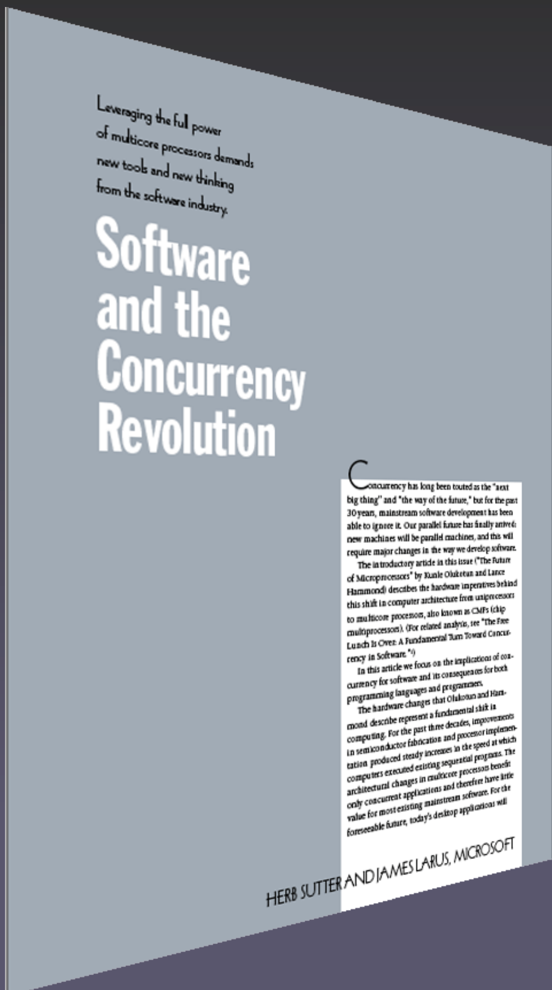
Comparative Assessment of Testing and Model Checking Using Program Mutation

Jeremy S. Bradbury
Faculty of Science • University of Ontario Institute of Technology
Oshawa • Ontario • Canada
jeremy.bradbury@uoit.ca

James R. Cordy, Juergen Dingel
School of Computing • Queen's University
Kingston • Ontario • Canada
{ cordy, dingel }@cs.queensu.ca

Mutation 2007 • Sept. 10-11, 2007

SUPPORTED BY NSERC CRSNG


Leveraging the full power of multicore processors demands new tools and new thinking from the software industry.

Software and the Concurrency Revolution

Concurrency has long been touted as the "next big thing" and "the way of the future," but for the past 30 years, mainstream software development has been able to ignore it. Our parallel future has finally arrived: new machines will be parallel machines, and this will require major changes in the way we develop software.

The introductory article in this issue ("The Future of Microprocessors" by Kunle Olukotun and Lance Hammond) describes the hardware imperatives behind this shift in computer architecture from uniprocessors to multicore processors, also known as CMPs (chip multiprocessors). (For related analysis, see "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.")

In this article we focus on the implications of concurrency for software and its consequences for both programming languages and programmers.

