A Model Checking Perspective on White-Box Testing

Helmut Veith
Technische Universität Wien

veith@forsyte.at

for(syte
Formal Methods
in Systems Engineering

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)

FoRTAS **DFG / FWF**

RiSE
Rigorous Systems Engineering

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

Andreas Holzer, Christian Schallhart, Michael Tautschnig, Helmut Veith: **On the Structure and Complexity of Rational Sets of Regular Languages.** FSTTCS 2013: 377-388

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

Andreas Holzer, Visar Januzaj, Stefan Kugele, Boris Langer, Christian Schallhart, Michael Tautschnig, Helmut Veith: **Seamless Testing for Models and Code.** FASE 2011: 278-293

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: **How did you specify your test suite.** ASE 2010: 407-416

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

Andreas Holzer, Christian Schallhart, Michael Tautschnig, Helmut Veith: **FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement.** CAV 2008: 209-213

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Model Checking and Testing

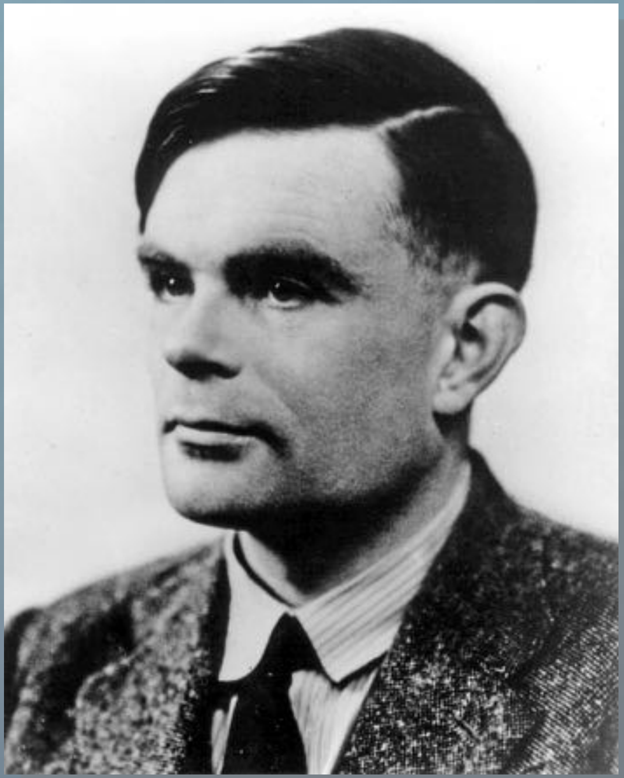
Theoretical Background
How we came to work on Testing

Undecidability (of Verification)

Incompleteness.

Non-elementary complexity.

NP-completeness.



Plains of Theoretical Computer Science

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.

Garmisch-Partenkirchen 1968
NATO Conference on Software Engineering

“ the first open admission of the software crisis.”

(Dijkstra, The Humble Programmer)

Dijkstra’s Turing Award Lecture 1972

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.

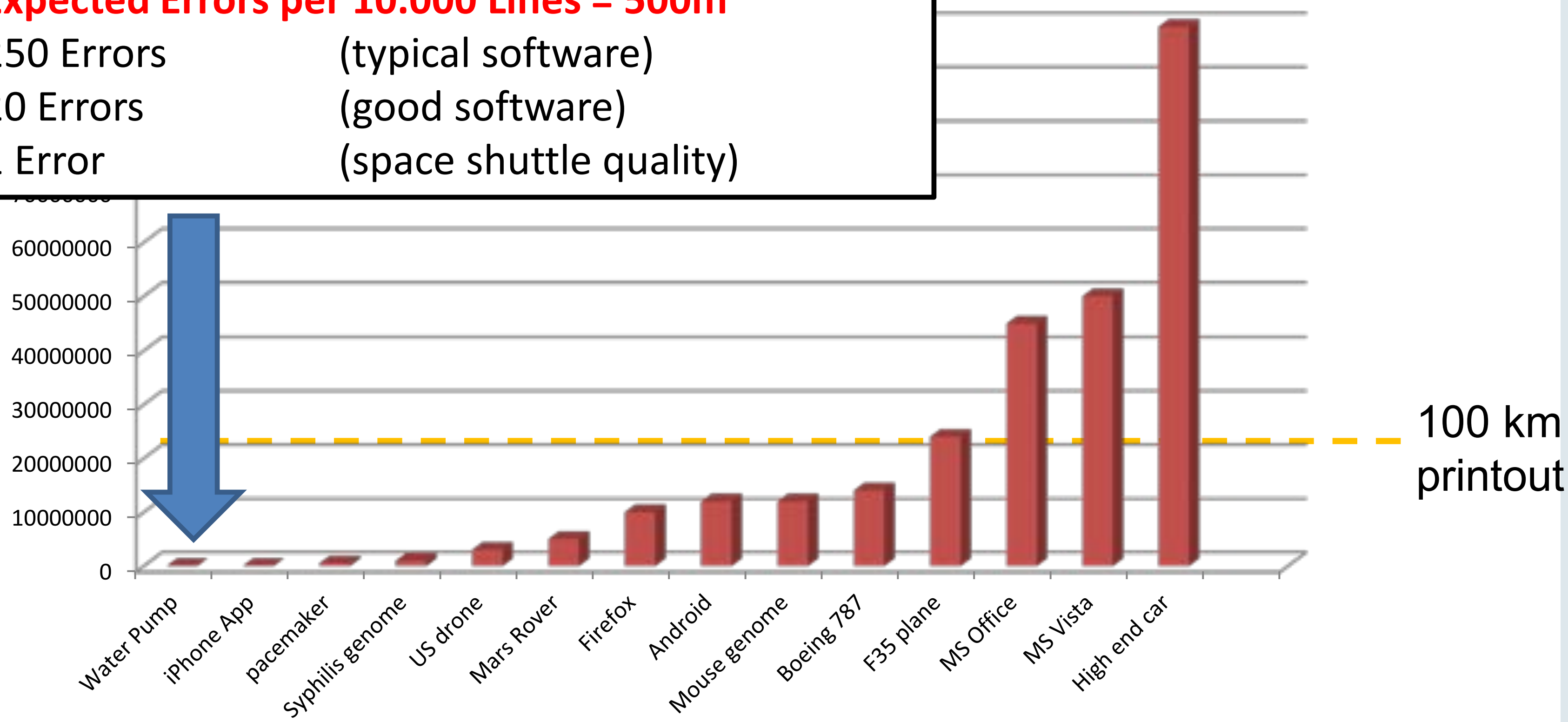
By definition this approach is only applicable when we restrict ourselves to intellectually manageable programs

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

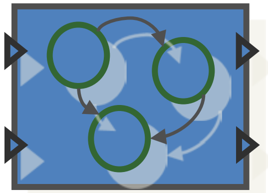
critical property

model checking

program

compilation

executable



code assertions
absence of deadlocks
termination
correct API use
path feasibility
memory violations
safety & liveness

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

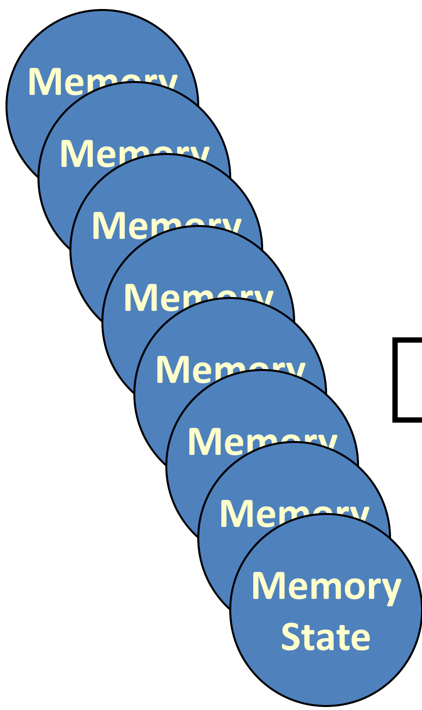
  Formal evidence for unreachability, spurious counterexamples due to abstract semantics
  e.g. SLAM (MSR), BLAST (Berkeley), CPA (Passau)

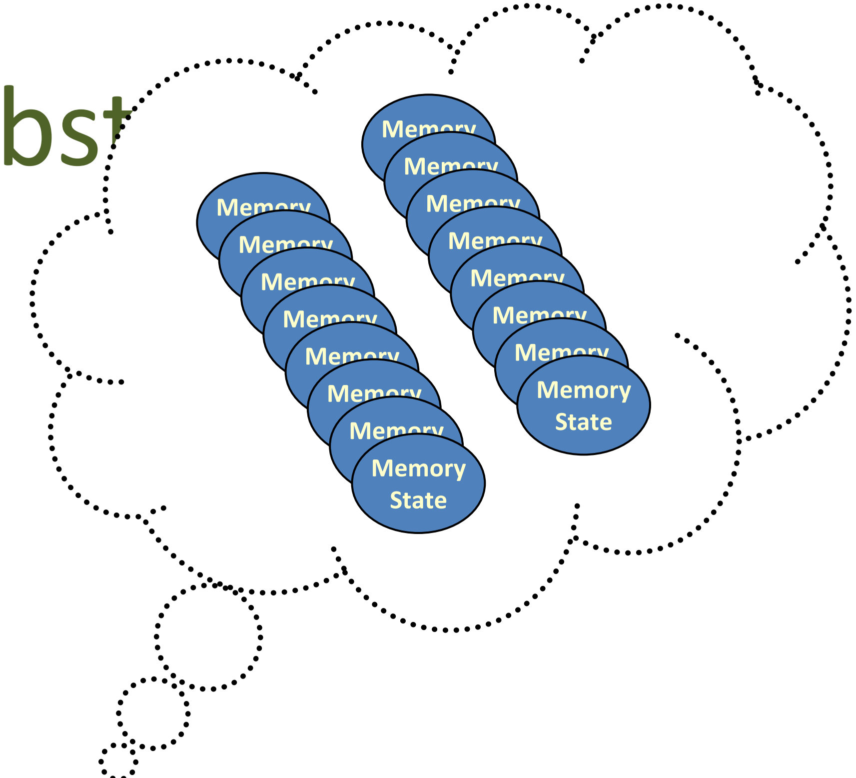
- **Bounded Model Checking**
  underapproximation of the state space

  Formal evidence for reachability, precise semantics, bounded size counterexamples e.g. CBMC (Kröning)

de facto combined using SAT / SMT solvers

Predicate Abst





$(x \geq y) \wedge (z + 5 < y)$

Abstract memory states are *formulas* describing properties of the memory content.

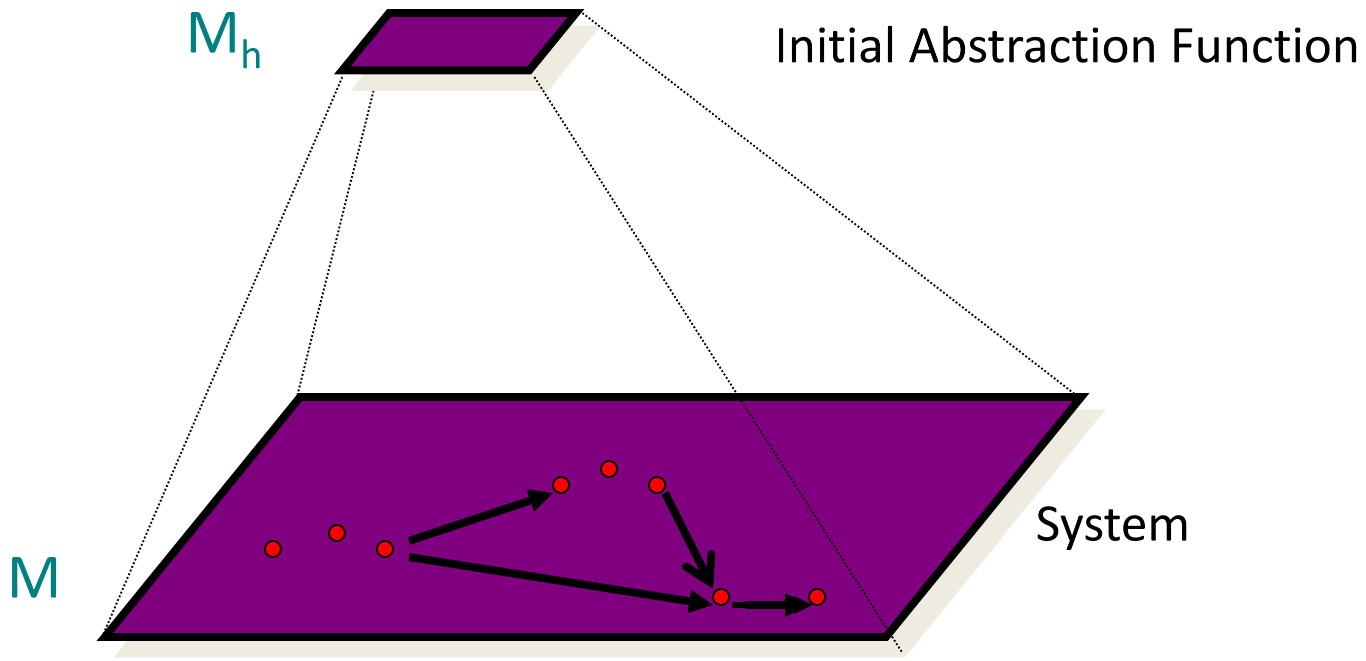
► Decision procedures, SAT solver, SMT, …

► **Need for classical logic!**

Graf, Saidi 1997

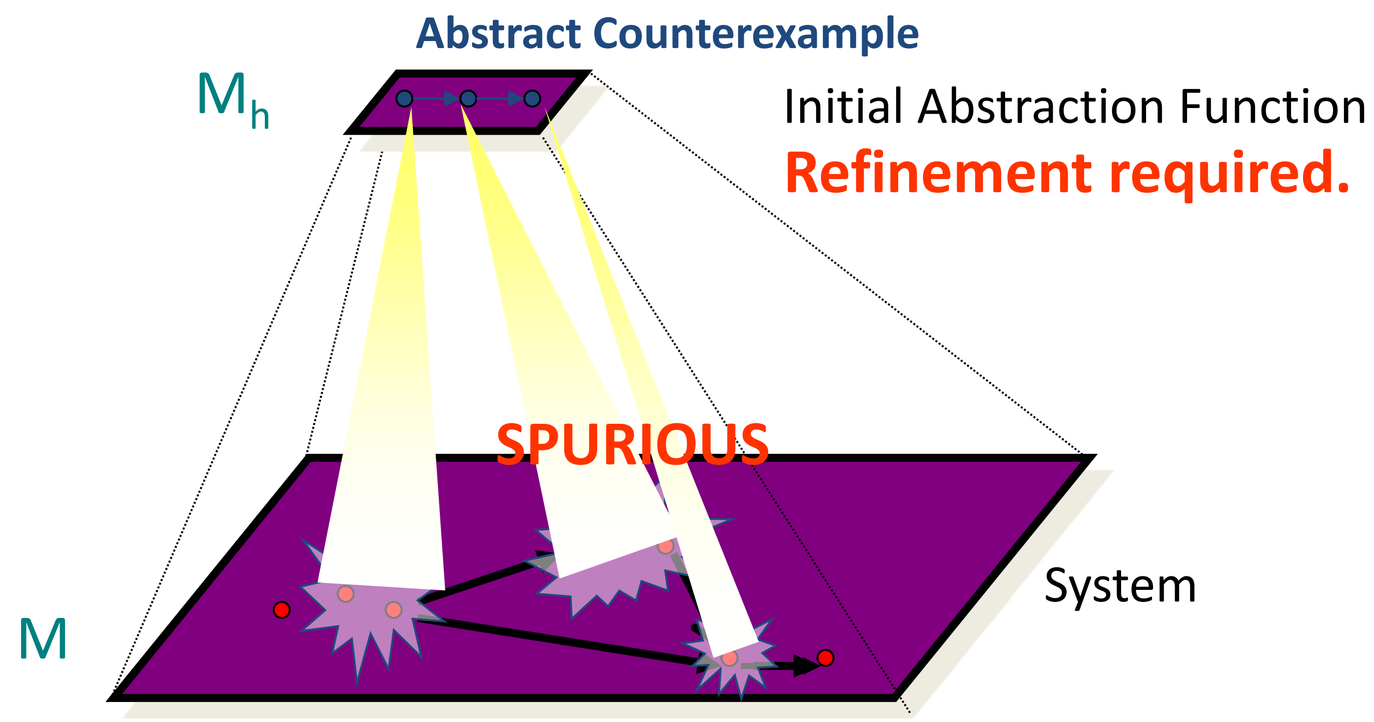
CEGAR (Counterexample-Guided Abstraction Refinement)

