A Model Checking Perspective on White-Box Testing

Helmut Veith
Technische Universität Wien

veith@forsyte.at
Students and Collaborators

Andreas Holzer (Toronto)
Michael Tautschnig (Queen Mary)
Christian Schallhart (Google)
Azadeh Farzan (Toronto)
Niloofar Razavi (Toronto)
Visar Januzaj (Darmstadt)
Stefan Kugele (Munich)
Boris Langer (Diehl Aerospace)
Raimund Kirner (Hertfordshire)

DFG / FWF
RiSE
National Research Network (FWF)
Main Publications on Testing 2008-2013

Dirk Beyer, Andreas Holzer, Michael Tautschnig, Helmut Veith: **Information Reuse for Multi-goal Reachability Analyses.** ESOP 2013: 472-491


Azadeh Farzan, Andreas Holzer, Niloofar Razavi, Helmut Veith: **Con2colic testing.** ESEC/SIGSOFT FSE 2013: 37-47


Andreas Holzer, Michael Tautschnig, Christian Schallhart, Helmut Veith: **An Introduction to Test Specification in FQL.** Haifa Verification Conference 2010: 9-22


Andreas Holzer, Christian Schallhart, Michael Tautschnig, Helmut Veith: **Query-Driven Program Testing.** VMCAI 2009: 151-166

Model Checking and Testing

Theoretical Background
How we came to work on Testing
Undecidability (of Verification)
Incompleteness.
Non-elementary complexity.
NP-completeness.
Turing's Quote on Program Verification

“How can one check a routine in the sense of making sure that it is right?”

“The programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.”

Quote by A. M. Turing on 24 June 1949 at the inaugural conference of the EDSAC computer at the Mathematical Laboratory, Cambridge.
“the first open admission of the software crisis.”

(Dijkstra, The Humble Programmer)
The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.

By definition this approach is only applicable when we restrict ourselves to intellectually manageable programs.
Limitations of Human Reasoning

Lines of Code in Modern Computer Programs

Expected Errors per 10,000 Lines = 500m
250 Errors (typical software)
20 Errors (good software)
1 Error (space shuttle quality)
Software Model Checking

Critical property

Model checking

Program

Compilation

Executable

Property violations documented by program traces! I know a bug when I see it.
Software Model Checking

Critical property
model checking
program
Compilation
Executable

Property violations documented by program traces! I know a bug when I see it.
Software Model Checking Paradigms

- **Predicate abstraction**
  - overapproximation of the state space
  
  Formal evidence for unreachability, spurious counterexamples due to abstract semantics
  e.g. SLAM (MSR), BLAST (Berkeley), CPA (Passau)

- **Bounded Model Checking**
  - underapproximation of the state space
  
  Formal evidence for reachability, precise semantics, bounded size counterexamples e.g. CBMC (Kröning)

*de facto combined using SAT / SMT solvers*
Abstract memory states are formulas describing properties of the memory content.

- Decision procedures, SAT solver, SMT, ...
- **Need for classical logic!**
CEGAR (Counterexample-Guided Abstraction Refinement)

Adaptive Strategy
CEGAR (Counterexample-Guided Abstraction Refinement)

Adaptive Strategy

Initial Abstraction Function

Refinement required.

Check if counterexample is feasible: SAT / SMT solver
Refinement: Craig Interpolation
CEGAR (Counterexample-Guided Abstraction Refinement)

Adaptive Strategy

Check if counterexample is feasible: SAT / SMT solver
Refinement: Craig Interpolation
SAT/SMT for path feasibility

i. Choose a program path
ii. Convert to single static assignment form
iii. Replace if-then-else by assume:

if \((x > 5)\) then \(A\)  \hspace{1cm} \text{assume}(\neg(x>5));
else \(B\) \hspace{1cm} \(B\);

iv. Extract a formula representing the path
\(x_1=x_0+5; \ \text{assume}(\neg(x_1>5)); \ x_2 = x_1-5;\)
\(\text{(x1=x0+5) \& \neg(x1>5) \& (x2 = x1-5)}\)

v. Logical satisfiability of the formula = feasibility of the path

Idealizing assumption: SMT is a reliable oracle.
2000s: development of industrial strength C model checkers

“rivals theorem proving for many verification tasks” (Rushby)

