FQL: A Query Language for Program Testing

Joint work with Andreas Holzer, Christian Schallhart, and Helmut Veith
White Box Testing Targets

- Structural coverage (basic block, condition, decision, paths)
- Data flow coverage (def-use pairs)
- Variable valuations

- Specific program executions
- Any of these within a fragment of the program

- Independent of test case generation technique
White Box Testing Targets

- Informal
- No pre-existing formalism to describe all these coverage criteria
- Coverage analysis tools apply contradictory definitions

- Structural coverage (basic block, condition, decision, paths)
- Data flow coverage (def-use pairs)
- Variable valuations
- Specific program executions
- Any of these within a fragment of the program

- Independent of test case generation technique
### White Box Testing Targets

- Structural coverage (basic block, condition, decision, paths)
- Data flow coverage (def-use pairs)
- Variable valuations
- Specific program executions
- Any of these within a fragment of the program
- Independent of test case generation technique

- **Informal**
- No pre-existing formalism to describe all these coverage criteria
- Coverage analysis tools apply contradictory definitions

- **Ad-hoc solutions**
White Box Testing Targets

- Structural coverage (basic block, condition, decision, paths)
- Data flow coverage (def-use pairs)
- Variable valuations
- Specific program executions
- Any of these within a fragment of the program
- Independent of test case generation technique

Informal

- No pre-existing formalism to describe all these coverage criteria
- Coverage analysis tools apply contradictory definitions

Ad-hoc solutions

Database analogue!

- Easy-to-use query language building on mathematical core
- Efficient backend (CAV’08, VMCAI’09)
Related Work
Related Work

- Coverage formalizations
  - Using Z (Vilkomir, Bowen FORTEST’08)
  - Temporal logics (Hong et al. TACAS’02)
Related Work

- Coverage formalizations
  - Using Z (Vilkomir, Bowen FORTEST’08)
  - Temporal logics (Hong et al. TACAS’02)

- Test case generation techniques
  - Random testing, fuzz testing
  - Directed testing (Godefroid et al. PLDI’05)
  - Model-checking based (Beyer et al. ICSE’04)
Related Work

- Coverage formalizations
  - Using Z (Vilkomir, Bowen FORTEST’08)
  - Temporal logics (Hong et al. TACAS’02)

- Test case generation techniques
  - Random testing, fuzz testing
  - Directed testing (Godefroid et al. PLDI’05)
  - Model-checking based (Beyer et al. ICSE’04)
Related Work

- Coverage formalizations
  - Using Z (Vilkomir, Bowen FORTEST’08)
  - Temporal logics (Hong et al. TACAS’02)

- Test case generation techniques
  - Random testing, fuzz testing
  - Directed testing (Godefroid et al. PLDI’05)
  - Model-checking based (Beyer et al. ICSE’04)

- BLAST query language (Beyer et al. SAS’04)
Test Case Generation Using Model Checkers

- Program under test
- Safety property
- Model checker
- Counterexample trace
- Test case
Test Case Generation Using Model Checkers

- Naive approach requires manual annotation of assertions
Test Case Generation Using Model Checkers

- Naive approach requires manual annotation of assertions
- MC neither tailored for coverage nor efficient enumeration
Test Case Generation Using Model Checkers

- Naive approach requires manual annotation of assertions
- MC neither tailored for coverage nor efficient enumeration
- BLAST 2.0:
  - First automated test suite generation using model checkers
  - Basic block coverage hard coded
Test Case Generation Using Model Checkers

- Naive approach requires manual annotation of assertions
- MC neither tailored for coverage nor efficient enumeration

**BLAST 2.0:**
- First automated test suite generation using model checkers
- Basic block coverage hard coded

**BLAST query language**
- Convenient specification of reachability queries
- Description of paths requires observer automata (distinct formalism)
- Path coverage cannot be expressed easily
FQL: FShell Query Language