Adaptive Strategy



CEGAR (Counterexample-Guided Abstraction Refinement)

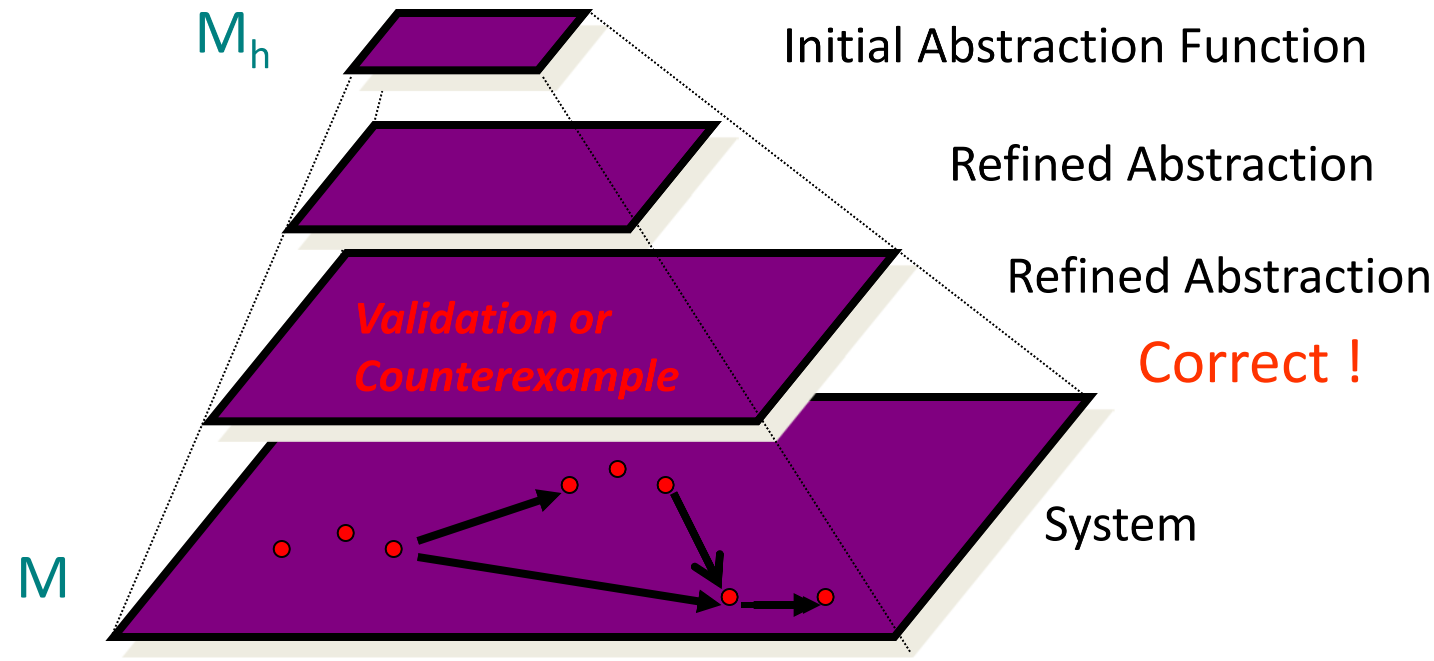
Adaptive Strategy



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
else B
```

⇒

```
assume(!(x>5));
B;
```

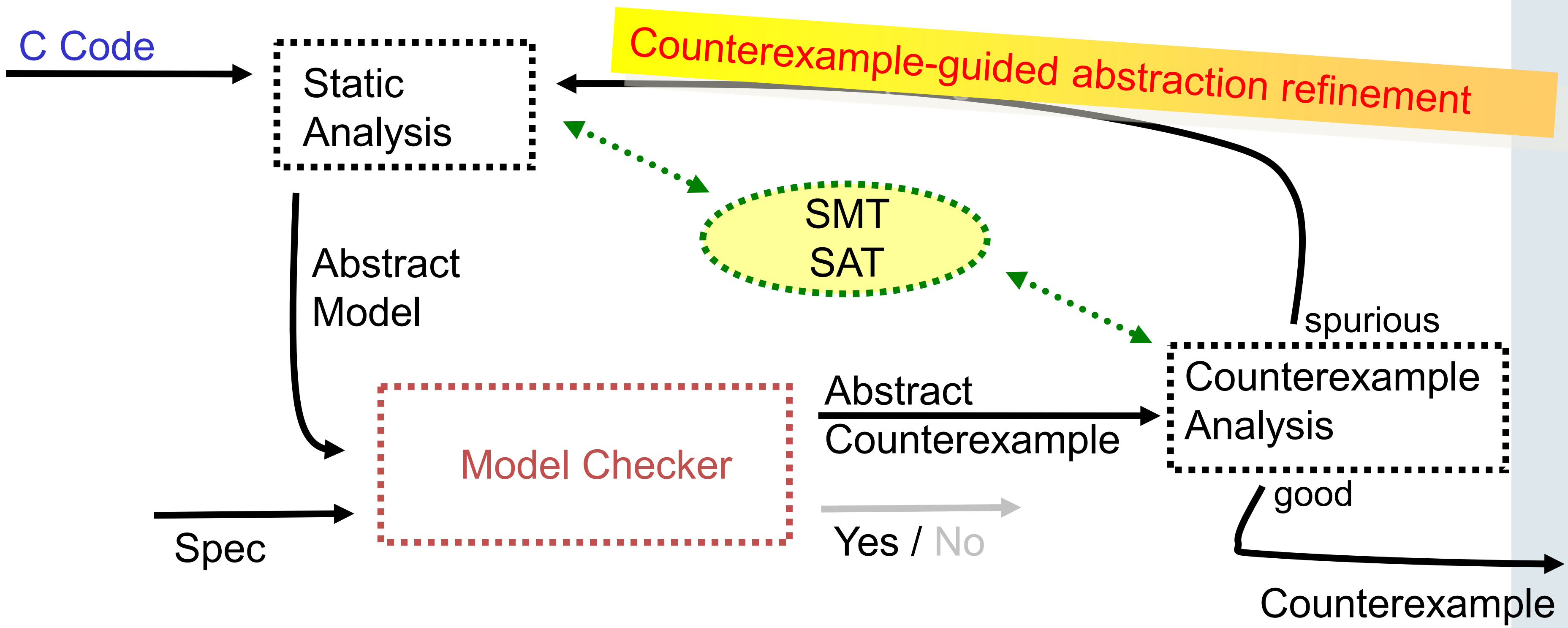
iv. Extract a formula representing the path
`x1=x0+5; assume(!(x1>5)); x2 = x1-5;`

⇒ (x1=x0+5) & (!(x1>5)) & (x2 = x1-5)

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

Iidealizing assumption: SMT is a reliable oracle.

Software Model Checking

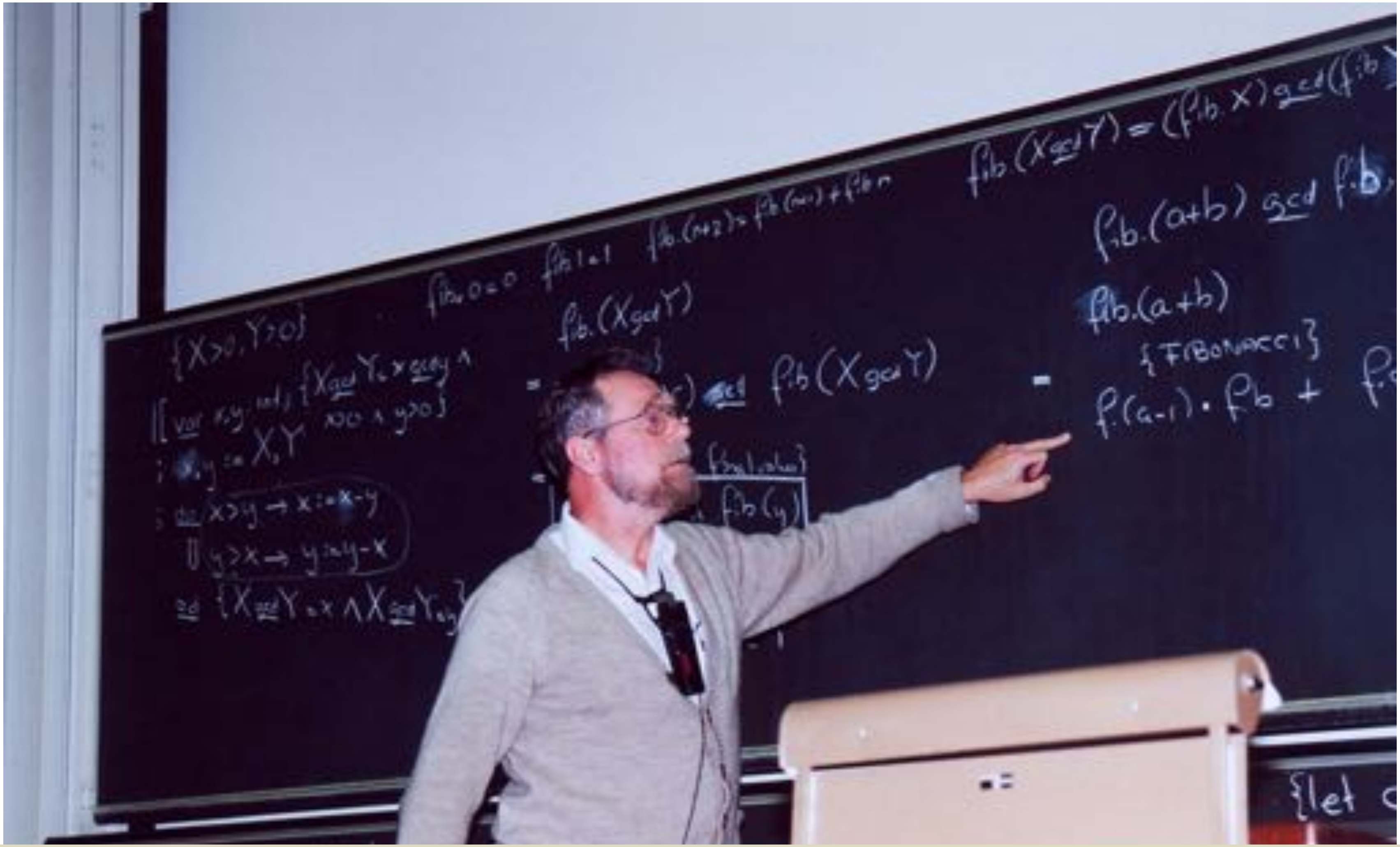


- 2000s: development of industrial strength C model checkers
- "   rivals theorem proving for many verification tasks" (Rushby)
- →Microsoft product for Windows device driver verification

Bill Gates 2002 on SW Model Checking

"device drivers we're building tools that do actual proofs about the software and how it works in order to guarantee the reliability."

Dijkstra’s Turing Award Lecture 1972



“Model checking is an acceptable crutch.”

Dijkstra’s Turing Award Lecture 1972

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”

Bill Gates 2002 on SW Model Checking

"device drivers we're building tools that do actual proofs about the software and how it works in order to guarantee the reliability."

Reduction of Model Checking to Testing

MC specification: no assertions are violated

AG(pc=l → assertion)

Program rewrite

`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*

high level model

synthesis / impl

Proper semantics ?

high level programming language

compilation

Compiler errors.

executable

execution

Execution time.
Power consumption.
Processor bugs.
Production errors.

hardware

Testing

“The purpose of abstraction is *not* to be vague, but to create a new semantic level in which one can be absolutely preci

We are not there yet.

FORTAS 2008-2011 (DFG/FWF)

(jointly with Real Time Systems Group, TU Vienna)

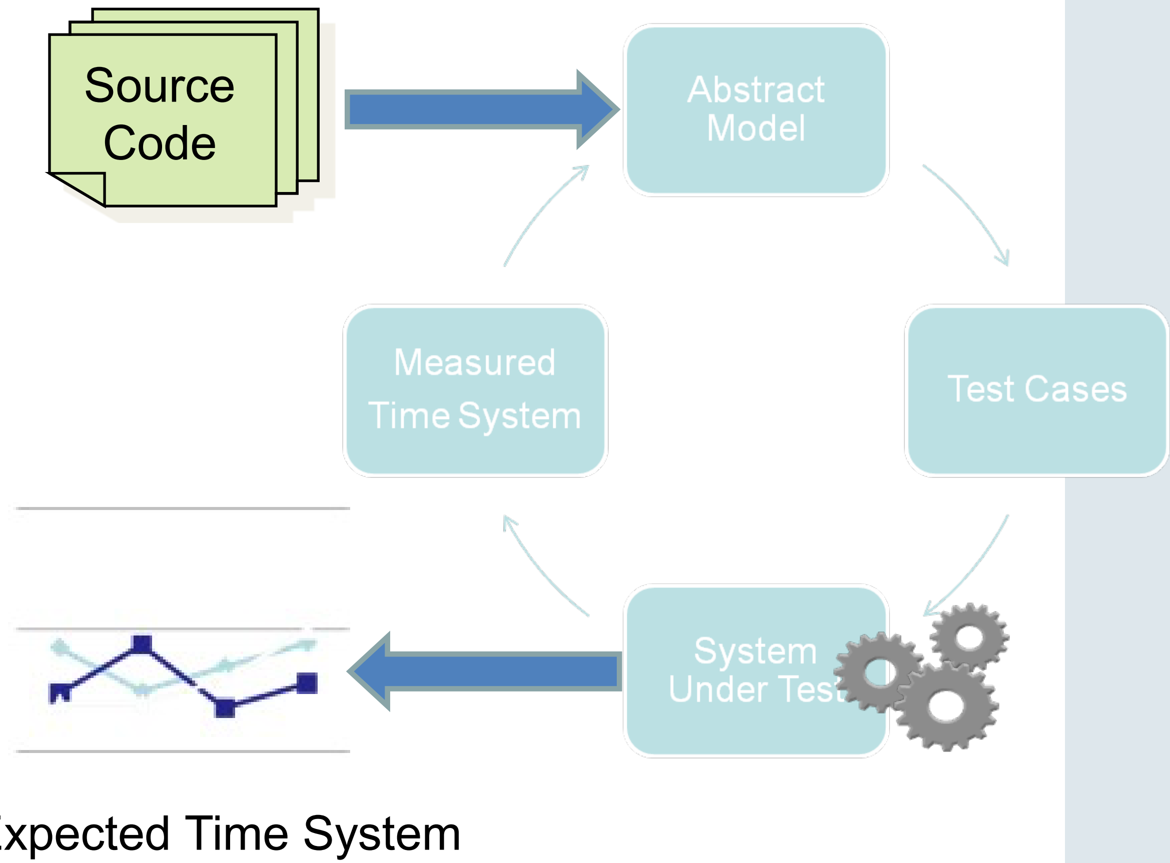
Execution time analysis in a white box setting

- C source code
- Focus on automatically generated code

Abstracts from platform

Execution times obtained through measurements

Requires large data sets, possibly with code coverage



Reduction of Testing to Model Checking

Test goal: Cover line 4242 of the program.

Model Checking Specification: **AG(pc != 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*

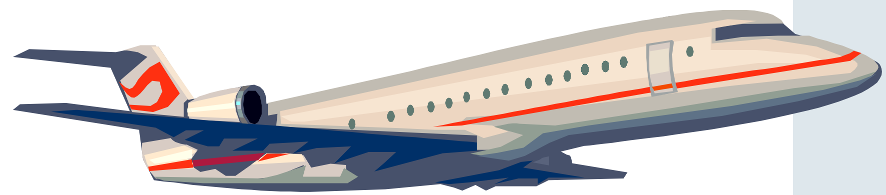
Precision of Coverage Criteria

„Condition Coverage“

cover all program c

SOFTWARE CONSIDERATIONS
IN AIRBORNE SYSTEMS
AND EQUIPMENT CERTIFICATION

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



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

Condition coverage ?

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

Commercial Tools

Coverage Meter, CTC++

BullseyeCoverage

83% coverage

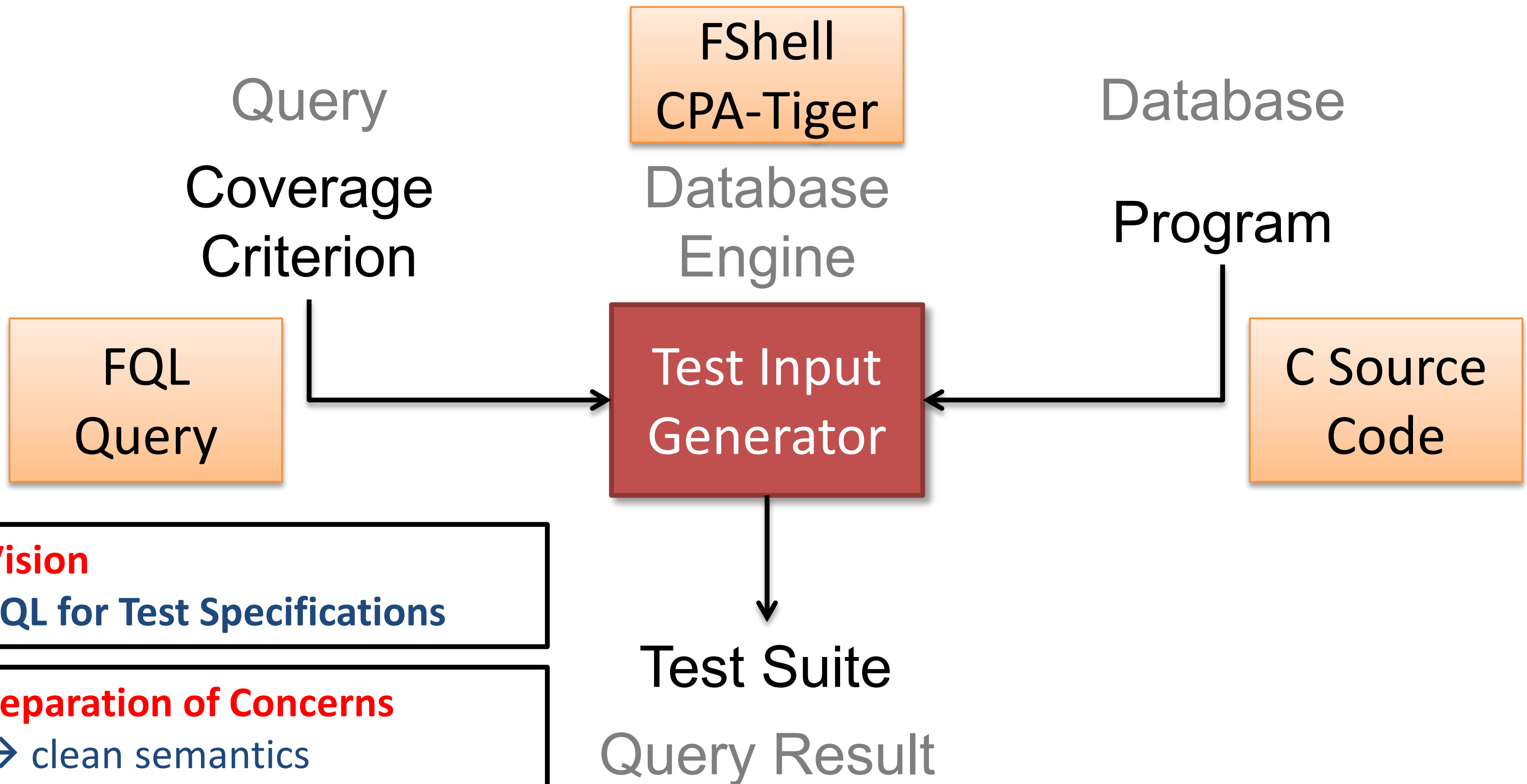
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

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



Vision
SQL for Test Specifications

Separation of Concerns
→ clean semantics
→ multiple engines
→ query dispatcher
→ stable query interface

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

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

SQL/Database Analogy

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

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., <u>multiple test cases</u>
2. Contains a <u>universal quantifier</u>
3. Assumes <u>knowledge about programs</u>.
   What IS a basic block for a logic ?
4. Has a meaning <u>independent of the program under test.</u>
   Can be translated into concrete specifications for a fixed program.

FQL

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 }
```