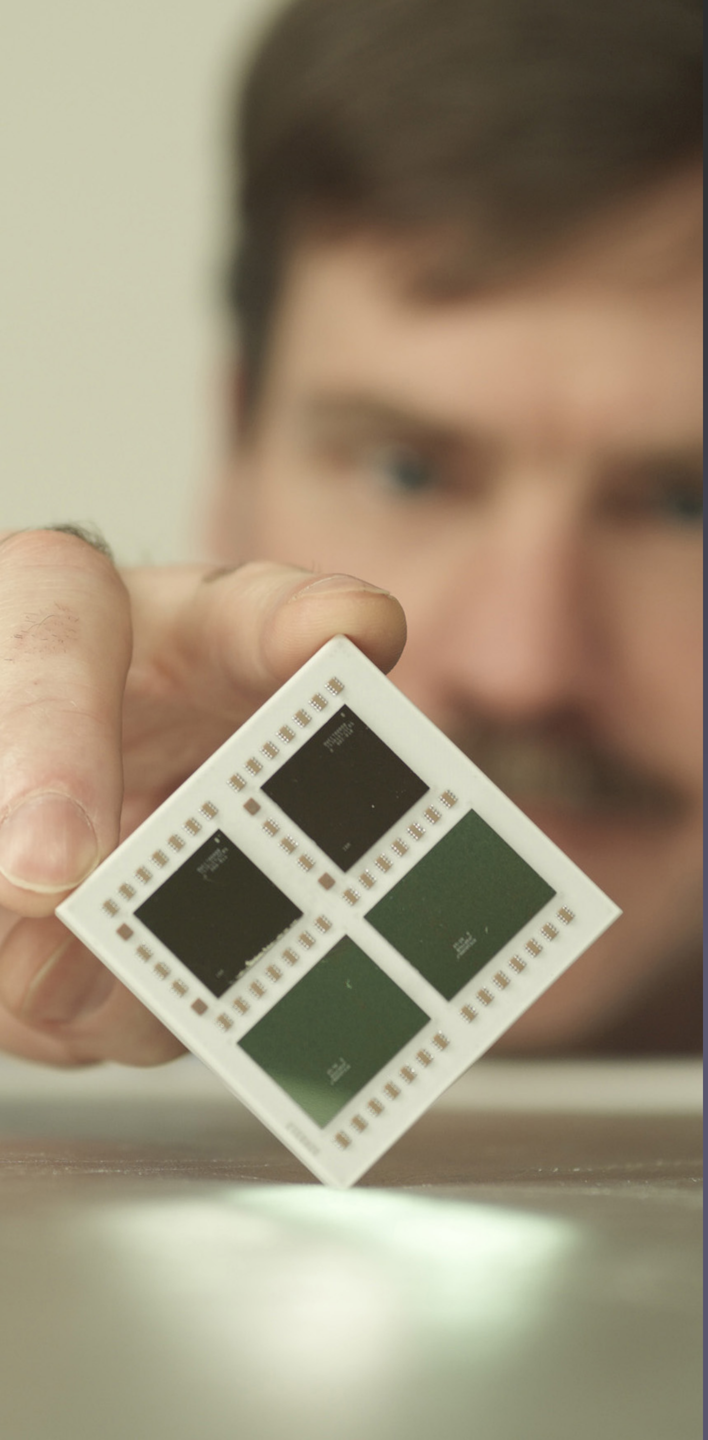
The hardware changes that Olukotun and Hammond describe represent a fundamental shift in computing. For the past three decades, improvements in semiconductor fabrication and processor implementation produced steady increases in the speed at which computers executed existing sequential programs. The architectural changes in multicore processors benefit only concurrent applications and therefore have little value for most existing mainstream software. For the foreseeable future, today's desktop applications will

HERB SUTTER AND JAMES LARUS, MICROSOFT

"...humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings..."

- Herb Sutter & James Larus, Microsoft [SL05]

[SL05] H. Sutter and J. Larus. Software and the concurrency revolution. Queue, 3(7):54–62, 2005.



In the future applications will need to be **concurrent** to fully exploit CPU throughput gains [Sut05]

[Sut05] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), Mar. 2005.

How can we increase our confidence in the correctness of concurrent programs?

Research Goals

1. To compare the effectiveness and efficiency of testing and model checking tools using mutation

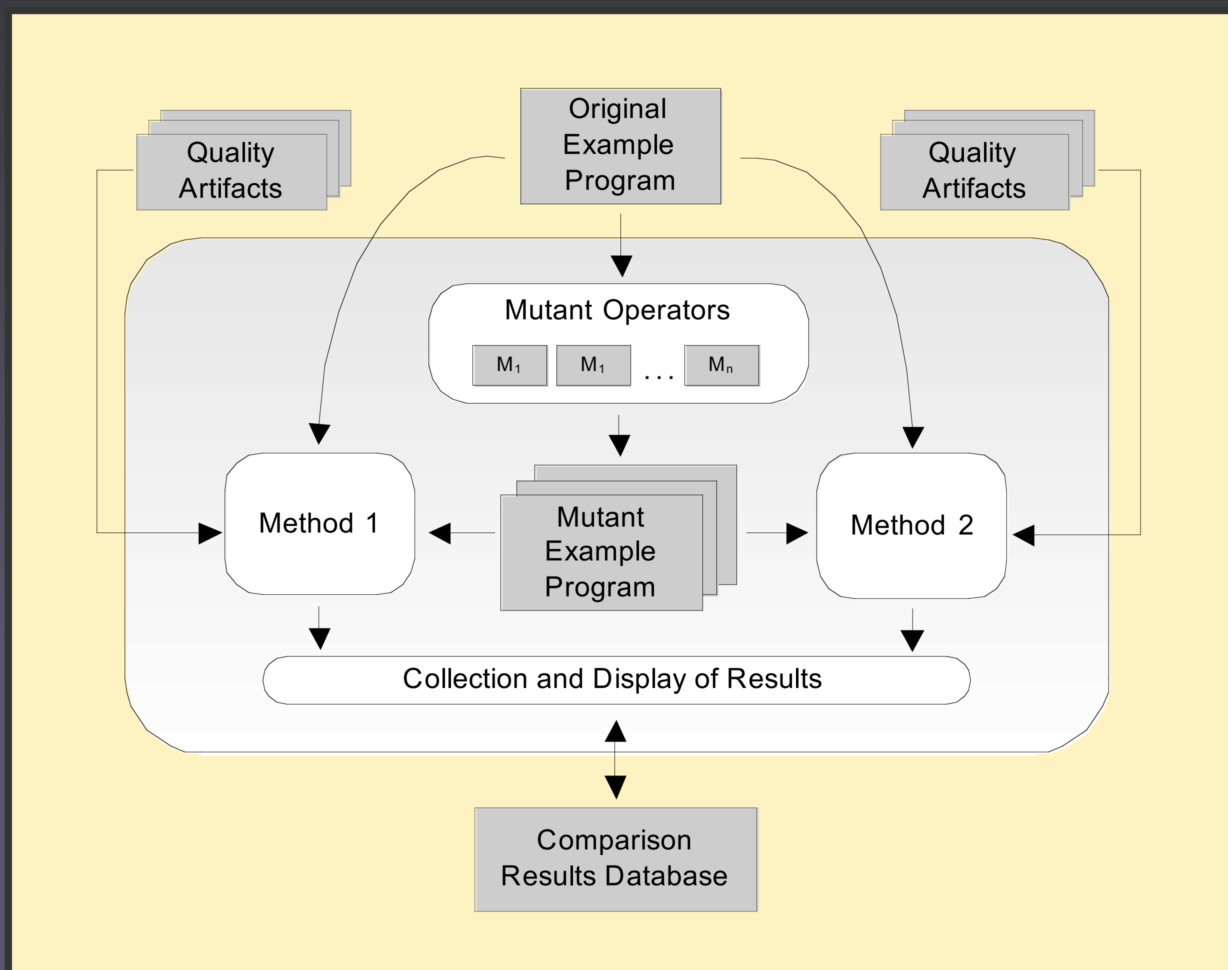
2. To better understand any complementary relationship that might exist between testing and model checking