→ Microsoft product for Windows device driver verification
“device drivers we’re building tools that do actual proofs about the software and how it works in order to guarantee the reliability.”
“Model checking is an acceptable crutch.”
“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”
“device drivers” we’re building tools that do actual proofs about the software and how it works in order to guarantee the reliability.”
MC specification: no assertions are violated

$$AG(pc=l \rightarrow assertion)$$

Program rewrite

```plaintext
assert(F)  if (!F) goto err;
```

Testing for coverage of err (or basic block coverage)

test case covering err = counterexample

Disadvantage

- assumes perfect test case generation
- similar to perfect oracle for path feasibility
“The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise. We are not there yet.”
Execution time analysis in a white box setting

- C source code
- Focus on automatically generated code

Abstracts from platform

Execution times obtained through measurements

Requires large data sets, possibly with code coverage

Expected Time System
Test goal: Cover line 4242 of the program.

Model Checking Specification: $AG(pc \neq 4242)$

Property correct: line 4242 is dead code
Counterexample: *trace leading to line 4242*

**Disadvantages**
- one model checking call per test goal
- redundant calls
- does not scale to large programs
- no support for coverage criteria beyond simple test goals
Precision of Coverage Criteria

„Condition Coverage“
cover all program conditions

1 void foo( int x) {
2    int a = x > 2 && x < 5;
3    if (a) { 0; } else { 1; }
4 }  

There is no general purpose formalism for white box test case specification!

Commercial Tools
Coverage Meter, CTC++
BullseyeCoverage

SOFTWARE CONSIDERATIONS IN AIRBORNE SYSTEMS AND EQUIPMENT CERTIFICATION

DOCUMENT NO. RTCA/DO-178B
December 1, 1992
Prepared by: SC-167

Condition coverage?

100% coverage

BullseyeCoverage
83% coverage
Is there a systematic way to specify coverage criteria and leverage model checking for test case generation?

Query-Driven Test Case Generation
Query-Driven Program Testing

Programs as Databases

Query Coverage Criterion

FQL Query

Test Input Generator

FShell CPA-Tiger

Database Engine

Test Suite

Query Result

Database Program

C Source Code

Vision
SQL for Test Specifications

Separation of Concerns
- clean semantics
- multiple engines
- query dispatcher
- stable query interface
Query-Driven Test Case Generation

I. Test Specification Language FQL

II. Test Case Generation Backends
   a. FShell: Based on CBMC / SAT
   b. CPA-Tiger: Based on CPA / abstraction

III. FQL Theoretical Background
FQL Design Challenge

Usage Scenarios

- **Test Case Generation**
  
  (generic and ad hoc coverage criteria)

- **Systematic Reasoning about Test Specifications**
  
  (Optimization, Subsumption etc.) cf. database theory

- **Certification & Coverage Evaluation**
  
  e.g. measure coverage achieved by existing test suite

- **Requirement-Driven Testing**
  
  translate requirements into FQL
FQL Design Challenge
Language Design Principles

Precise Semantics

Expressive Power
small number of orthogonal concepts suffice to express large classes of specifications

Simplicity and Code Independence
tool for the working programmer
simple specs easily expressible
relative stability during code refactoring

Encapsulation of Language Specifics
easily adaptable to a large class of imperative programming languages

Tool Support for Real World Code
test case generation engines

SQL/Database Analogy
FQL Design Challenge

More Language Desiderata

FQL should capture

- Syntax of the program
- Semantics of the program
- Reasonably language independent

User friendly:

- Easy to write
- Easy to understand
- Natural to use
- Predictable performance

Logic and Algorithms

- High expressive power
- Tractable to evaluate
FQL Challenge
Example: Basic Block Coverage

„for each basic block in the program there is a test case in the test suite which covers the basic block“

1. Specifies a test suite, i.e., multiple test cases
2. Contains a universal quantifier
3. Assumes knowledge about programs. What IS a basic block for a logic?
4. Has a meaning independent of the program under test. Can be translated into concrete specifications for a fixed program.
FQL

Program Executions as Regular Expressions

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```
FQL

Program Executions as Regular Expressions

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```
FQL

Program Executions as Regular Expressions

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```
Program Executions as Regular Expressions

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```
Program Executions as Regular Expressions

