Mining Specifications

Andreas Zeller
Saarland University
zeller@acm.org
FSE ‘10 Keynote

Avoiding the Classic Catastrophic Computer Science Failure Mode

Ralph Johnson
Research Failures

1. Knowledge-Based Software Engineering  - no ontologies
2. Semantic Web  - no annotations
3. Software Verification  - no specifications
Research Failures

- no ontologies
- no annotations
- no specifications
Research Failures

- no specifications
Saarland Graduates

Michael Backes
TR35 in 2009

Andrej Rybalchenko
TR35 in 2010
Michael Backes

Andrej Rybalchenko

secure protocols
loop termination

hard to verify
buffer overflow
information flow
secure protocols

resource leaks
liveness
loop termination

easy to specify
hard to verify
∀i ∈ \{0, \ldots, |x'|\} : x'[i] < x'[i + 1]

|x| = |x'|

∀i ∈ \{0, \ldots, |x|\} : \forall i' ∈ \{0, \ldots, |x'|\} : x[i] = x'[i']

∀i' ∈ \{0, \ldots, |x'|\} : \forall i ∈ \{0, \ldots, |x|\} : x'[i'] = x[i]

easy to verify

hard to specify
is-sorted\( (x') \) \& is-permutation\( (x, x') \) 

still hard to specify
microsoft word
travel booking
airplane control

mobile phones
operating systems
banking systems

easy to verify
hard to specify
hard to specify

new language • duplicate effort • can’t abstract from details
- no specifications
Specification Crisis

“I think you should be more explicit here in step two.”
Mining Specifications

- <init>()
- openPort()
- auth()
- quit()

States:
- socket: null state: NOT_CON
- socket: null state: AUTH
- socket: null state: PLAIN
<init>()

socket: null
state: NOT_CON

auth()!

openPort()

socket: ¬null
state: PLAIN

quit()

socket: ¬null
state: AUTH

auth()
<init>()

socket: null
state: NOT_CON

openPort()

socket: null
state: AUTH

socket: ¬null
state: PLAIN

auth()
of

quit()
what’s the abstraction behind this?
state of the art
sqrt (x: REAL, eps: REAL): REAL is
   -- Square root of x with precision eps
require
   x >= 0 ∧ eps > 0     -- precondition
external
csqrt (x: REAL, eps: REAL): REAL
do
   Result := csqrt (x, eps)
ensure
   abs (Result ^ 2 – x) <= eps     -- postcondition
end -- sqrt
\begin{verbatim}
sqrt (x: REAL, eps: REAL): REAL is  
  -- Square root of x with precision eps
external
  csqrt (x: REAL, eps: REAL): REAL
do
  Result := csqrt (x, eps)
end  -- sqrt

- How do we infer the postcondition?
\end{verbatim}
sqrt (x: REAL, eps: REAL): REAL is
    -- Square root of x with precision eps
end -- sqrt

- How do we infer the postcondition?
Static Analysis

\[
\text{sqrt (x: REAL, eps: REAL): REAL is}
\]

\[
\quad \text{\textit{Square root of } x \text{ with precision } eps}
\]

\end

\[
\text{end \textit{-- sqrt}}
\]

- Analysis applies to \textit{all possible runs}
- Fails in presence of obscure code
- Hard to maintain precision while scaling
Dynamic Analysis

sqrt (x: REAL, eps: REAL): REAL is
  -- Square root of x with precision eps
end -- sqrt

• Analysis applies to observed runs
• Needs actual (many) executions
• Scales, but underapproximates
Daikon

sqrt (x: REAL, eps: REAL): REAL is
  -- Square root of x with precision eps
end  -- sqrt

require
  x ≥ 4 ∧ x ≤ 256 ∧ eps > 0  -- too specific

ensure
  Result * Result – x < eps  -- abs() is missing
  Result ≥ 2 ∧ Result ≤ 16  -- is this important?
Mining Behavior