Our Approach

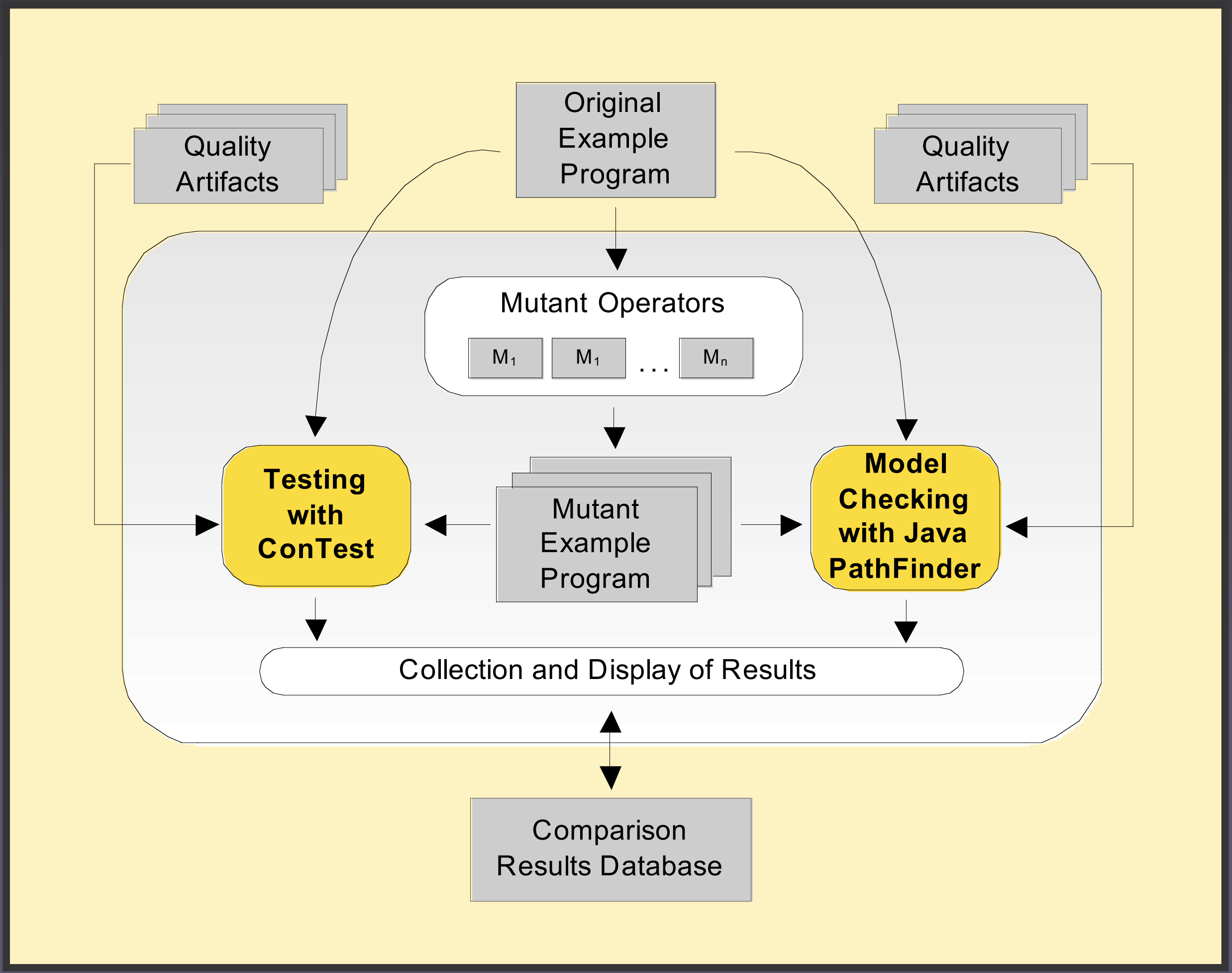
- Conduct a controlled experiment to evaluate the ability of testing and model checking
- We use mutation to generate the faulty concurrent programs required for our experiments
- Mutation [DLS78] traditionally used within the sequential testing community
  - evaluate the effectiveness of test suites