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```

Line 1
Line 2
Line 3
Line 4
Line 5
Line 8
Line 9
Line 10
Program Executions as Regular Expressions

```c
1 int max(int x, int y) {
2    int tmp;
3    if (x >= y)
4        tmp = x;
5    else
6        tmp = y;
7    return tmp;
8 }
9
```
Program Executions as Regular Expressions

```c
1 int max(int x, int y) {
2     int tmp;
3
4     if (x >= y)
5         tmp = x;
6     else
7         tmp = y;
8
9     return tmp;
10 }
```
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
FQL

Program Executions as Regular Expressions

```c
1 int max(int x, int y) {
2   int tmp;
3
4   if (x >= y)
5     tmp = x;
6   else
7     tmp = y;
8
9   return tmp;
10 }
```
Program Executions as Regular Expressions

1  int max(int x, int y) {
2      int tmp;
3
4      if (x >= y)
5          tmp = x;
6      else
7          tmp = y;
8
9      return tmp;
10  }

Program Executions as Regular Expressions

- Several Paths

```c
1 int max(int x, int y) {
2     int tmp;
3
4     if (x >= y) {
5         tmp = x;
6     } else {
7         tmp = y;
8     }
9     return tmp;
10 }
```
Program Executions as Regular Expressions

- Several Paths

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```
FQL

Program Executions as Regular Expressions

- Several Paths
- Language: Set of program executions

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```

Several Paths

Language: Set of program executions

Test goals are also sets of program executions

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```

In practice, the alphabet is more complex than line numbers.
Several Paths

Language: Set of program executions

Test goals are also sets of program executions

Practical test goals can be expressed using regular languages

How to express sets of test goals?

In practice, the alphabet is more complex than line numbers.
FShell Query Language (FQL)

Quoted Regular Expressions

$A_1 = \{a^*b.a^* + c.d*\}$

$\mathcal{L}(A_1) = \{b, ab, ba, aba, aab, aaba, \ldots, c, cd, cdd, \ldots\}$

Infinite number of specific paths

$A_2 = \{a^*.b.a^* + c.d\}$

$\mathcal{L}(A_2) = \{a^*.b.a^* + c.d^*\}$

One test goal

$A_3 = \{a^*.b.a^*\}$

$\mathcal{L}(A_3) = \{a^*.b.a^*, c.d^*\}$

Two test goals
Filter Functions

```c
int max(int x, int y) {
    int tmp;

    if (x >= y)
        tmp = x;
    else
        tmp = y;

    return tmp;
}
```
Filter Functions: knowledge about programs

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```
Filter Functions: knowledge about programs

- @ID
- @BASICBLOKENTRY

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```

Line 2 + Line 5 + Line 7 + Line 9
Filter Functions: knowledge about programs

- @ID
- @BASICBLOCKENTRY
- @ENTRY

```c
1 int max(int x, int y) {
2   int tmp;
3   
4   if (x >= y)
5     tmp = x;
6   else
7     tmp = y;
8   return tmp;
9 }
```

Line 1
Filter Functions: knowledge about programs

- @ID
- @BASICBLOCKENTRY
- @ENTRY
- @EXIT

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```

Line 10
Filter Functions: knowledge about programs

- @ID
- @BASICBLOCKENTRY
- @ENTRY
- @EXIT
- @LINE(7)

1 int max(int x, int y) {
2    int tmp;
3
4    if (x >= y)
5        tmp = x;
6    else
7        tmp = y;
8
9    return tmp;
10 }

Line 7
Filter Functions: knowledge about programs

- @ID
- @BASICBLOCKENTRY
- @ENTRY
- @EXIT
- @LINE(7)
- ...

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```
Filter Functions: knowledge about programs

- @ID
- @BASICBLOCKENTRY
- @ENTRY
- @EXIT
- @LINE(7)
- ...

Filter functions can be combined:

```c
int max(int x, int y) {
    int tmp;
    if (x >= y)
        tmp = x;
    else
        tmp = y;
    return tmp;
}
```
Filter Functions: knowledge about programs

- @ID
- @BASICBLOCKENTRY
- @ENTRY
- @EXIT
- @LINE(7)
- ...