`Line 1.Line 2.Line 3.Line 4.Line 5.Line 8.Line 9.Line 10`

FQL

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 }
```

Line 1.Line 2.Line 3.Line 4.Line 5.Line 8.Line 9.Line 10

FQL

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  }
```

`Line 1.`**`Line 2`**`.Line 3.Line 4.Line 5.Line 8.Line 9.Line 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 }
```

`Line 1.Line 2.`**`Line 3`**`.Line 4.Line 5.Line 8.Line 9.Line 10`

FQL

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 }
```

Line 1.Line 2.Line 3.**Line 4**.Line 5.Line 8.Line 9.Line 10

FQL

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 }
```

`Line 1.Line 2.Line 3.Line 4.`**`Line 5`**`.Line 8.Line 9.Line 10`

FQL

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 }
```

`Line 1.Line 2.Line 3.Line 4.Line 5.`**`Line 8`**`.Line 9.Line 10`

FQL

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 }
```

`Line 1.Line 2.Line 3.Line 4.Line 5.Line 8.`**`Line 9`**`.Line 10`

FQL

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 }
```

`Line 1.Line 2.Line 3.Line 4.Line 5.Line 8.Line 9.`**`Line 10`**

FQL

Program Executions as Regular Expressions

```
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

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

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

  return tmp;
}
```

Program Executions as Regular Expressions

- Several Paths

```
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 1.Line 2.Line 3.Line 4.`**`(Line 5 + Line 7)`**`.Line 8.Line 9.Line 10`

Program Executions as Regular Expressions

- Several Paths
- *Language*: Set of program executions

```
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 1.Line 2.Line 3.Line 4.`**`(Line 5 + Line 7)`**`.Line 8.Line 9.Line 10`

Program Executions as Regular Expressions

- Several Paths
- *Language*: Set of program executions
- *Test goals* are also sets of program executions

```
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 1.Line 2.Line 3.Line 4.`**`(Line 5 + Line 7)`**`.Line 8.Line 9.Line 10`

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

FQL

Program Executions as Regular Expressions

- 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?

```
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 }
```

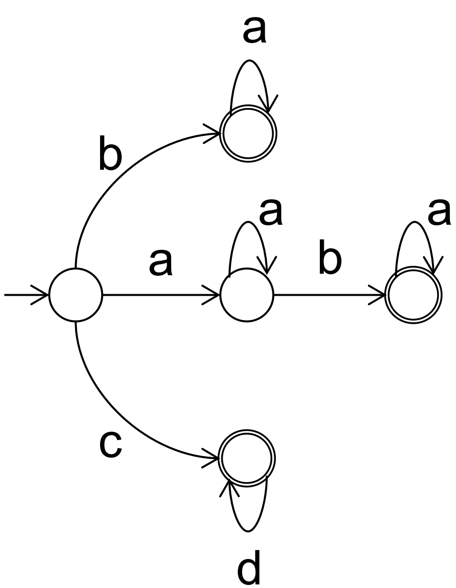
`Line 1.Line 2.Line 3.Line 4.`**`(Line 5 + Line 7)`**`.Line 8.Line 9.Line 10`

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

FShell Query Language (FQL)

Quoted Regular Expressions

A_1

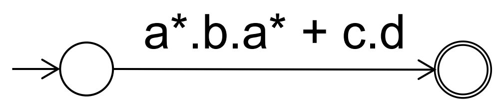


a*.b.a* + c.d*

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

Infinite number of specific paths

A_2

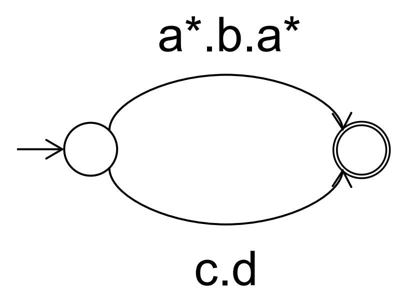


“a*.b.a* + c.d*“

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

One test goal

A_3



“a*.b.a*“ + “c.d*“

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

Two test goals

FQL

Filter Functions

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

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

  return tmp;
}
```

FQL

Filter Functions: knowledge about programs

- `@ID`

```
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 1 + Line 2 + Line 3 + Line 4 + Line 5 + Line 6 + Line 7 + Line 8 + Line 9 + Line 10`

FQL

Filter Functions: knowledge about programs

- @ID
- @BASICBLOCKENTRY

```
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 2 + Line 5 + Line 7 + Line 9

FQL

Filter Functions: knowledge about programs

- @ID
- @BASICBLOCKENTRY
- @ENTRY

```
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 1

FQL

Filter Functions: knowledge about programs

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

```
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 10`

FQL

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

FQL

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 }
```

FQL

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 }
```

- Filter functions can be combined:

FQL

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 }
```

- Filter functions can be combined:
  - `@BASICBLOCKENTRY(@FUNCTION(f))`
  - `@BASICBLOCKENTRY(@FUNCTION(f)|@FUNCTION(g))`
  - `...`

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“

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“

C Source Code

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


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“

FQL Query

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

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

...

C Source Code

```
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;
```

Test Suite

a) x = 10

b) x = 11

...

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"

FQL Query

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

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

...

a)

VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Coverage Criteria as FQL Queries

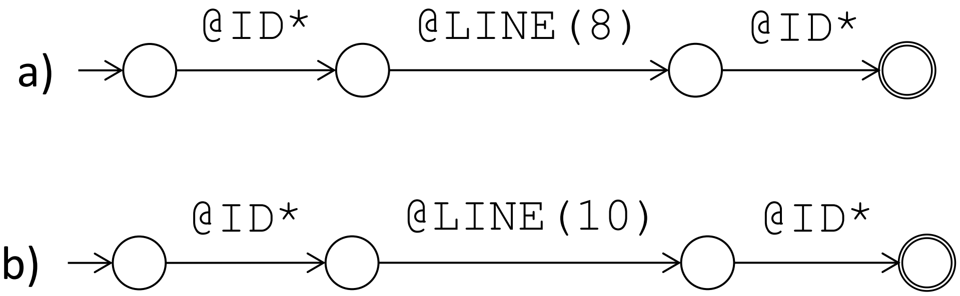
“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

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

b) **`cover`** `"@ID*".@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"

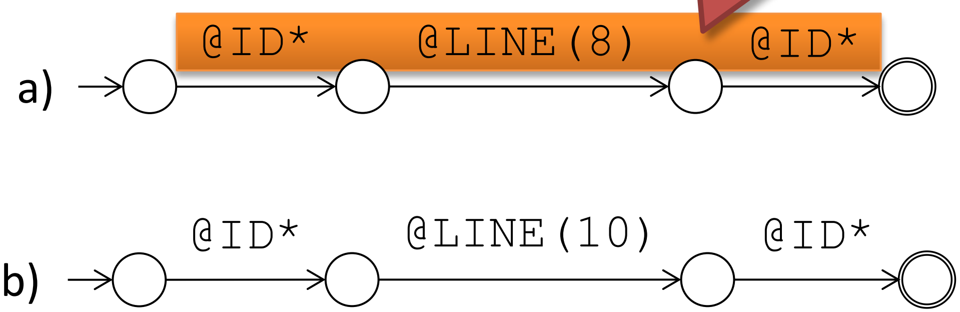
FQL Query

a) `cover "@ID*".@LINE`

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

...

word = regular expression



...

VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Coverage Criteria as FQL Queries

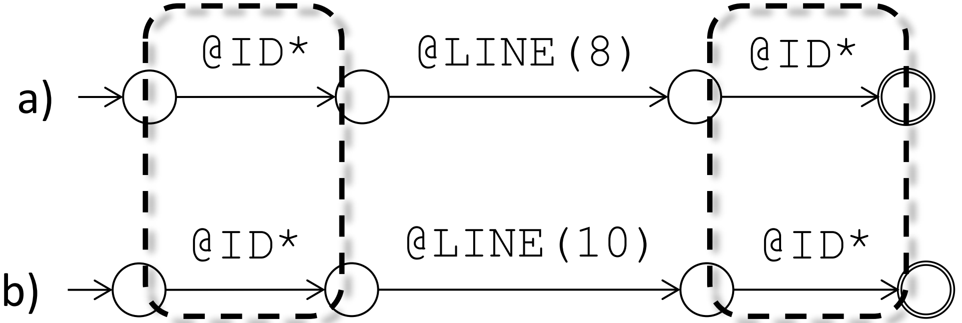
"for each basic block in the program there is a test case in the test suite which covers the basic block"

FQL Query

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

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

...



VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Coverage Criteria as FQL Queries

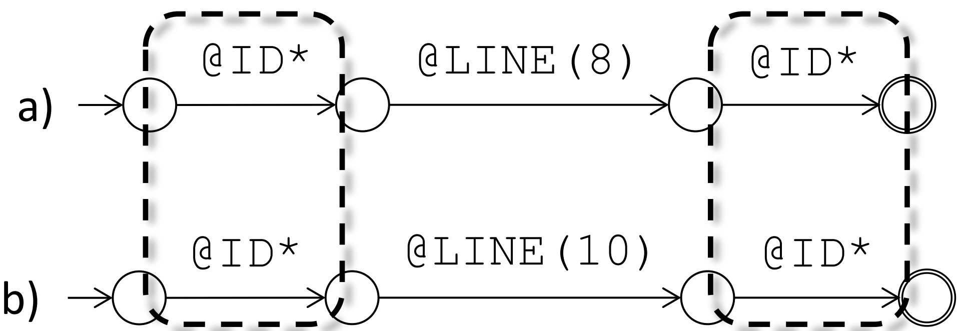
“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

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

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

…



VMCAI’09, ASE’10, HVC’10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Coverage Criteria as FQL Queries

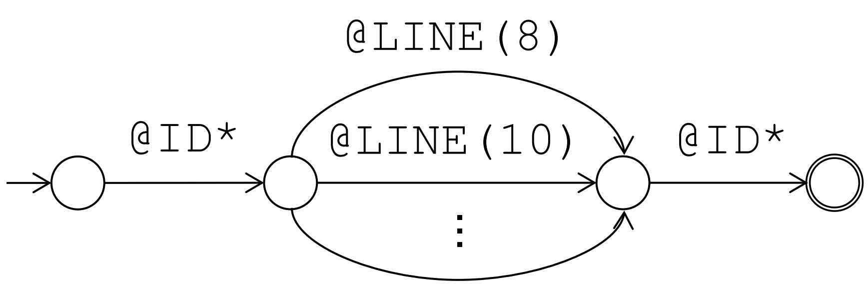
“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

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

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

...



VMCAI’09, ASE’10, HVC’10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Coverage Criteria as FQL Queries

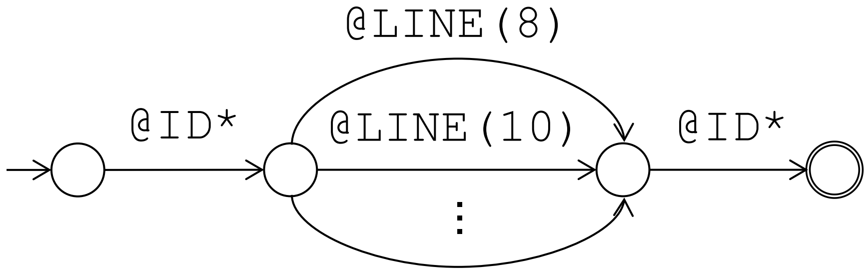
“for each basic block in the program there is a test case in the test suite which covers the basic block“

FQL Query

a) **cover** “@ID*“.@LINE(8).“@ID*“

b) **cover** “@ID*“.@LINE(10).“@ID*“

...



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"

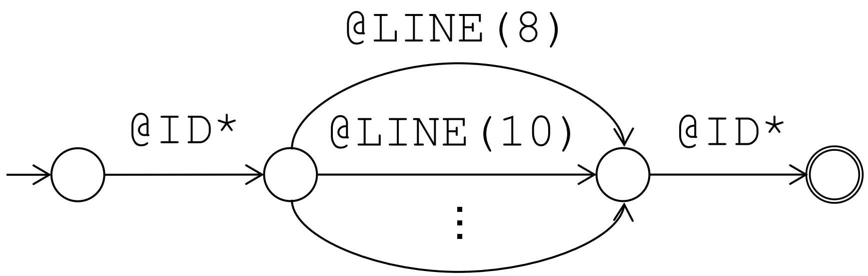
FQL Q

a) **cover** "@

b) **cover** "@

...

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

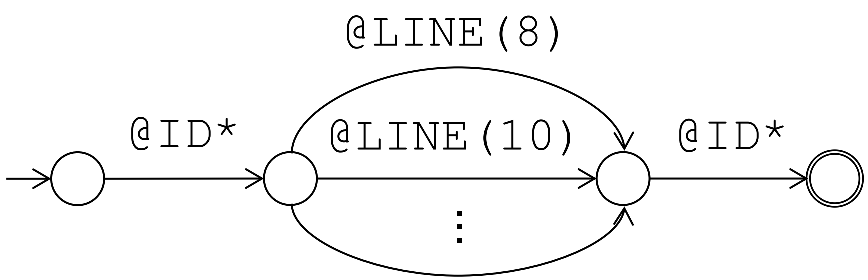
"for each basic block in the program there is a test case in the test suite which covers the basic block"

FQL Query

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

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

...



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

VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

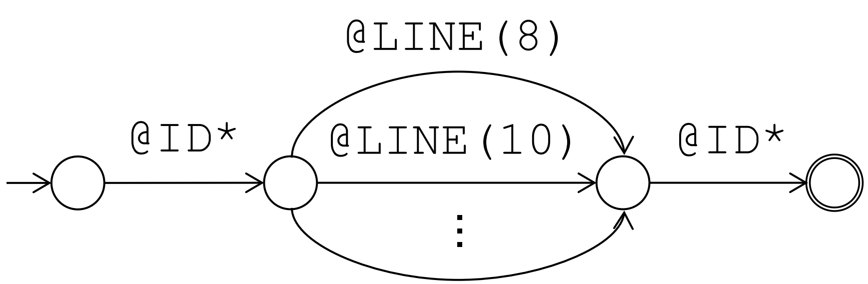
Coverage Criteria as FQL Queries

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*"

VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

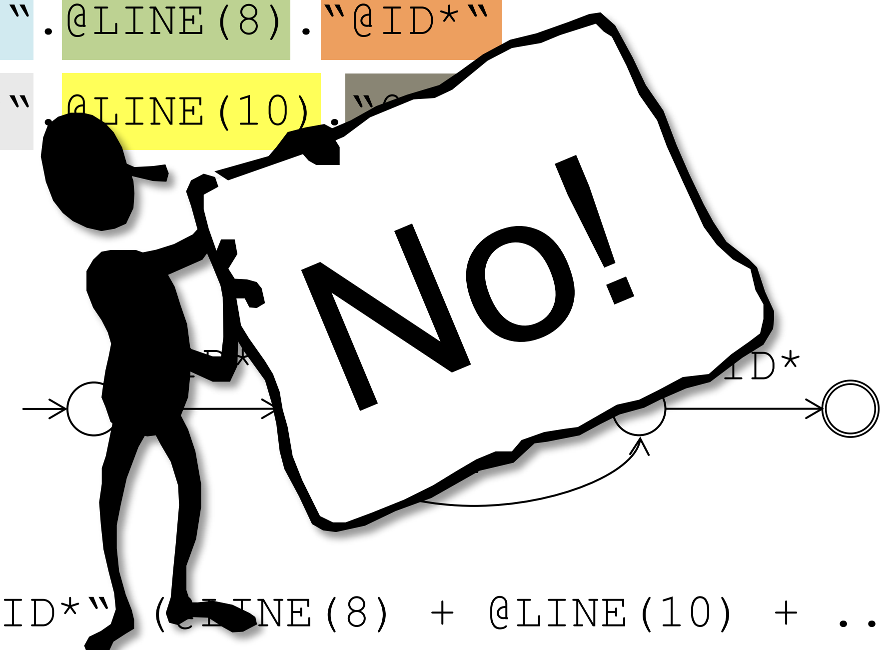
Coverage Criteria as FQL Queries

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

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

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

...



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

VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Filter Functions

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

FShell Query Language (FQL)

Filter Functions Revisited

- `@ID`