- FShell: Test case generation based on CBMC code
- Shell-like interface to state FQL queries
- Full ANSI-C support
- Efficient SAT-based enumeration of test cases using incremental constraint strengthening (VMCAI’09)
Program Representation: Control Flow Automaton (CFA)

```c
int test(int a, int b, int c, int d)
{
    if ((a || b) && c) d = 0;
    else unimplemented();
    return d*2;
}
```
Program Representation: Control Flow Automaton (CFA)

int test(int a, int b, int c, int d)
{
    if ((a || b) && c) d = 0;
    else unimplemented();
    return d*2;
}
1 int test(int a, int b, int c, int d)
2 {
3  if ((a || b) && c) d = 0;
4  else unimplemented();
5  return d*2;
6 }
Program Representation: Control Flow Automaton (CFA)

1 int test(int a, int b, int c, int d)
2 {
3    if ((a || b) && c) d = 0;
4    else unimplemented();
5    return d*2;
6  }

assume(!a)
assume(a)
assume(!b)
assume(b)
assume(!c)
assume(c)
Program Representation: Control Flow Automaton (CFA)

```c
1 int test(int a, int b, int c, int d)
2 {
3    if ((a || b) && c) d = 0;
4    else unimplemented();
5    return d*2;
6 }
```
Filter Functions and Target Graphs

\[ \text{assume(!a)} \]
\[ \text{assume(a)} \]
\[ \text{assume(!b)} \]
\[ \text{assume(b)} \]
\[ \text{assume(!c)} \]
\[ \text{assume(c)} \]
\[ d = 0 \]
\[ \text{unimplemented()} \]
\[ \text{return d*2} \]
Filter Functions and Target Graphs

- Testing targets

\[
\begin{align*}
\text{assume}(a) & \quad \text{assume}(!a) & \quad \text{assume}(b) \\
\text{assume}(c) & \quad \text{assume}(!c) & \quad \text{unimplemented}() \\
d = 0 & \quad \text{return } d \times 2
\end{align*}
\]
Filter Functions and Target Graphs

- Testing targets
  - cover decision edges

\[ @\text{decisionedge} \]

\[ \text{assume}(a) \]
\[ \text{assume}(b) \]
\[ \text{assume}(c) \]
\[ d = 0 \]
\[ \text{unimplemented}() \]

\[ \text{return } d^2 \]
Filter Functions and Target Graphs

- Testing targets
  - cover decision edges
    \[@\text{decisionedge}\]
  - cover condition edges
    \[@\text{conditionedge}\]
    \[@\text{conditiongraph}\]

\begin{align*}
\text{assume}(!a) \\
\text{assume}(a) \\
\text{assume}(b) \\
\text{assume}(!b) \\
\text{assume}(c) \\
\text{assume}(!c) \\
d = 0 \\
\text{unimplemented()} \\
r\text{eturn } d^*2
\end{align*}
Filter Functions and Target Graphs

- **Testing targets**
  - cover decision edges
    - @decisionedge
  - cover condition edges
    - @conditionedge
    - @conditiongraph

### Target graph

- `assume(!a)`
- `assume(a)`
- `assume(!b)`
- `assume(b)`
- `assume(!c)`
- `assume(c)`
- `d = 0`
- `unimplemented()`
- `return d*2`

### Filter function

- `d = 0`
- `unimplemented()`
- `return d*2`
Filter Functions and Target Graphs

- **Testing targets**
  - cover decision edges
    - \@decisionedge
  - cover condition edges
    - \@conditionedge
    - \@conditiongraph
  - cover line 5
    - \@line(5) or @5

```
assume(!a)
assume(a)
assume(!b)
assume(b)
assume(!c)
assume(c)
d = 0
unimplemented()
return d*2
```
Filter Functions and Target Graphs