Filter functions can be combined:

- @BASICBLOCKENTRY(@FUNCTION(f))
- @BASICBLOCKENTRY(@FUNCTION(f) | @FUNCTION(g))
- ...

```c
1 int max(int x, int y) {
2     int tmp;
3     if (x >= y)
4         tmp = x;
5     else
6         tmp = y;
7     return tmp;
8 }
```
Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block”
“for each basic block in the program there is a test case in the test suite which covers the basic block”
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) `cover "@ID*" .@LINE(8) "\@ID*"`

b) `cover "@ID*" .@LINE(10) "\@ID*"`

C Source Code

```c
1 if (x > 10) f1 = false;
2 else f1 = true;
3 if (x == 100) f2 = false;
4 if (f1) s = f2;
5 else s = f1;
```

Test Suite

a) `x = 10`

b) `x = 11`
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) \textbf{cover} "@ID*".\textbf{LINE}(8)."@ID*"

b) \textbf{cover} "@ID*".\textbf{LINE}(10)."@ID*"

...
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) \text{cover} "@ID*" . @LINE(8) . "@ID*"

b) \text{cover} "@ID*" . @LINE(10) . "@ID*"

...
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) cover "@ID*". @LINE(8). @ID*

b) cover "@ID*". @LINE(10). @ID*

...
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) \textbf{cover} \texttt{"@ID*".@LINE(8)."@ID*"}

b) \textbf{cover} \texttt{"@ID*".@LINE(10)."@ID*"}

...
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) cover "@ID*" @LINE(8) "@ID*"

b) cover "@ID*" @LINE(10) "@ID*"

...
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) \text{cover} \text{ "@ID*.@LINE(8)."@ID*"}

b) \text{cover} \text{ "@ID*.@LINE(10)."@ID*"}

...
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

a) cover "@ID*".@LINE(8)."@ID*"

b) cover "@ID*".@LINE(10)."@ID*"

...
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL allows the Kleene-star only inside of quotes!

cover "@ID*".(@LINE(8) + @LINE(10) + ...)."@ID*"

VMCAI’09, ASE’10, HVC’10: Holzer, Schallhart, Tautschnig, Veith
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

a) \texttt{cover \textasciitilde@ID\~.\textasciitilde@LINE(8)\~.\textasciitilde@ID\~} 

b) \texttt{cover \textasciitilde@ID\~.\textasciitilde@LINE(10)\~.\textasciitilde@ID\~}

\[\texttt{cover \textasciitilde@ID\~.(@LINE(8) + @LINE(10) + \ldots).\textasciitilde@ID\~}\]
Coverage Criteria as FQL Queries

Do we have to express a coverage criterion for each program individually?

a) cover "@ID*".@LINE(8)."@ID*"

b) cover "@ID*".@LINE(10)."@ID*"

... 

cover "@ID*".(@LINE(8)+@LINE(10)+...)."@ID*"
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

Do we have to express a coverage criterion for each program individually?

a) \text{cover} \quad "@ID\ast" \cdot \text{LINE}(8) \cdot "@ID\ast"

b) \text{cover} \quad "@ID\ast" \cdot \text{LINE}(10) \cdot "@ID\ast"

... 

\text{cover} \quad "@ID\ast" \cdot (\text{LINE}(8) + \text{LINE}(10) + \ldots) \cdot "@ID\ast"
FShell Query Language (FQL)

Filter Functions

1. if (x > 10)
2.   f1 = false;
3. else
4.   f1 = true;
5. if (x == 100)
6.   f2 = false;
7. if (f1)
8.   s = f2;
9. else
10.  s = f1;
FShell Query Language (FQL)

Filter Functions Revisited

- @ID

```plaintext
1 if (x > 10)
2 f1 = false;
3 else
4 f1 = true;
5 if (x == 100)
6 f2 = false;
7 if (f1)
8 s = f2;
9 else
10 s = f1;
```
FShell Query Language (FQL)

Filter Functions Revisited

- @ID
- @LINE(8)

```
1 if (x > 10)
2   f1 = false;
3 else
4   f1 = true;
5 if (x == 100)
6   f2 = false;
7 if (f1)
8   s = f2;
9 else
10  s = f1;
```

Line 8
FShell Query Language (FQL)

Filter Functions Revisited

- @ID
- @LINE(8)
- NOT(@LINE(8))

```java
if (x > 10)
    f1 = false;
else
    f1 = true;
if (x == 100)
    f2 = false;
if (f1)
    s = f2;
else
    s = f1;
```

Line 1 + Line 2 + Line 3 + Line 4 + Line 5 + Line 6 + Line 7 + Line 9 + Line 10
FShell Query Language (FQL)

Filter Functions Revisited

- @ID
- @LINE(8)
- NOT(@LINE(8))
- @BASICBLOCKENTRY

```java
1  if (x > 10)
2    f1 = false;
3  else
4    f1 = true;
5  if (x == 100)
6    f2 = false;
7  if (f1)
8    s = f2;
9  else
10   s = f1;
```