```
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 1 + Line 2 + Line 3 + Line 4 + Line 5 + Line 6 + Line 7 + Line 8 + Line 9 + Line 10`

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))`

```
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 1 + Line 2 + Line 3 + Line 4 + Line 5 + Line 6 + Line 7 + Line 9 + Line 10`

VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Filter Functions Revisited

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

```
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`
- ...

```
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)

Coverage Criteria as FQL Queries

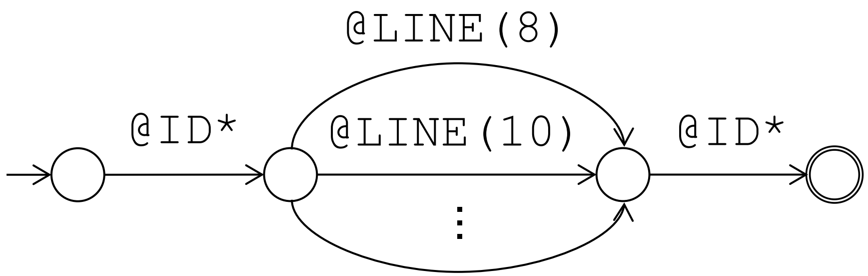
"for each basic block in the program there is a test case in the test suite which covers the basic block"

FQL Query

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

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

...



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

VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Coverage Criteria as FQL Queries

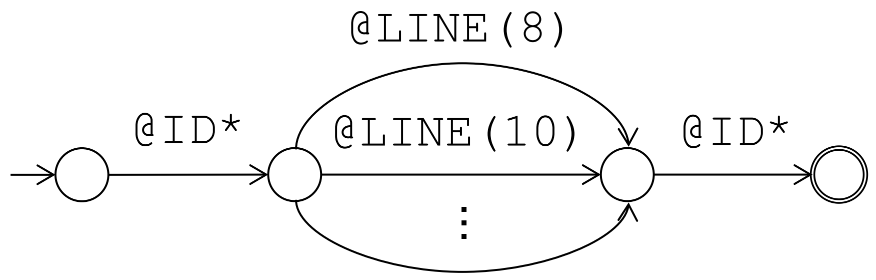
"for each basic block in the program there is a test case in the test suite which covers the basic block"

FQL Query

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

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

...



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

VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

FShell Query Language (FQL)

Passing Clauses in Coverage Criteria