We need to

• infer preconditions
• infer postconditions
• rank specifications

“Common behavior is correct behavior”
Mining Behavior

Learning behavior examples from

- real behavior
- generated behavior
- elicited behavior

“Common behavior is correct behavior”
Learning behavior examples from

- real behavior
- generated behavior
- elicited behavior

“Common behavior is correct behavior”
```c
// some info output to file header
OUTPUT_BYTE(order1);  // some info
OUTPUT_BYTE(order2);  // some info in output
OUTPUT_BYTE(order3);  // order of symbols in output
// output upper byte then upper byte
OUTPUT_BYTE(h>>BYTE_SIZE);
OUTPUT_BYTE(h);  
OUTPUT_BYTE(w>>BYTE_SIZE);
OUTPUT_BYTE(w);  

order1 = order >> 4;
order2 = order & 15;

#ifndef TRACE
    if (!fpm = fopen("ppmenc.doc", "wb")
    {
        fprintf(stderr, " \n Error: Can
        exit(2);
    }
#endif

/* allocate 'order+1' elements
order5[] is used to store
the t...*/
<init>()

socket: null
state: NOT_CON

openPort()

auth()

socket: null
state: PLAIN

quit()

socket: null
state: AUTH

auth()!
Specifications

finite state models with pre- and postconditions (for now)
socket: null
state: NOT_CON

<init>()

openPort()

auth()!

quit()

socket: ¬null
state: AUTH

socket: ¬null
state: PLAIN
"Common behavior is correct behavior"
"Common behavior is correct behavior"
for (Iterator iter = itdFields.iterator();
    iter.hasNext()); {
    ...
    for (Iterator iter2 = worthRetrying.iterator();
        iter.hasNext()); {
        ...) \textit{should be iter2}
    }
}
public String getRetentionPolicy ()
{
    ...
    for (Iterator it = ...; it.hasNext();)
    {
        ... = it.next();
        ...
    }
    return retentionPolicy;
}

should be fixed
static int dcc_listen_init (...) {
    dcc->sok = socket (...);
    if (...) {
        while (...) {
            ... = bind (dcc->sok, ...);
        }
        /* with a small port range, reUseAddr is needed */
        setsockopt (dcc->sok, ..., SO_REUSEADDR, ...);
    }
    listen (dcc->sok, ...);
}
"Common behavior is correct behavior"
checkmycode.org
powered by the software engineering chair at Saarland University

About The Tool:

- Tutorial
- Publications

Authors:

- Andrzej Wasylkowski
- Andreas Zeller

Contact:

- SE chair at Saarland University
- feedback@checkmycode.org

ABOUT CHECKMYCODE.ORG

checkmycode.org is a service that allows you to compare your code with the "wisdom of the crowds" — over 200 million lines of C code from the Gentoo Linux distribution. All anomalies found in your code will be reported to you along with an explanation of why parts of your code are considered anomalous — and therefore possibly buggy. For details about the technique, please consult our publications.

Note that we only store your code during THIS analysis process. Your submission will be analyzed, the results will be presented to you and any code will be deleted from our server afterwards.

>> run analysis
Your result is now ready. Please click on a violation (red marked line) to see a detailed description for the violation.

```c
void foo (void)
{
    struct sockaddr_in SAddr;
    int sok;
    socklen_t len;

    sok = socket (AF_INET, SOCK_STREAM, 0);
    memset (&SAddr, 0, sizeof (struct sockaddr_in));
    SAddr.sin_family = AF_INET;
    SAddr.sin_addr.s_addr = ip;
    SAddr.sin_port = htons (port);
    bind (sok, (struct sockaddr *)&SAddr, sizeof (SAddr));

