SMT
  - [OOPSLA'18]

- **SAT/SMT solving**
  - improvements in scale and expressiveness
Diagnosing SMT solver behavior

- **File systems** [ASPLOS’16, OSDI’16]
- **Operating systems** [SOSP’17, OSDI’18]
- **Memory models** [PLDI’17]

---

- **Solver-aided languages**
  - front-end abstractions for verification/synthesis [POPL’16]
- **Symbolic evaluation**
  - algorithms to translate programs to SAT/SMT [OOPSLA’18]
- **SAT/SMT solving**
  - improvements in scale and expressiveness

---

- **Metasketches**
- **Symbolic profiling** [OOPSLA’18]
Diagnosing SMT solver behavior

- Z3 version 4.8.3 slower and unable to solve problem that was solved by Z3 version 4.8.1
  #1979 by pjljvandelaar was closed on Dec 11, 2018

- Slow performance on simple query that uses equalities for assignments
  #1602 by 4tXJ7i was closed on Nov 25, 2018

- The solver slows down of java version when using multi-thread
  #1504 by destinyfucker was closed on Feb 24, 2018

- bv2int and int2bv slow?
  #1481 by kren1 was closed on Feb 14, 2018

- Incremental floating point is much slower than one-shot on certain short problems
  #1459 opened on Jan 24, 2018 by arotenberg

- Suspiciously slow on simple example
  #1425 opened on Dec 31, 2017 by DennisYurichev

- Performance surprisingly slow: since it can be solved very fast...
  #1352 by pjljvandelaar was closed on Dec 12, 2018

- (+ (- 1) str.len) instead of (- 1 str.len) make problem very slow to execute
  #1140 by jawline was closed on Jul 11, 2017

SAT/SMT solving improvements in scale and expressiveness
Diagnosing SMT solver behavior

File systems [ASPLOS'16, OSDI'16]

Operating systems [SOSP'17, OSDI'18]

Memory models [PLDI'17]

Solver-aided languages: front-end abstractions for verification/synthesis

Metasketches [POPL'16]

Symbolic evaluation: algorithms to translate programs to SAT/SMT

Symbolic profiling [OOPSLA'18]

SAT/SMT solving: improvements in scale and expressiveness

Solver profiling
Self-optimizing automated tools

File systems
[ASPLOS'16, OSDI'16]

Operating systems
[SOSP'17, OSDI'18]

Memory models
[PLDI'17]

Solver-aided languages
front-end abstractions for verification/synthesis

Metasketches
[POPL'16]

Symbolic evaluation
algorithms to translate programs to SAT/SMT

Symbolic profiling
[OOPSLA'18]

SAT/SMT solving
improvements in scale and expressiveness

Solver profiling

Exploit this profiling data for profile-guided optimization
Application opportunities

**File systems**
[ASPLOS’16, OSDI’16]

**Operating systems**
[SOSP’17, OSDI’18]

**Memory models**
[PLDI’17]

- Hardware accelerator design/programming
- High-performance low-precision kernels
New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.
New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.