```
cover "@ID*".@BASICBLOCKENTRY."@ID*"
passing "@ID*.NOT(@FUNCTION(unimplemented)).@ID*"
```


VMCAI'09, ASE'10, HVC'10: *Holzer , Schallhart, Tautschnig, Veith*

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,?:)))

Complex Coverage Criteria

"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

Complex Coverage Criteria

"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

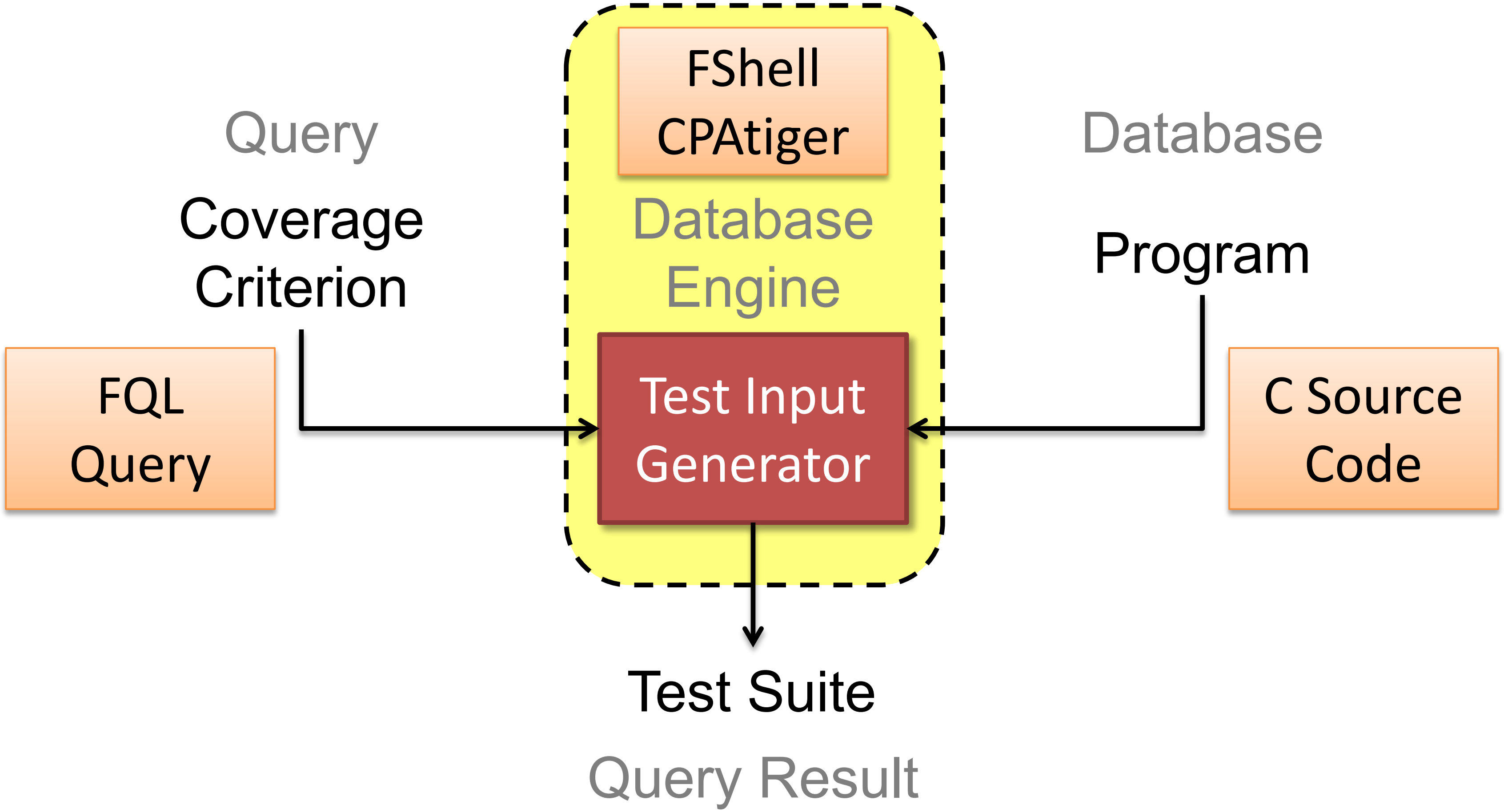
I. Test Specification Language FQL

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

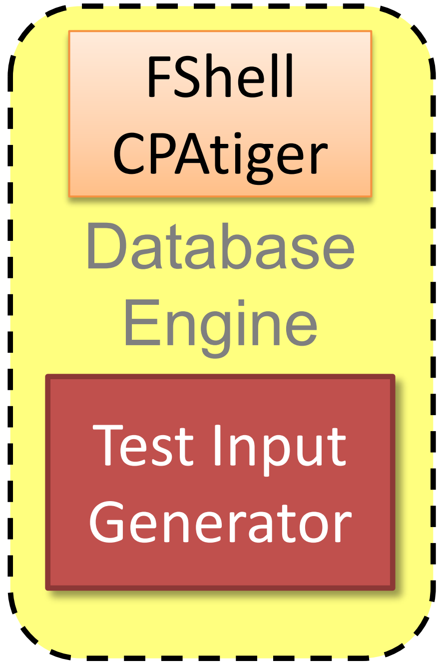
Programs as Databases



Query-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

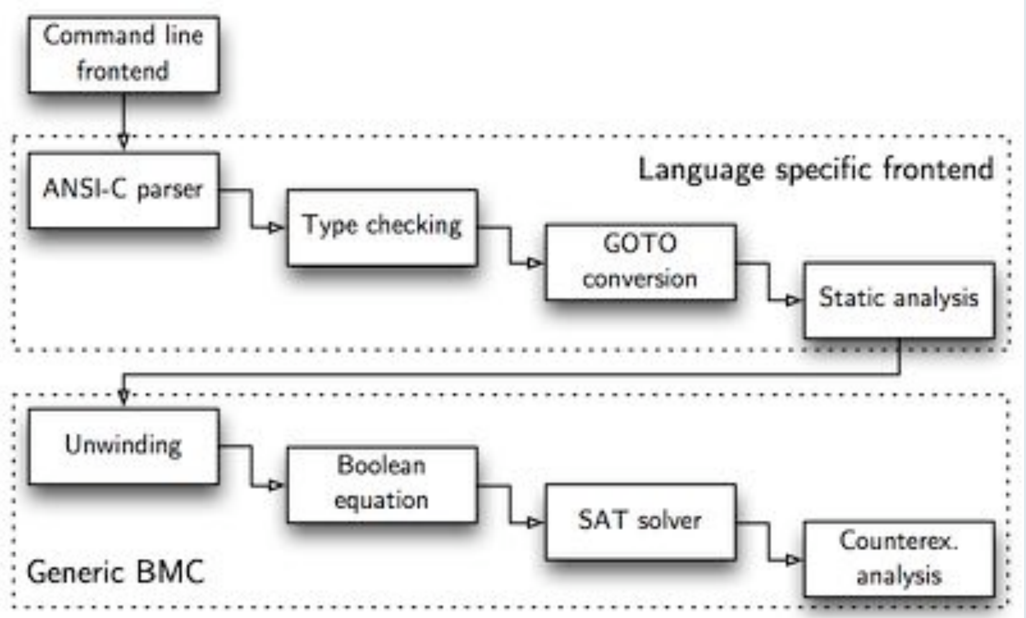
Background: Kröning's CBMC

C Bounded Model Checker

Clean and stable code base

Full ANSI-C support

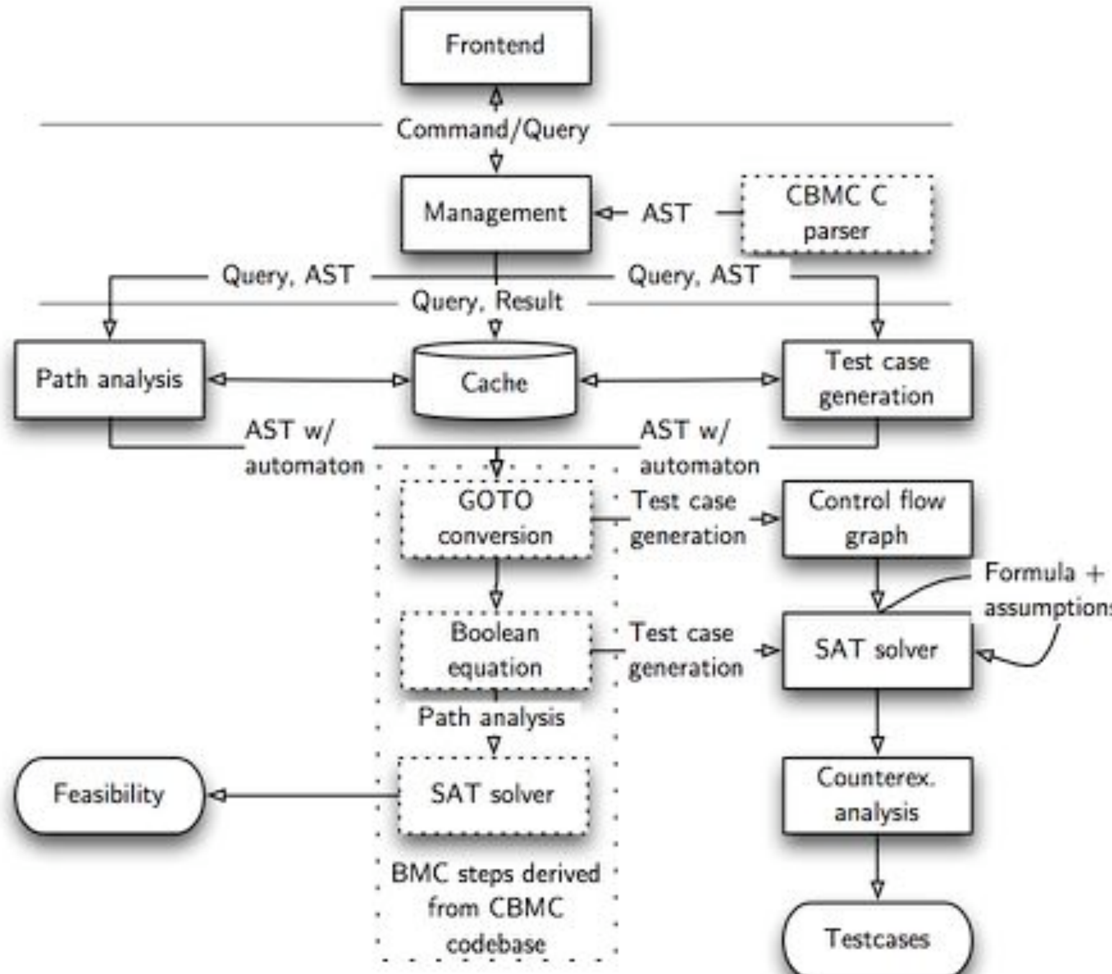
SAT solver =
decision procedure



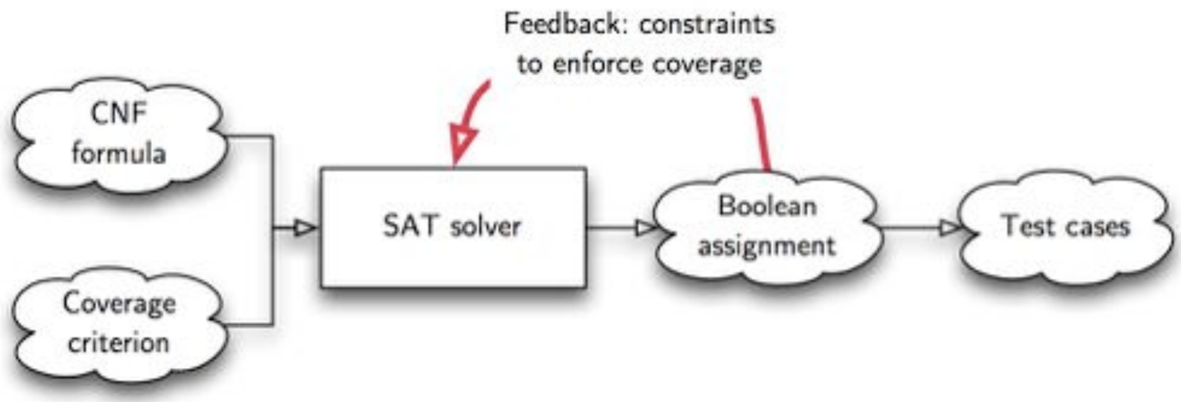
FShell: Architecture

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 $\Phi[\Pi_A^{\mathcal{T}}]$

Example test goals $\{\Psi_1, \Psi_2, \Psi_3, \Psi_4\}$

Initial constraint: "reach some test goal"

$$\mathsf{ICSPC}_0 = \Phi[\Pi_A^{\mathcal{T}}] \wedge \bigvee_{a=1\ldots4} (S_a \wedge \Phi[\Psi_a])$$

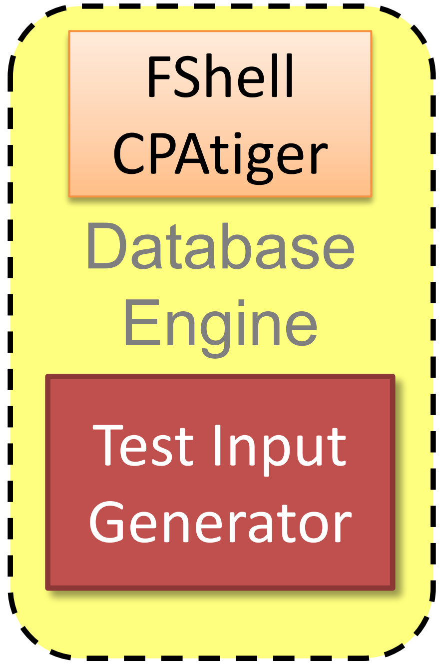
Assume Ψ_1 and Ψ_3 are satisfied by first solution:
"reach new test goal"

$$\mathsf{ICSPC}_1 = \mathsf{ICSPC}_0 \wedge \neg S_1 \wedge \neg S_3$$

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

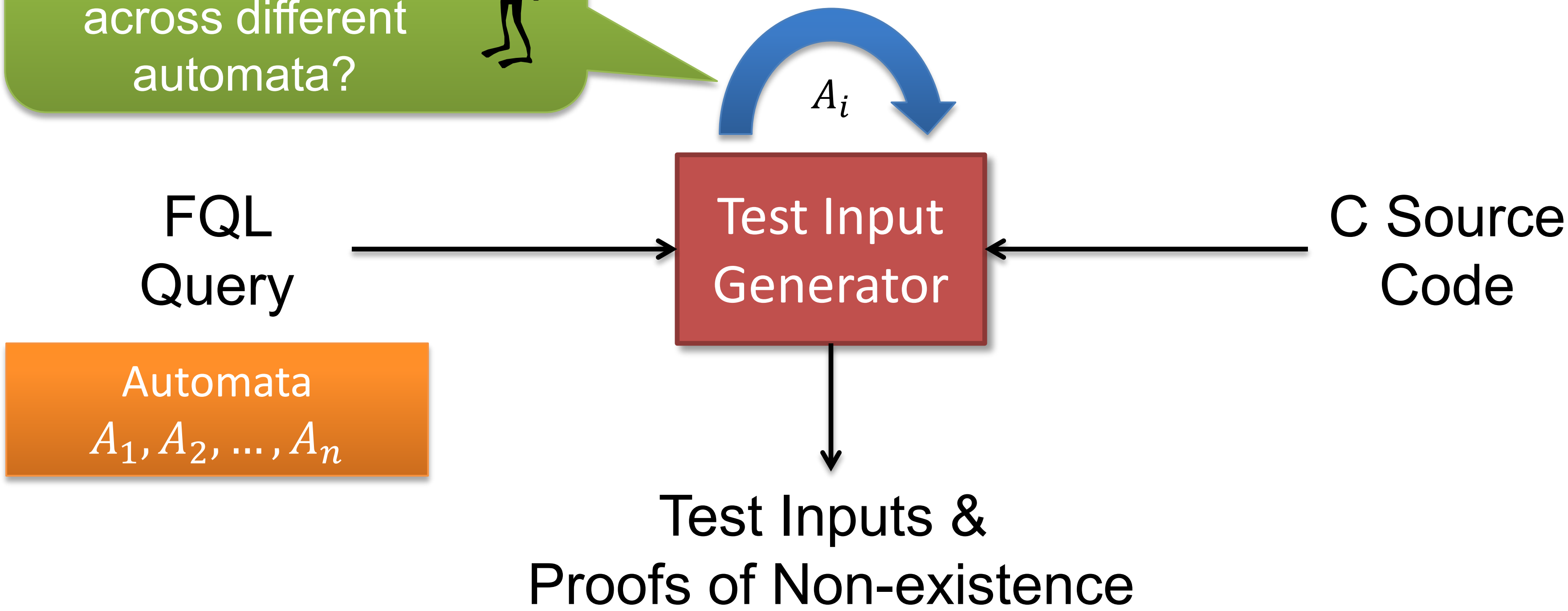
ogram

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

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

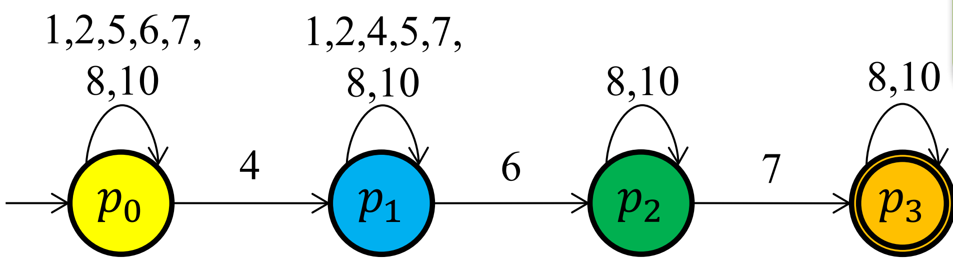
How can we reuse analysis results across different automata?

Repeated Invocation of an Automaton-guided Reachability Analysis