Line 2 + Line 4 + Line 6 + Line 8 + Line 10
FShell Query Language (FQL)

Filter Functions Revisited

- `@ID`
- `@LINE(8)`
- `NOT(@LINE(8))`
- `@BASICBLOCKENTRY`
- ...

```java
if (x > 10)
    f1 = false;
else
    f1 = true;
if (x == 100)
    f2 = false;
if (f1)
    s = f2;
else
    s = f1;
```
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) cover "@ID*". @LINE(8). "@ID*"

b) cover "@ID*". @LINE(10). "@ID*"

cover "@ID*". (@LINE(8) + @LINE(10) + ...). "@ID*"
FShell Query Language (FQL)

Coverage Criteria as FQL Queries

“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) \texttt{cover} "@ID*".\texttt{LINE}(8)."@ID*"

b) \texttt{cover} "@ID*".\texttt{LINE}(10)."@ID*"

\texttt{cover} "@ID*".\texttt{BASICBLOCKENTRY}."@ID*"
FShell Query Language (FQL)

Passing Clauses in Coverage Criteria

```plaintext
cover "@ID*".@BASICBLOCKENTRY."@ID*
passing "@ID*.NOT(@FUNCTION(unimplemented)).@ID*"
```
Simple Coverage Criteria

„Block Coverage“
cover all program blocks
cover @BASICBLOCKENTRY

„Condition Coverage“
cover all program conditions
cover @CONDITIONEDGE
cover EDGES(INTERSECT(@CONDITIONEDGE, @STMTTYPE(if,switch,for,while,?:))))
“Restricted Scope of Analysis”
Condition coverage in function partition with test cases that reach line 7 at least once.

in @FUNC(partition) cover @CONDITIONEDGE passing @7

“Condition/Decision Coverage”
Condition/decision coverage (the union of condition and decision coverage)

cover @CONDITIONEDGE + @DECISIONEDGE

“Interaction Coverage”
Cover all possible pairs between conditions in function sort and basic blocks in function eval, i.e., cover all possible interactions between sort and eval.

cover (@CONDITIONEDGE & @FUNC(sort)) . “ID*“ . (@BASICBLOCKENTRY & @FUNC(eval))

“Cartesian Block Coverage”
Cover all pairs of basic blocks in function partition.

cover @BASICBLOCKENTRY. “ID*“ . @BASICBLOCKENTRY
“Constrained Program Paths”
Basic block coverage with test cases that satisfy the assertion $j > 0$
before executing line 3.

cover @BASICBLOCkENTRY passing @LINE(2) .$j>0$

“Constrained Inputs”
Basic block coverage in function sort with test cases that use a list
with 2 to 15 elements.

cover @ENTRY(sort).{len>=2}.{len<=15}.
.“NOT(@EXIT(sort))“. 
@BASICBLOCkENTRY

“Recursion Depth”
Cover function eval with condition coverage and require each test
case to perform three recursive invocations of eval.

in @FUNC(eval) cover @CONDITIONEDGE
passing @CALL(eval).NOT(@EXIT(eval))*.@CALL(eval)
 .NOT(@EXIT(eval))*.@CALL(eval)
Complex Coverage Criteria

“Acyclic Path Coverage”
Cover all acyclic paths through functions main and insert.
cover PATHS(@FUNC(main) | @FUNC(insert), 1)

“Loop-Bounded Path Coverage”
Cover all paths through main and insert which pass each statement at most twice.
cover @DEF(t)

“Def Coverage”
Cover all statements defining variable t.
cover PATHS(@FUNC(main) | @FUNC(insert), 2)

“Use Coverage”
Cover all statements which use variable t as right hand side value.
cover @USE(t)

“Def-Use Coverage”
Cover all def-use pairs of variable t.
cover @DEF(t) . “NOT(@DEF(t))*” . @USE(t)
Query-Driven Test Case Generation

I. Test Specification Language FQL

II. Test Case Generation Backends
   a. FShell: Based on CBMC / SAT
   b. CPA-Tiger: Based on CPA / abstraction

III. FQL Theoretical Background
Query-Driven Program Testing

Programs as Databases

Query

Coverage Criterion

FQL Query

Query Result

Database Engine

Test Input Generator

C Source Code

Test Suite

Database

Program

FShell CPAtiger
Query-Driven Program Testing

FShell
- Bounded Model Checking
- Loop Bounds
- Can’t show non-existence of test case