    len = 1;
    setsockopt (sok, SOL_SOCKET, SO_REUSEADDR, (char *)&len, sizeof (len));
    listen (sok, 1);
}
```
Mining Behavior

Learning behavior examples from

• real behavior
• generated behavior
• elicited behavior

“Common behavior is correct behavior”
Mining Behavior

Learning behavior examples from

• real behavior

• generated behavior

• elicited behavior

“Common behavior is correct behavior”
Enriching specifications

void ProtocolTest() {
    Protocol p = new ...
    p.conn();
    p.send(x);
    p.quit();
}

Dallmeier et al: “Generating Test Cases for Specification Mining”, ISSTA 2010
void ProtocolTest() {
    Protocol p = new ...
    p.conn();
    p.send(x);
    p.quit();
}

void TestMutant1() {
    Protocol p = new ...
    p.conn();
    p.send(x);
    p.conn();
    p.quit();
}

SMTPProtocol

Dallmeier et al: “Generating Test Cases for Specification Mining”, ISSTA 2010
Dallmeier et al: “Generating Test Cases for Specification Mining”, ISSTA 2010
Generate test cases
to systematically
explore specification

Assess executions
to learn about
software behavior

Dallmeier et al: “Generating Test Cases for Specification Mining”, ISSTA 2010
Evaluation

Test subject

TAUTOKO typestate miner

Initial typestate

Enriched typestate

Application using subject

Mutated application

JFTA static typestate verifier

Error reports
Do enriched specs contain more information?

Enriched specs have more regular and exceptional transitions
How effective are enriched specifications?

Enriched specs can be almost as effective as manually crafted specs

<table>
<thead>
<tr>
<th></th>
<th>SMTPProtocol</th>
<th>Signature</th>
<th>ZipOutputStream</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reported at correct location</td>
<td>total reported</td>
<td></td>
</tr>
<tr>
<td>initial</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>enriched</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>manual</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enriched specs are better suited to finding errors than initial specs
Enriched specs can be almost as effective as manually crafted specs
A new kind of Analysis

- Static analysis
- Dynamic analysis
- Experimental analysis

- 0 runs
- n given runs
- n generated runs
Mining Behavior

Learning behavior examples from

- real behavior
- generated behavior
- elicited behavior

"Common behavior is correct behavior"
Mining Behavior

Learning behavior examples from

- real behavior
- generated behavior
- elicited behavior

“Common behavior is correct behavior”
Generate test cases to systematically explore specification

Assess executions to learn about software behavior

Dallmeier et al: “Generating Test Cases for Specification Mining”, ISSTA 2010
Generate test cases to systematically explore specification

Assess executions to learn about software behavior
Eliciting Behavior

Generate test cases to systematically explore specification

Assess executions to learn about software behavior
code

initial test case
• easy to understand
• include oracles
• act as specifications
sqrt (x: REAL, eps: REAL): REAL is
    -- Square root of x with precision eps
end -- sqrt

sqrt(4, 0) = 2?
sqrt (x: REAL, eps: REAL): REAL is
  -- Square root of x with precision eps
end -- sqrt

sqrt(9, 0) = 2?

✘
sqrt (x: REAL, eps: REAL): REAL is
   -- Square root of x with precision eps
end -- sqrt

sqrt(9, 0) = 3?

✔
sqrt (x: REAL, eps: REAL): REAL is
   -- Square root of x with precision eps
end  -- sqrt

require
  x >= 0 ∧ eps > 0

sqrt(–1, 0) = ∞?

Oops!
sqrt (x: REAL, eps: REAL): REAL is
   -- Square root of x with precision eps
end -- sqrt

require
   x >= 0 ∧ eps > 0

Result ^ 2 = x?
\texttt{sqrt (x: REAL, eps: REAL): REAL is}
\begin{itemize}
\item \textit{Square root of x with precision eps}
\end{itemize}
\texttt{end -- sqrt}
\texttt{require}
\begin{align*}
\texttt{x} & \geq 0 \land \texttt{eps} > 0 \\
\texttt{ensure}
\begin{align*}
\texttt{Result} & \land 2 = \texttt{x} - \texttt{eps}
\end{align*}
\end{align*}
Generate test cases to systematically explore specification

Assess test cases to refine software behavior
Eliciting Use Cases
Mining Behavior

Learning behavior examples from

• real behavior
• generated behavior
• elicited behavior

“Common behavior is correct behavior”
Mining Specifications

Specification Crisis

Extracting Specifications

Enriching specifications

Eliciting Behavior

- Generate test cases to systematically explore specification
- Assess test cases to refine software behavior

Execute and extract initial spec
Generate test mutants and enrich specs

Dallmeier et al: "Generating Test Cases for Specification Mining"; ISSTA 2010