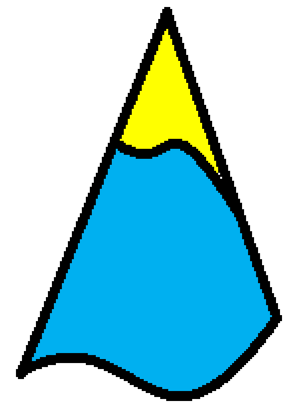


[ESOP'13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse

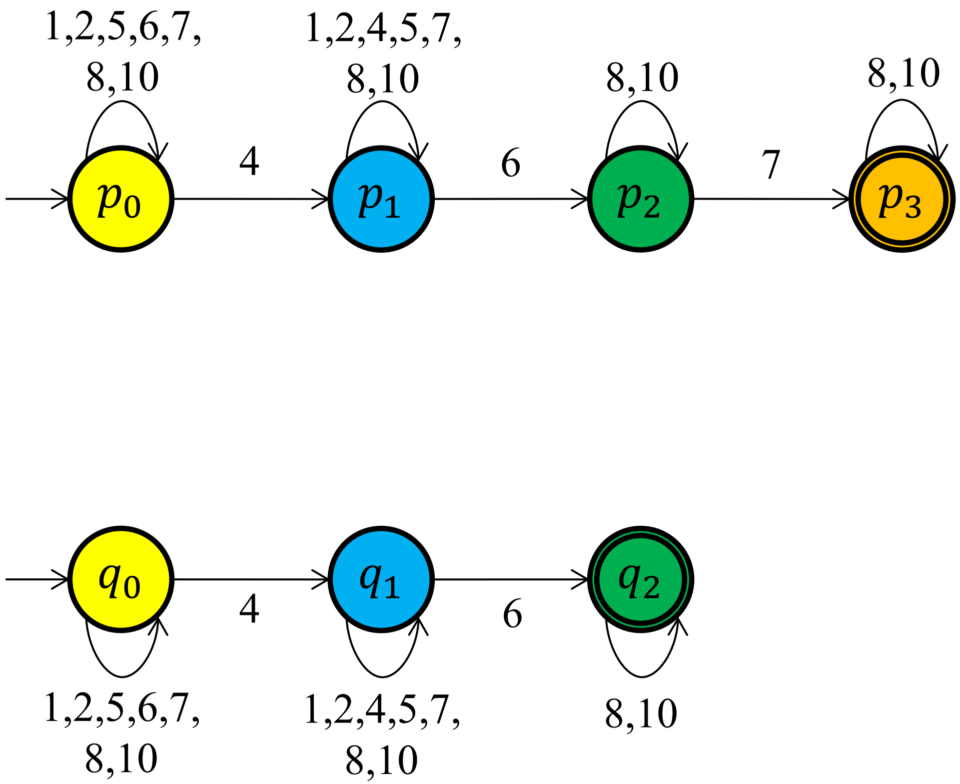


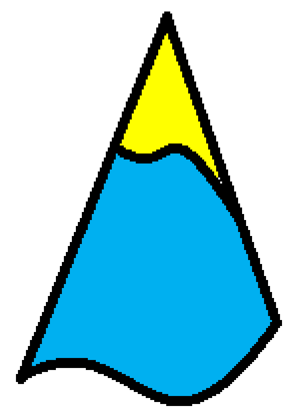
For this automaton
we ran a
reachability analysis



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

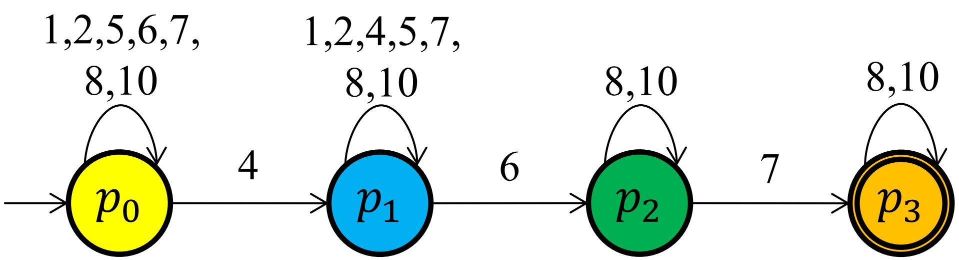
Information Reuse

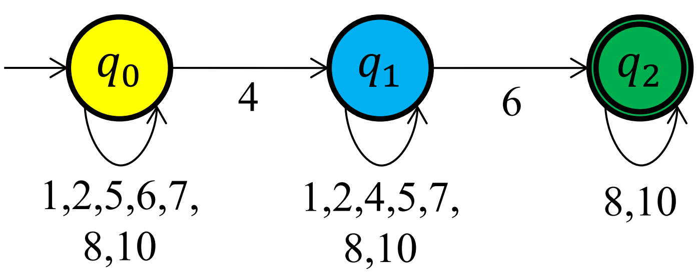


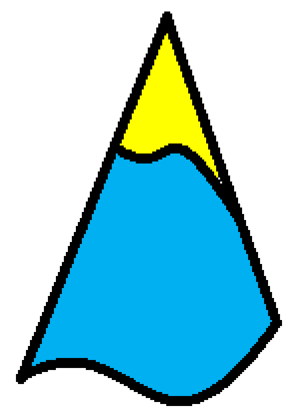


[ESOP'13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse



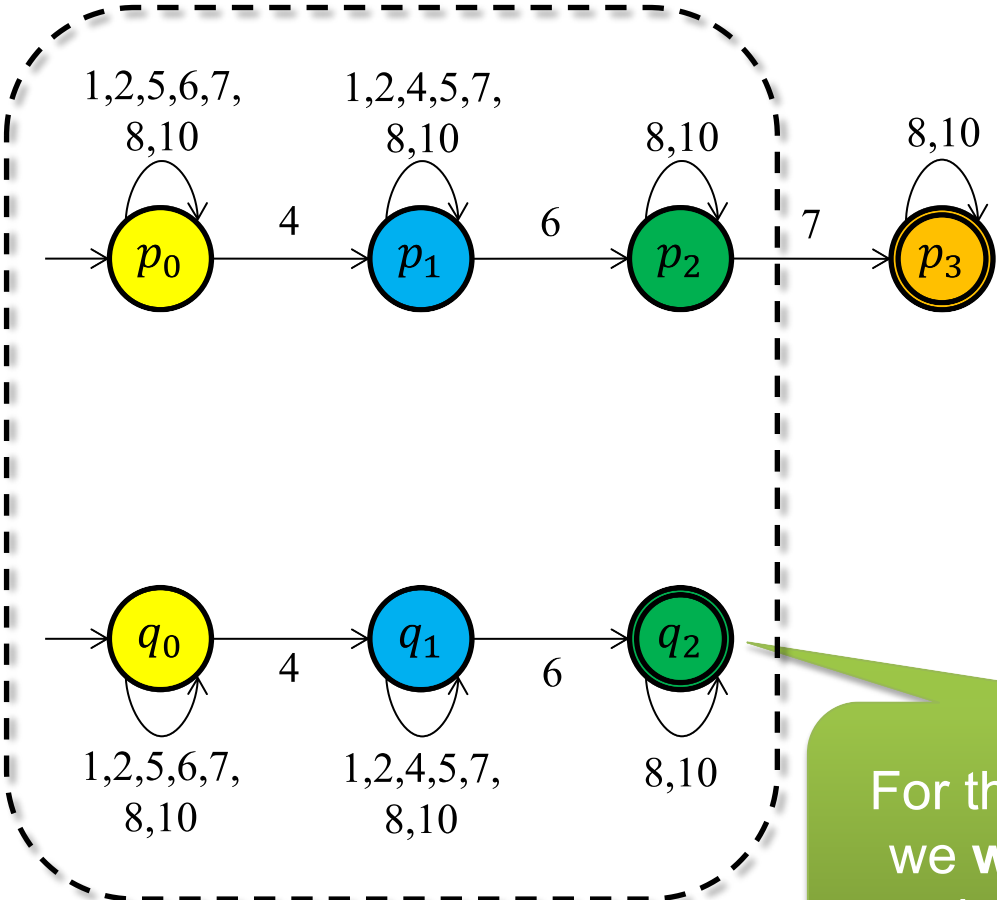


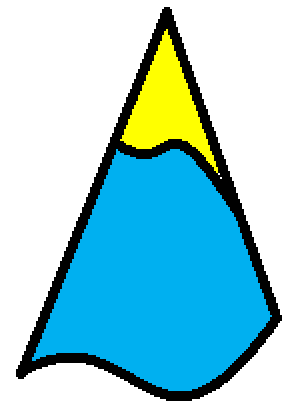


For this automaton we **want to** run a reachability analysis

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

Information Reuse

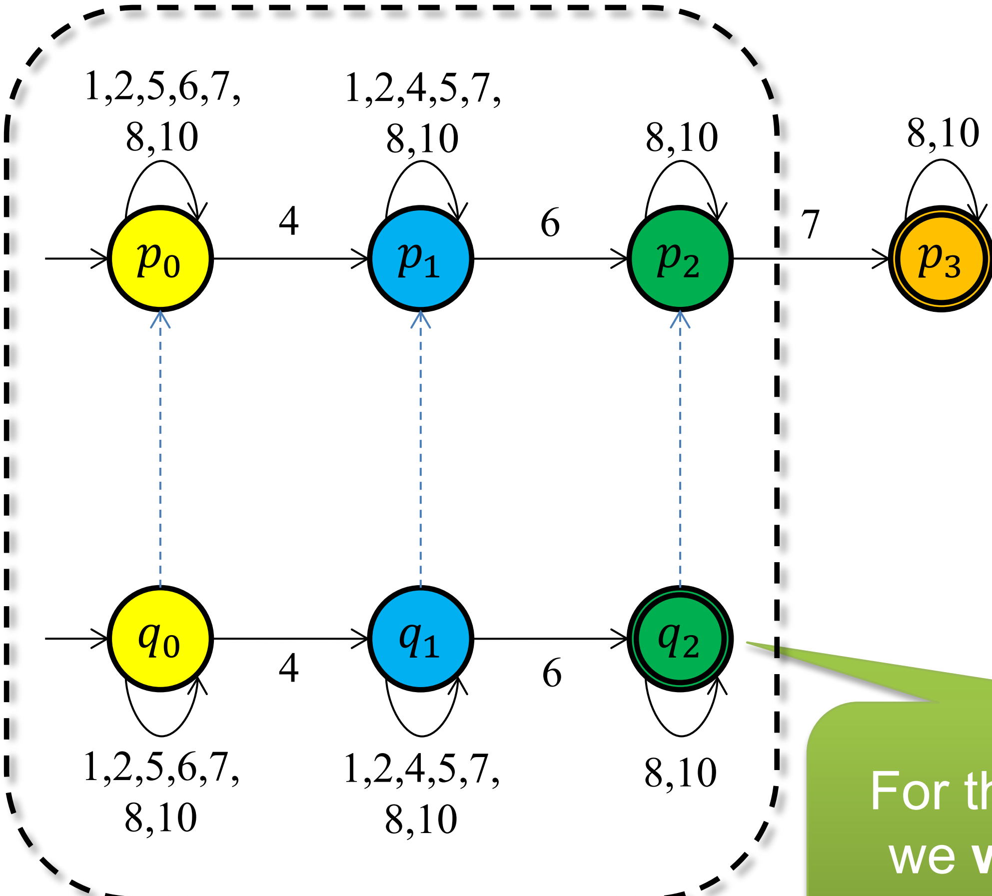


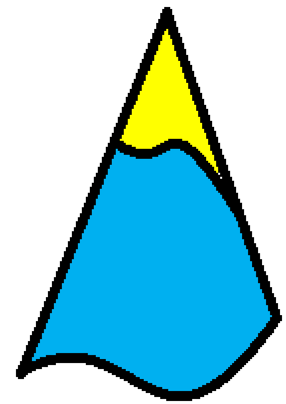


For this automaton we **want to** run a reachability analysis

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

Information Reuse

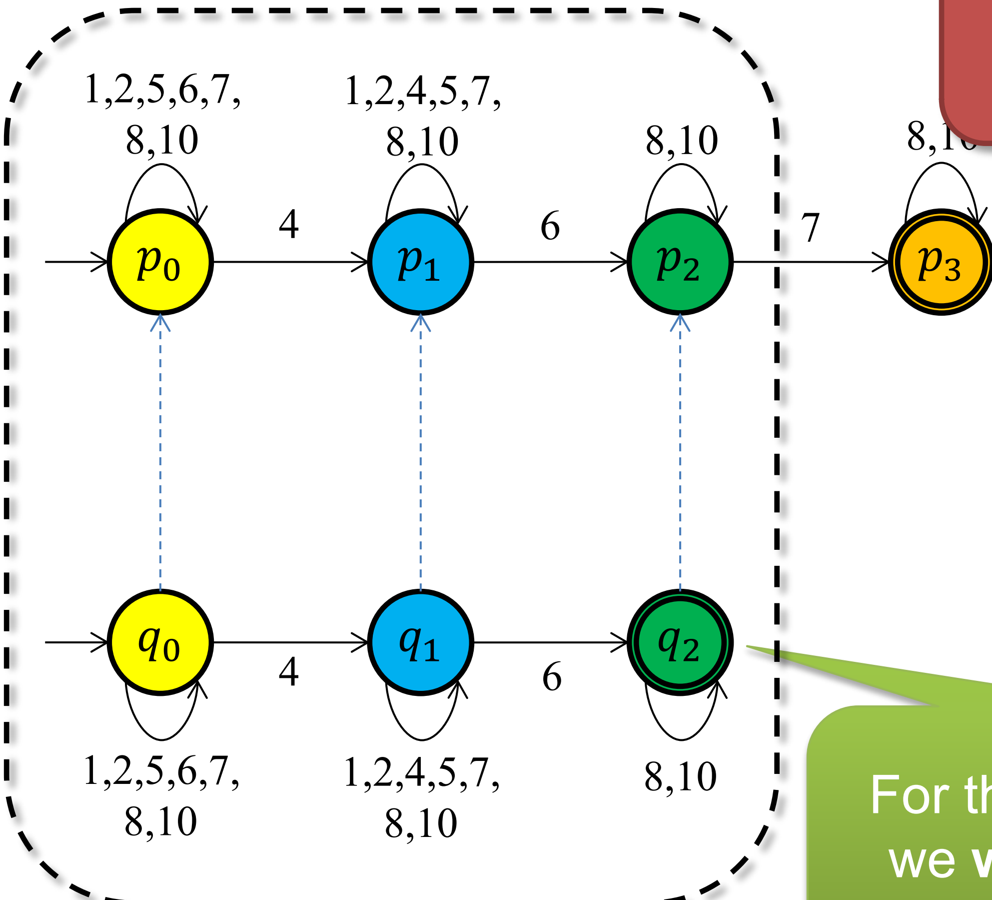




For this automaton we **want to** run a reachability analysis

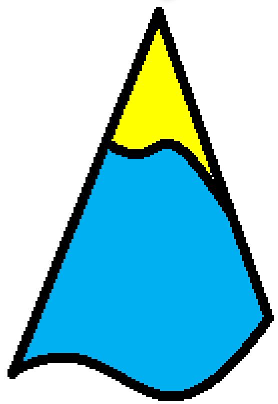
[ESOP'13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse



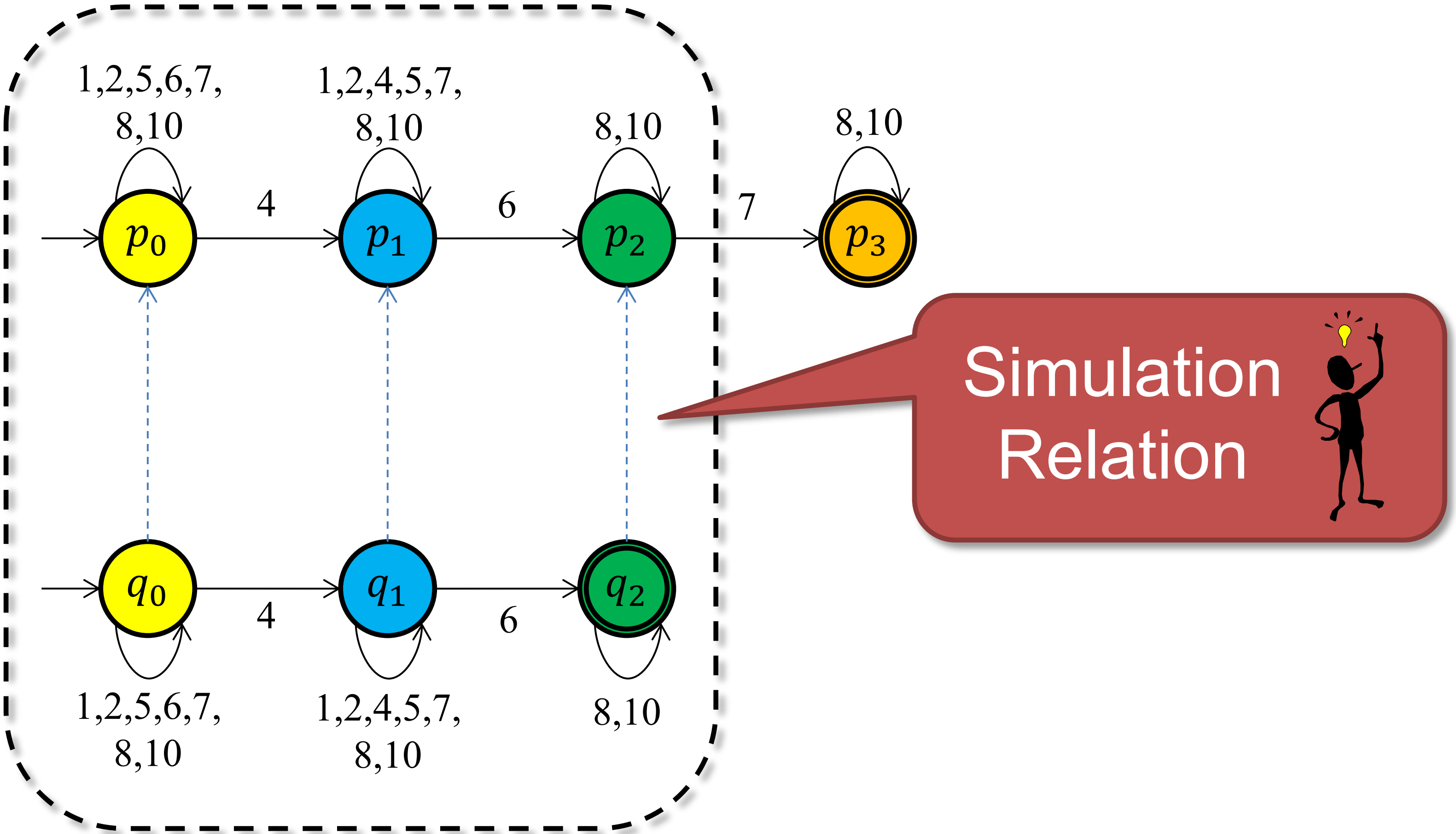
Reuse all reachability infos!

For this automaton we **want to** run a reachability analysis



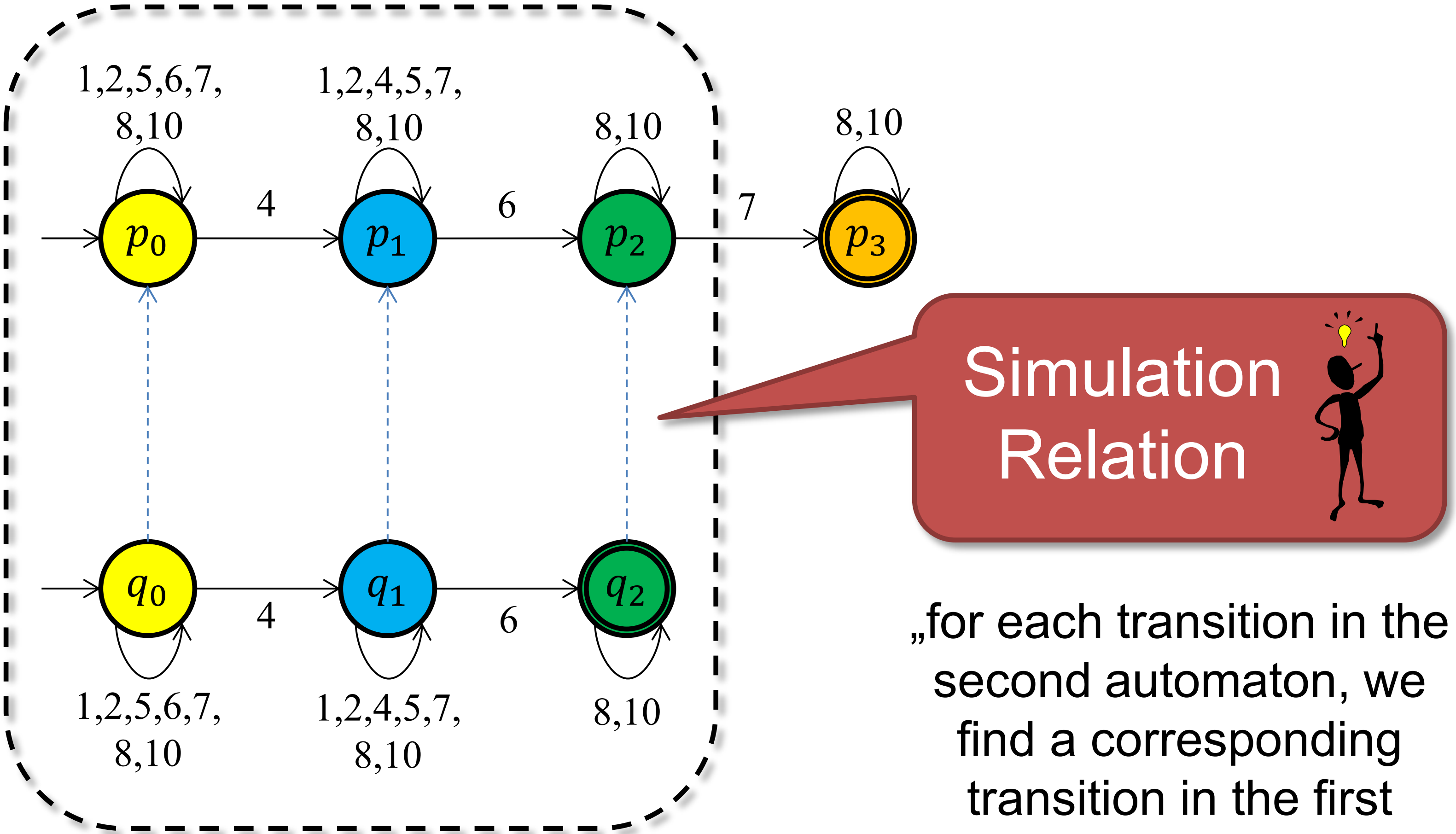
[ESOP'13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse



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

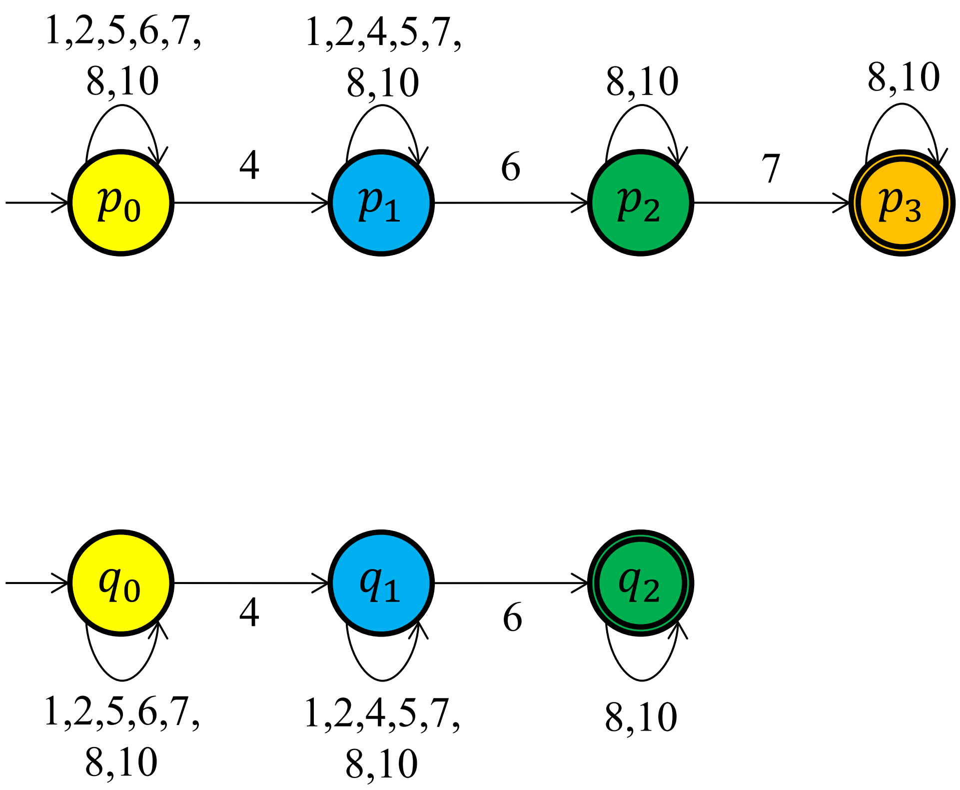
Information Reuse



„for each transition in the second automaton, we find a corresponding transition in the first automaton"

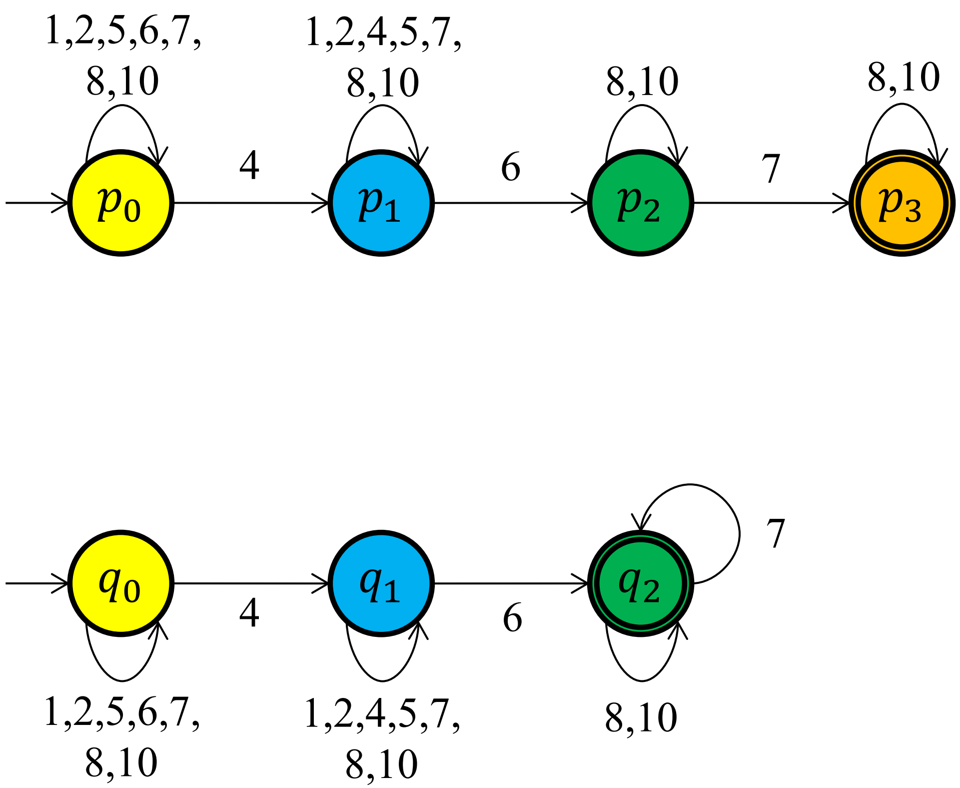
[ESOP'13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse



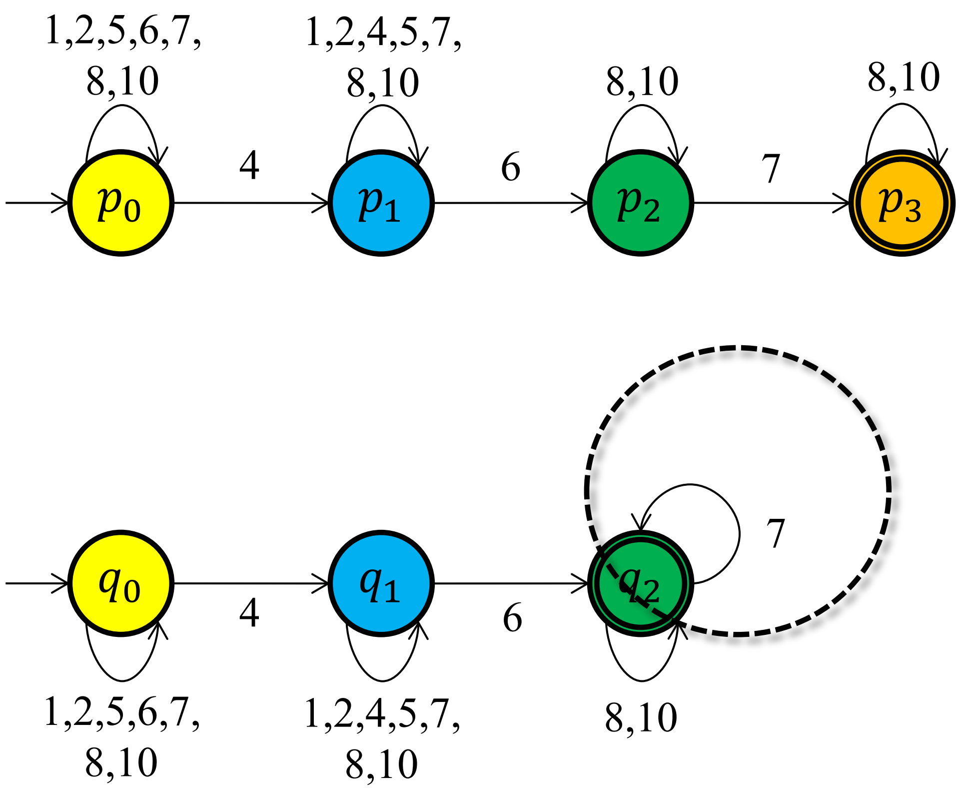