CPAtiger
- Predicate Abstraction
- No Loop Bounds
- Proves existence and non-existence of test cases
FShell: Approach

**Query:** test specification

```
COVER @basicblockentry
```

**Step 1: Program Instrumentation**
- Add monitoring layer to C program + query specific monitor
- Encode *all* test goals into monitor

**Step 2: Test Case Generation**
- Use Kroening’s CBMC + Guided SAT Enumeration for Efficient Enumeration of Test Cases
**Background: Kröning’s CBMC**

*C Bounded Model Checker*

Clean and stable code base

Full ANSI-C support

SAT solver = decision procedure
FShell: Architecture

- Reuses parts of CBMC
- Interactive command-line frontend
Iterative Constraint Strengthening for Fast Test Case Generation

- Fast computation of solutions by SAT enumeration
- Incremental SAT solving
  - Clause database is updated on-the-fly
  - SAT solver suspended during database update by FShell
  - Conflict database is kept and reused
- Instance becomes unsatisfiable iff remaining goals infeasible
- Complex coverage criteria:
  groupwise constraint strengthening
Iterative Constraint Strengthening

Program + monitors described by CNF formula \( \Phi[\Pi_A^T] \)

Example test goals \( \{\Psi_1, \Psi_2, \Psi_3, \Psi_4\} \)

Initial constraint: “reach some test goal”

\[
\text{ICSPC}_0 = \Phi[\Pi_A^T] \land \bigvee_{a=1\ldots 4} (S_a \land \Phi[\Psi_a])
\]

Assume \( \Psi_1 \) and \( \Psi_3 \) are satisfied by first solution: “reach new test goal”

\[
\text{ICSPC}_1 = \text{ICSPC}_0 \land \neg S_1 \land \neg S_3
\]
Query-Driven Program Testing

**FShell**
- Bounded Model Checking
- Loop Bounds
- Can’t show non-existence of test case

**CPAtiger**
- Predicate Abstraction
- No Loop Bounds
- Proves existence and non-existence of test cases
Repeated Invocation of an Automaton-guided Reachability Analysis

For real-world programs the number of automata becomes huge!

How can we reuse analysis results across different automata?

Model Checker: Is there a program execution that is accepted by the automaton?

FQL Query

Test Input Generator

Automata $A_1, A_2, ..., A_n$

Test Inputs & Proofs of Non-existence

C Source Code

[ESOP’13] Beyer, Holzer, Tautschnig, Veith
For this automaton we ran a reachability analysis.
Information Reuse

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
For this automaton we want to run a reachability analysis.
For this automaton we want to run a reachability analysis.

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
For this automaton we want to run a reachability analysis.

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
For this automaton we want to run a reachability analysis

Reuse all reachability infos!
Information Reuse

Simulation Relation

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
Information Reuse

Simulation Relation

“for each transition in the second automaton, we find a corresponding transition in the first automaton”
Information Reuse

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
Information Reuse

\[ p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \]

\[ q_0 \rightarrow q_1 \rightarrow q_2 \]

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
Information Reuse

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
Information Reuse

$p_0$ 1,2,5,6,7, 8,10 4
$p_1$ 1,2,4,5,7, 8,10 6
$p_2$ 8,10 7
$p_3$ 8,10

$q_0$ 1,2,5,6,7, 8,10 4
$q_1$ 1,2,4,5,7, 8,10 6
$q_2$ 8,10 7

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
Information Reuse

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
Information Reuse

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
Information Reuse

Simulation Relation modulo \{ (q_2, 7, q_2) \}

[ESOP'13] Beyer, Holzer, Tautschnig, Veith
CPAtiger

- Based on Dirk Beyer´s SW model checker CPAchecker

- Experiments in Holzer´s thesis
  - Windows NT Drivers
  - Variants of Basic Block Coverage:
    - $BB$: Cover each basic block
    - $BB^2$: Cover each pair of basic blocks
    - $BB^3$: Cover each triple of basic blocks
  - Bounded-Path Coverage

[ESOP’13] Beyer, Holzer, Tautschnig, Veith
Experiments ($BB^2$ Coverage)

Improvements over naive iteration approach

- Time-out (>15000s)