- **Testing targets**
  - cover decision edges
    - @decisionedge
  - cover condition edges
    - @conditionedge
    - @conditiongraph
  - cover line 5
    - @line(5) or @5
  - cover function calls
    - @calls
    - @call(unimplemented)

```plaintext
assume(!a)
assume(a)
assume(b)
assume(!b)
assume(!c)
assume(c)
d = 0
unimplemented()
return d*2
```
Filter Functions and Target Graphs

- **Testing targets**
  - cover decision edges
    - @decisionedge
  - cover condition edges
    - @conditionedge
    - @conditiongraph
  - cover line 5
    - @line(5) or @5
  - cover function calls
    - @calls
    - @call(unimplementeded)

- **Interface to programming language**
Filter Functions and Target Graphs

```
assume(!a)
assume(a)
assume(!b)
assume(b)
assume(!c)
assume(c)
d = 0
unimplemented()
return d*2
```
Filter Functions and Target Graphs

- \texttt{union(@5, @decisionedge)}

- \texttt{d = 0}
- \texttt{unimplemented()}
- \texttt{return d*2}
Filter Functions and Target Graphs

- \( \text{union}(\neg a) \)
- \( \text{complement}(\neg a) \)
- \( \text{assume}(\neg a) \)
- \( \text{assume}(a) \)
- \( \text{assume}(\neg b) \)
- \( \text{assume}(b) \)
- \( \text{assume}(\neg c) \)
- \( \text{assume}(c) \)
- \( d = 0 \)
- \( \text{unimplemented}() \)
- \( \text{return } d \times 2 \)

- \( \text{union}(\neg 5, \text{decisionedge}) \)
- \( \text{complement}(\neg 5) \)
Filter Functions and Target Graphs

- `union(@5,@decisionedge)`
- `complement(@5)`
- `setminus(@decisionedge,@calls)`

`d = 0`
`
unimplemented()

return d*2`
Filter Functions and Target Graphs

- \( \text{union}(@5, \text{decisionedge}) \)
- \( \text{complement}(@5) \)
- \( \text{setminus}(@\text{decisionedge},@\text{calls}) \)
- \( \text{intersect}(@5, @\text{conditionedge}) \)

\( \text{assume}(!a) \)

\( \text{assume}(a) \)

\( \text{assume}(b) \)

\( \text{assume}(!b) \)

\( \text{assume}(c) \)

\( \text{assume}(!c) \)

\( \text{d} = 0 \)

\( \text{unimplemented}() \)

\( \text{return d*2} \)
Test Goals

- assume(!a)
- assume(!b)
- assume(c)
- assume(!c)
- d = 0

unimplemented()

return d*2
Test Goals

- cover \textbf{STATES}(@conditiongraph)
  - $a = 1, c = 1$
  - $a = 0, b = 0$

\begin{itemize}
  \item \textbf{STATES}(@conditiongraph)
  \item $a = 1, c = 1$
  \item $a = 0, b = 0$
\end{itemize}
Test Goals

- cover **STATES**(@conditiongraph)
  - \(a = 1, c = 1\)
  - \(a = 0, b = 0\)
- cover **EDGES**(@conditiongraph)
  - \(a = 1, c = 1\)
  - \(a = 0, b = 0\)
  - \(a = 0, b = 1, c = 0\)

\[
\begin{align*}
\text{assume(!a)} \\
\text{assume(b)} \\
\text{assume(!b)} \\
\text{assume(c)} \\
\text{assume(!c)} \\
d = 0 \\
\text{unimplemented()}
\end{align*}
\]

\[
\text{return } d \times 2
\]
Test Goals

- cover \textsc{STATES}(@conditiongraph)
  - $a = 1, c = 1$
  - $a = 0, b = 0$
- cover \textsc{EDGES}(@conditiongraph)
  - $a = 1, c = 1$
  - $a = 0, b = 0$
  - $a = 0, b = 1, c = 0$
- cover \textsc{PATHS}(@conditiongraph,1)
  - $a = 1, c = 1$
  - $a = 0, b = 0$
  - $a = 0, b = 1, c = 0$
  - $a = 0, b = 1, c = 1$
  - $a = 1, c = 0$

```
d = 0
unimplemented()
return d*2
```
Test Goals (continued)