[ESOP'13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse



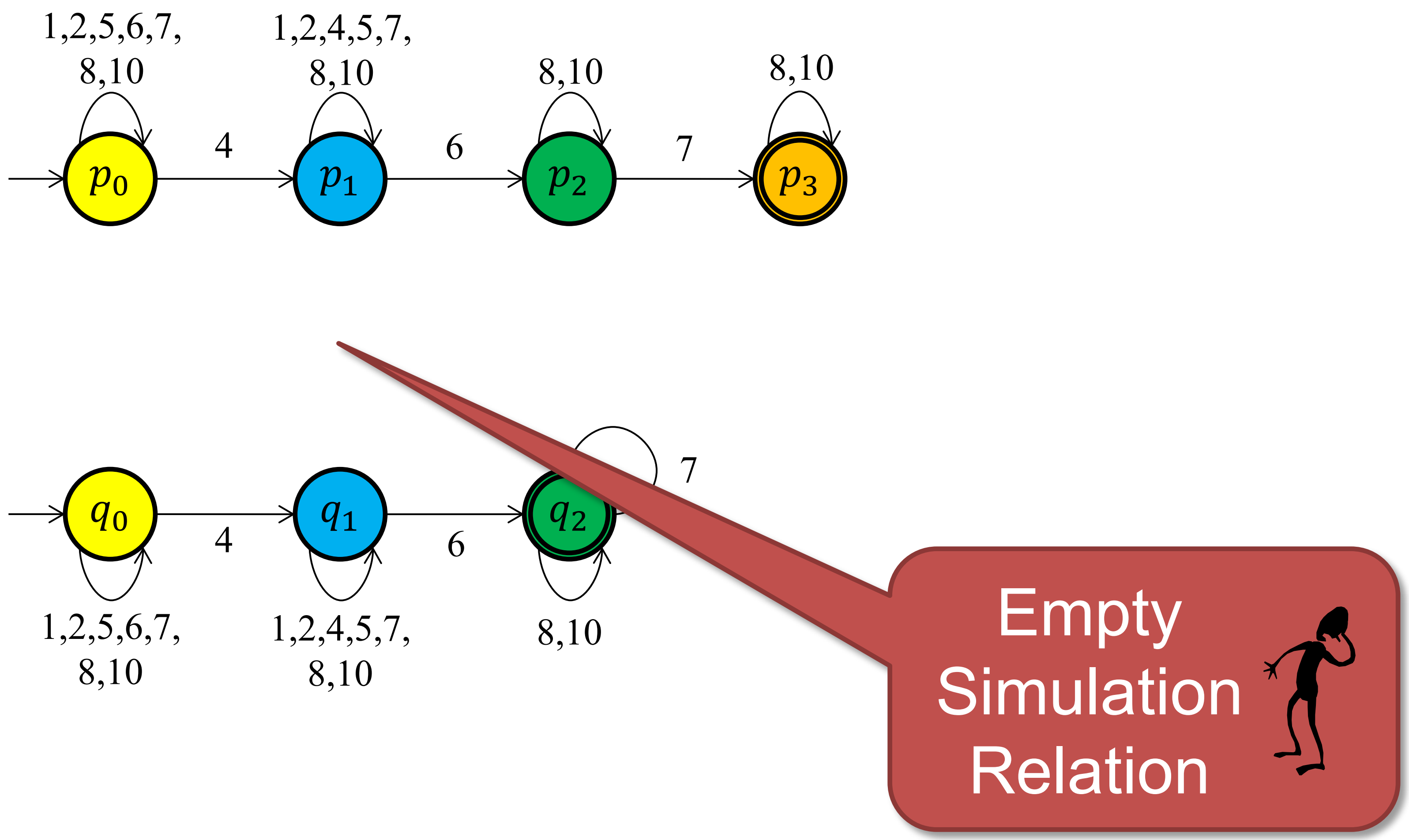
[ESOP'13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse



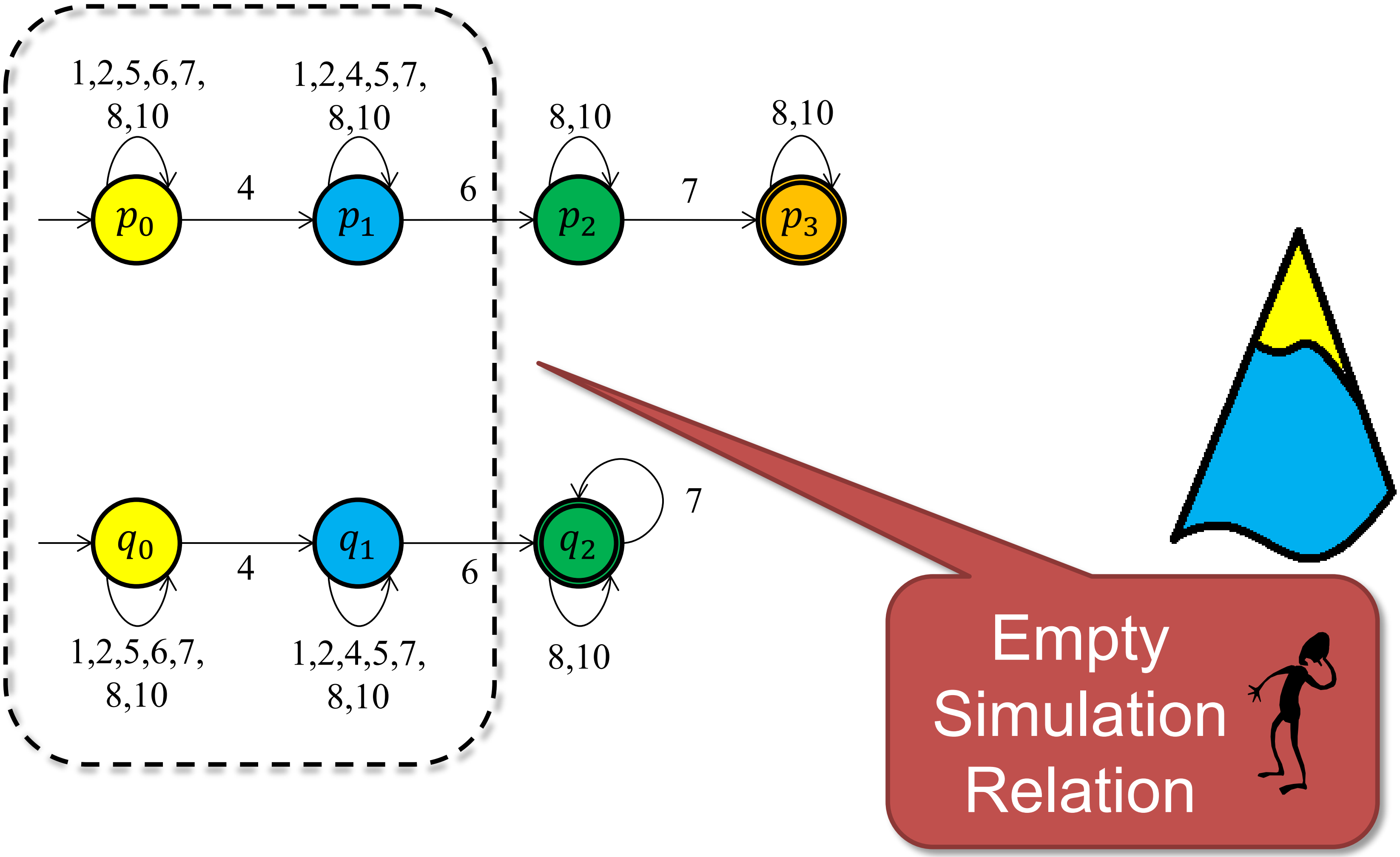
[ESOP'13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse



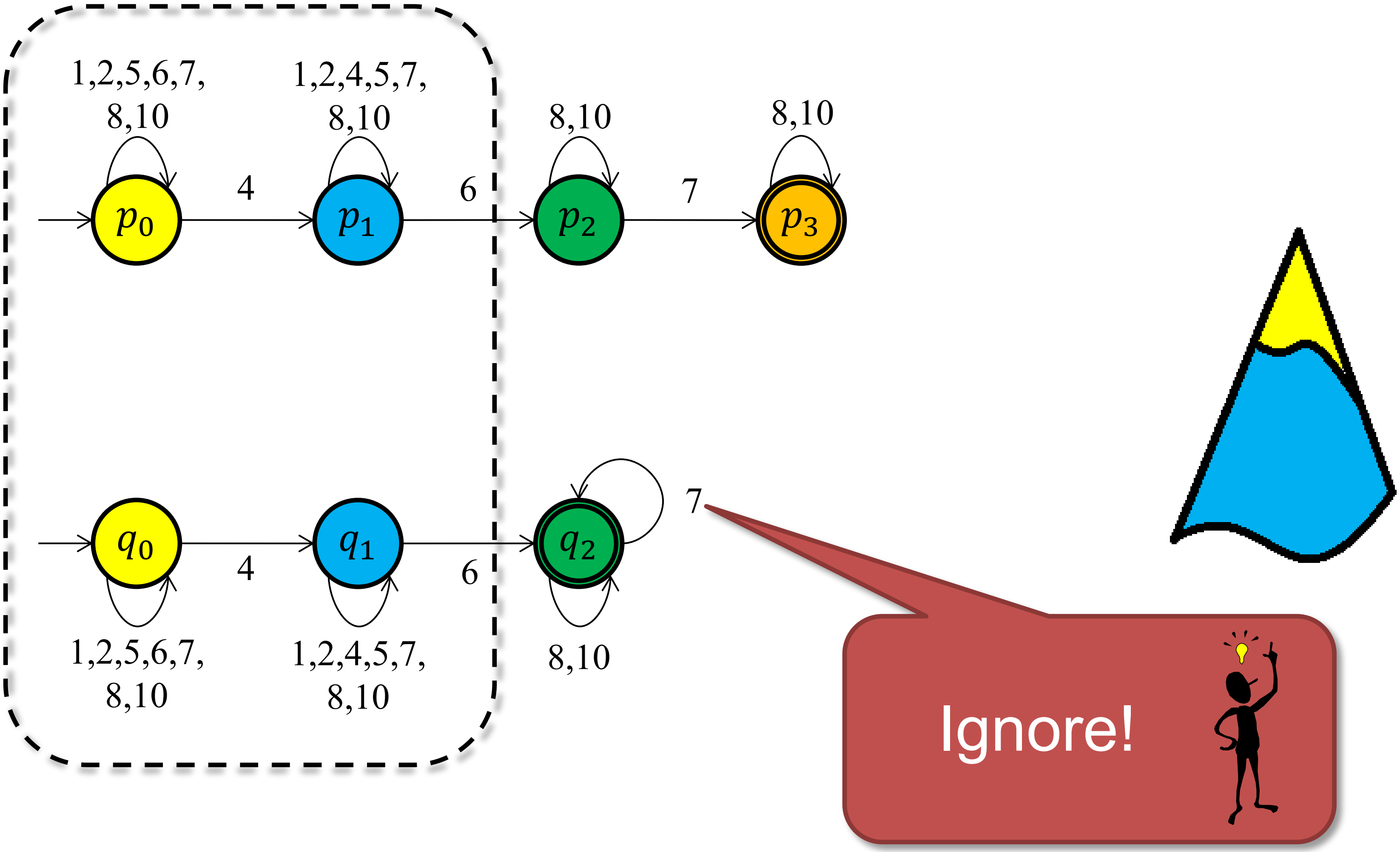
[ESOP‘13] *Beyer, Holzer, Tautschnig, Veith*

Information Reuse



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

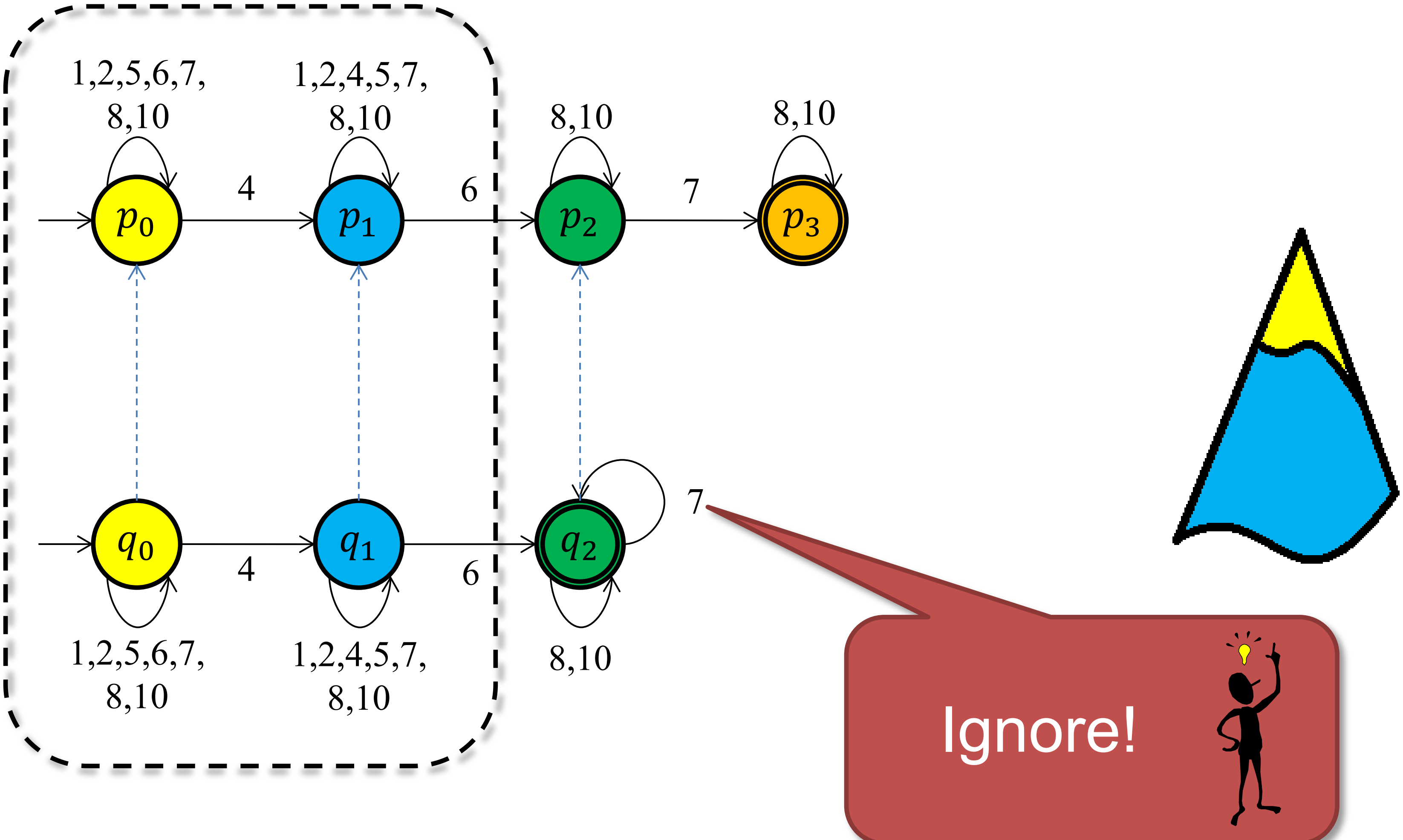
Information Reuse



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

Information Reuse

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



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

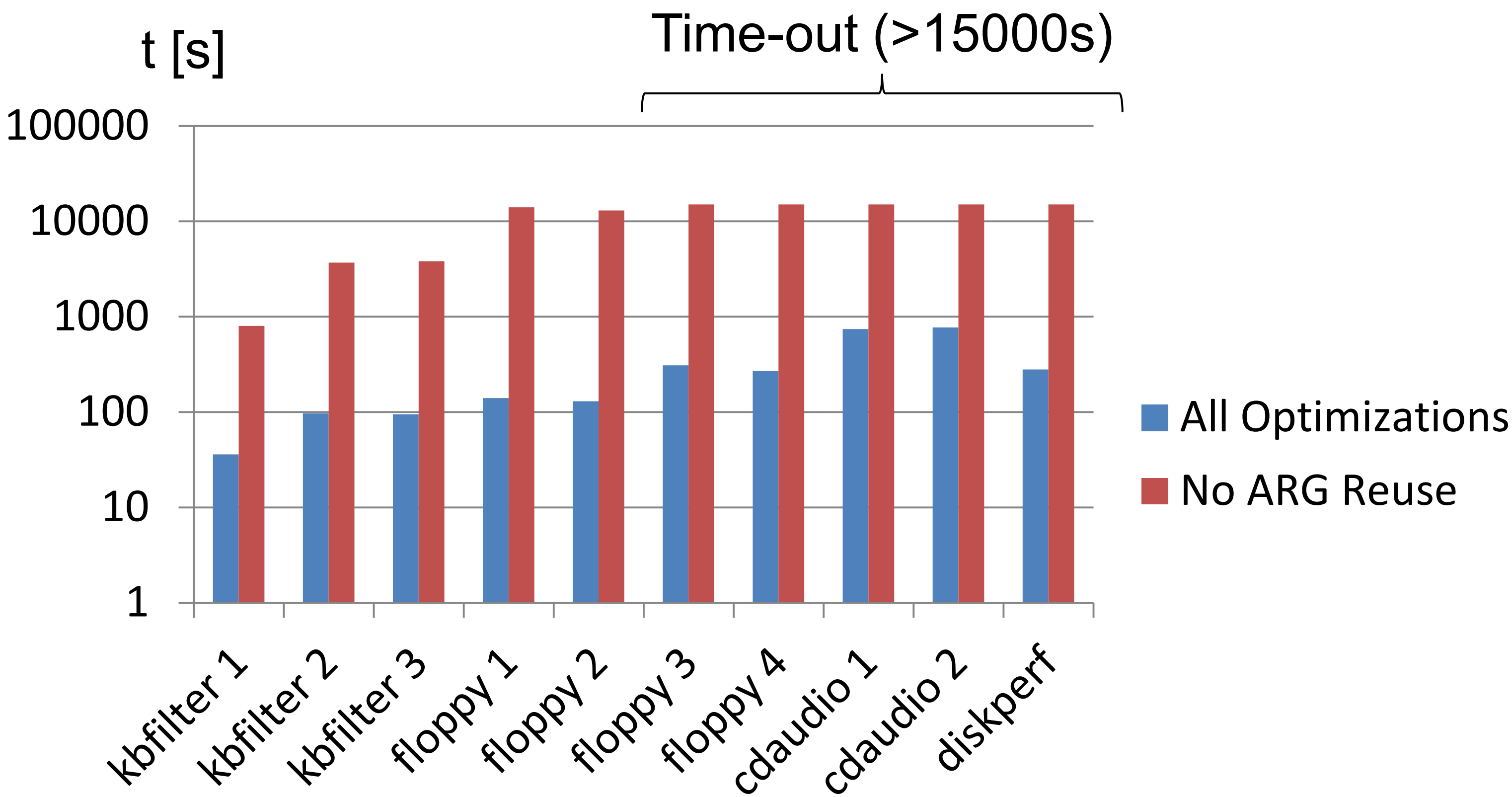
CPAtiger

- 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



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

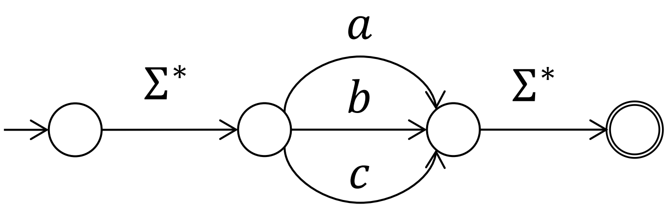
Query-Driven Test Case Generation

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

Semantic Foundations of FQL

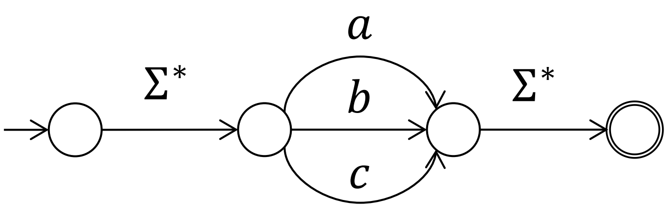


$$
\begin{aligned}
\{\quad & \Sigma^* \cdot \{a\} \cdot \Sigma^* \quad , \\
& \Sigma^* \cdot \{b\} \cdot \Sigma^* \quad , \\
& \Sigma^* \cdot \{c\} \cdot \Sigma^* \quad \}
\end{aligned}
$$

Semantic Foundations of FQL

Regular Sets of Rational Languages (RSRL)

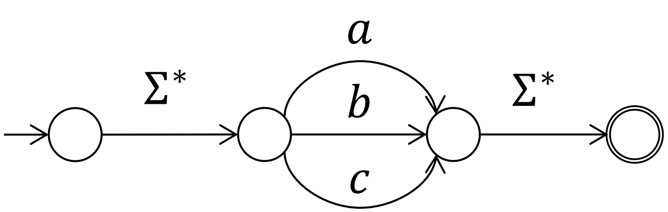
[Afonin and Khazova, 2005]

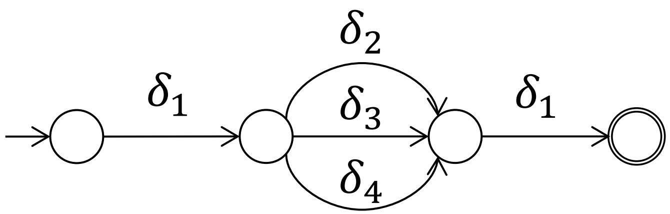


$$
\begin{aligned}
\{\ & \Sigma^* \cdot \{a\} \cdot \Sigma^* \ , \\
& \Sigma^* \cdot \{b\} \cdot \Sigma^* \ , \\
& \Sigma^* \cdot \{c\} \cdot \Sigma^* \ \}
\end{aligned}
$$

Regular Sets of Rational Languages (RSRL)

[Afonin and Khazova, 2005]





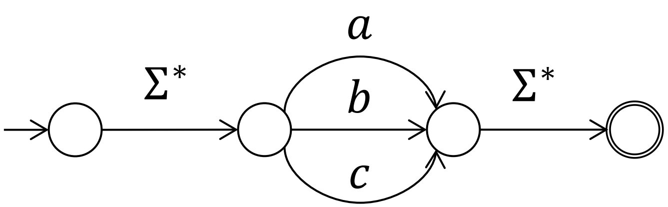