[ESOP’13] Beyer, Holzer, Tautschnig, Veith
Query-Driven Test Case Generation

I. Test Specification Language FQL

II. Test Case Generation Backends
   a. FShell: Based on CBMC / SAT
   b. CPA-Tiger: Based on CPA / abstraction

III. FQL Theoretical Background
\{ \Sigma^* \cdot \{a\} \cdot \Sigma^* \ , \\
\Sigma^* \cdot \{b\} \cdot \Sigma^* \ , \\
\Sigma^* \cdot \{c\} \cdot \Sigma^* \}
Regular Sets of Rational Languages (RSRL)
[Afonin and Khazova, 2005]

\[
\{ \Sigma^* \cdot \{a\} \cdot \Sigma^* , \\
\Sigma^* \cdot \{b\} \cdot \Sigma^* , \\
\Sigma^* \cdot \{c\} \cdot \Sigma^* \}
\]
Regular Sets of Rational Languages (RSRL)
[Afonin and Khazova, 2005]

\[
K = \{ \delta_1 \delta_2 \delta_1 , \delta_1 \delta_3 \delta_1 , \delta_1 \delta_4 \delta_1 \}
\]

\[
\{ \Sigma^* \cdot \{a\} \cdot \Sigma^* , \Sigma^* \cdot \{b\} \cdot \Sigma^* , \Sigma^* \cdot \{c\} \cdot \Sigma^* \}
\]
Semantic Foundations of FQL

Regular Sets of Rational Languages (RSRL)
[Afonin and Khazova, 2005]

\[
K = \{ \delta_1 \delta_2 \delta_1 , \delta_1 \delta_3 \delta_1 , \delta_1 \delta_4 \delta_1 \} \\
\phi(\delta_1) = \Sigma^* \\
\phi(\delta_2) = \{a\} \\
\phi(\delta_3) = \{b\} \\
\phi(\delta_4) = \{c\}
\]

\(\phi\) maps to a regular language

\[
\{ \Sigma^* \cdot \{a\} \cdot \Sigma^* , \Sigma^* \cdot \{b\} \cdot \Sigma^* , \Sigma^* \cdot \{c\} \cdot \Sigma^* \} \\
\equiv (K, \phi)
\]
The complement of an RSRL is **not** an RSRL.

RSRL are closed under product, Kleene star, union.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Finite Case (FQL)</th>
<th>Finite Case, fixed $\varphi$ (FQL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\cdot, \cup, \cap, -$</td>
<td>Closed</td>
<td>Not closed</td>
</tr>
</tbody>
</table>
Closure Properties 2

Point-wise Operators

cover "@ID*".@BASICBLOCKENTRY."@ID*"

passing @ID*.NOT(@FUNCTION(unimplemented)).@ID*

\[ \mathcal{R} \cap R = \{ L \cap R \mid L \in \mathcal{R} \} \]

<table>
<thead>
<tr>
<th>Point-wise Operators</th>
<th>Finite RSRL, fixed ( \varphi ) (FQL)</th>
<th>Finite RSRL (FQL)</th>
<th>RSRL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( *, \oplus, \neg, \cap, \cup, \ominus )</td>
<td>Not closed</td>
<td>Closed</td>
<td>Not Closed</td>
</tr>
</tbody>
</table>
### Complexities of Decision Problems

<table>
<thead>
<tr>
<th>Decision Problem</th>
<th>Kleene-star free case (FQL)</th>
<th>General case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalence</td>
<td>$\mathcal{R}_1 = \mathcal{R}_2$</td>
<td>PSPACE-complete</td>
</tr>
<tr>
<td>Inclusion</td>
<td>$\mathcal{R}_1 \subseteq \mathcal{R}_2$</td>
<td>PSPACE-complete</td>
</tr>
<tr>
<td>Membership</td>
<td>$L \in \mathcal{R}$</td>
<td>PSPACE-complete</td>
</tr>
</tbody>
</table>
Further Work

Model-Based Testing with FQL

Case Studies
Automotive, Avionic

Con2colic testing
Extension of concolic testing to systematically explore inputs and thread interference
→ Proceedings

Testing for Distributed Algorithms
Systems with vast non-determinism
Conclusion

FQL is an automata-based framework for specification of coverage criteria.

- Simple well-understood semantics
- Based on quoted regular expressions
- Separation between test specification and test case generation engine
- Easy to use for non-specialists
- Prototype implementations based on CBMC and CPA.