- Test goals for MC/DC-style dependency coverage: DEPS
Test Goals (continued)

- Test goals for MC/DC-style dependency coverage: DEPS
- Predicated CFAs: cover
  STATES(@conditiongraph, \{d > 5\}, \{a != 0\})
Test Goals (continued)

- Test goals for MC/DC-style dependency coverage: `DEPS`
- Predicated CFAs: cover `STATES(@conditiongraph, {d > 5}, {a != 0})`
- Pre- and post-conditions: cover `{a == 5}EDGES(@3) {d != 0}`

```
d = 0
unimplemented()
return d*2
```
Path Monitors

assume(a)

assume(b)

assume(c)

\( d = 0 \)

unimplemented()

return \( d \times 2 \)
Path Monitors

- How to exclude undesired paths?
- Describe specific executions?

```
assume(!a)
assume(a)
assume(!b)
assume(b)
assume(!c)
assume(c)
d = 0
unimplemented()
return d*2
```
Path Monitors

- How to exclude undesired paths?
- Describe specific executions?
- Regular expressions over CFA edges, use filter functions!

```
d = 0
assert(!a)
assert(!b)
assert(!c)
unimplemented()
return d*2
```
Path Monitors

- How to exclude undesired paths?
- Describe specific executions?
- Regular expressions over CFA edges, use filter functions!

- COMPLEMENT(@call(unimplemented))

\[
\text{d = 0} \\
\text{unimplemented()} \\
\text{return } d^*2
\]
Path Monitors

- How to exclude undesired paths?
- Describe specific executions?
- Regular expressions over CFA edges, use filter functions!

\[ \text{COMPLEMENT}(@\text{call(unimplemented)})^* \]
Path Monitors

- How to exclude undesired paths?
- Describe specific executions?
- Regular expressions over CFA edges, use filter functions!

- $\text{COMPLEMENT(}@\text{call(unimplemented)}\text{)}^*$
- $(\text{id*}.@\text{call(insert)}.\text{id*}) \geq 10. \ @\text{call(sort)} . \ \text{id}^*$
Coverage Sequences

- Cover all pairs of conditions in insert and sort:
  - `cover EDGES(intersect(@basicblockentry, @func(insert))) -> EDGES(intersect(@basicblockentry, @func(sort)))`

- Cartesian product of test goals

- Intermediate paths restricted using path monitors
  - `cover EDGES(@call(partition))`
  - `-[COMPLEMENT(@exit(partition))*]> EDGES(INTERSECT(@func(partition), @conditionedge))`
FQL Queries

- Condition coverage in function “compare” with test cases which call “compare” from inside function “sort” only
  - cover EDGES(INTERSECT(@CONDITIONEDGE,@FUNC(compare)))
    passing (COMPLEMENT(@CALL(compare)) + INTERSECT(@CALL(compare),@FUNC(sort))) *

- Cover all basic blocks in function “mainloop” while never calling “init”
  - cover EDGES(INTERSECT(@BASICBLOCKENTRY,@FUNC(mainloop)))
    passing COMPLEMENT(@CALL(init)) *

- Basic block coverage in function “sort” and require each test case to use a list of at least five but at most 15 elements
  - in @FUNC(sort) cover EDGES(@ENTRY(sort)){"len>=5 & & len<=15"}
    - [COMPLEMENT(@EXIT(sort))] > EDGES(@BASICBLOCKENTRY)
Current Work

- Finalizing FQL (dependency coverage)
- Completing FShell
- Backend using Configurable Program Analysis
- Eclipse plug-in
- DO-178B compatible test case generation: Slope Testing