$$\{\ \Sigma^* \cdot \{a\} \cdot \Sigma^*\ ,\quad \Sigma^* \cdot \{b\} \cdot \Sigma^*\ ,\quad \Sigma^* \cdot \{c\} \cdot \Sigma^*\ \}$$

$$K = \{\ \delta_1\delta_2\delta_1\ ,\quad \delta_1\delta_3\delta_1\ ,\quad \delta_1\delta_4\delta_1\ \}$$

Semantic Foundations of FQL

Regular Sets of Rational Languages (RSRL)

[Afonin and Khazova, 2005]



φ maps to a regular language

$$\{ \ \Sigma^* \cdot \{a\} \cdot \Sigma^* \ , \quad \Sigma^* \cdot \{b\} \cdot \Sigma^* \ , \quad \Sigma^* \cdot \{c\} \cdot \Sigma^* \ \}$$

$$K = \{ \ \delta_1\delta_2\delta_1 \ , \quad \delta_1\delta_3\delta_1 \ , \quad \delta_1\delta_4\delta_1 \ \}$$

$$\varphi(\delta_1) = \Sigma^*$$
$$\varphi(\delta_2) = \{a\}$$
$$\varphi(\delta_3) = \{b\}$$
$$\varphi(\delta_4) = \{c\}$$

$$\equiv (K, \varphi)$$

Closure Properties 1

- The complement of an RSRL is **not** an RSRL
- RSRL are closed under product, Kleene star, union

Operator	Finite Case (FQL)	Finite Case, fixed φ (FQL)
$\cdot, \cup, \cap, -$	Closed	Not closed

Under submission: *Holzer , Schallhart, Tautschnig, Veith*

Closure Properties 2

Point-wise Operators

```
cover "@ID*".@BASICBLOCKENTRY."@ID*"
passing @ID*.NOT(@FUNCTION(unimplemented)).@ID*
```

$$\mathcal{R} \cap R = \{ L \cap R \mid L \in \mathcal{R} \}$$

Point-wise Operators	Finite RSRL, fixed φ (FQL)	Finite RSRL (FQL)	RSRL
$^*, \overline{}, \cup, \cap, -$	Not closed	Closed	Not Closed

Under submission: *Holzer , Schallhart, Tautschnig, Veith*

Complexities of Decision Problems

Decision Problem		Kleene-star free case (FQL)	General case
Equivalence	$\mathcal{R}_1 = \mathcal{R}_2$	PSPACE-complete	?
Inclusion	$\mathcal{R}_1 \subseteq \mathcal{R}_2$	PSPACE-complete	?
Membership	$L \in \mathcal{R}$	PSPACE-complete	O(2EXPSPACE)

Under submission: *Holzer , Schallhart, Tautschnig, Veith*

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.