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints for test data selection: help for the practicing programmer. IEEE Computer, 11(4):34–41, Apr. 1978.

Research Methods

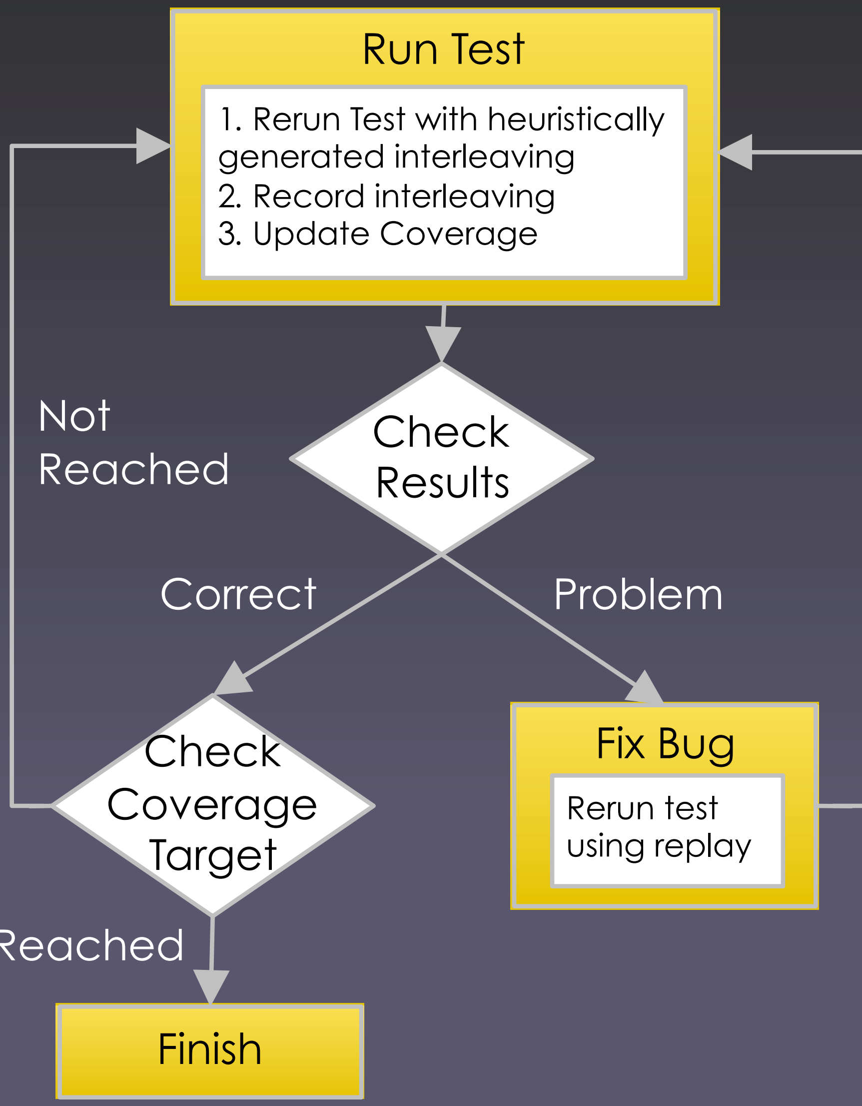


Experimental Setup



Approach Selection

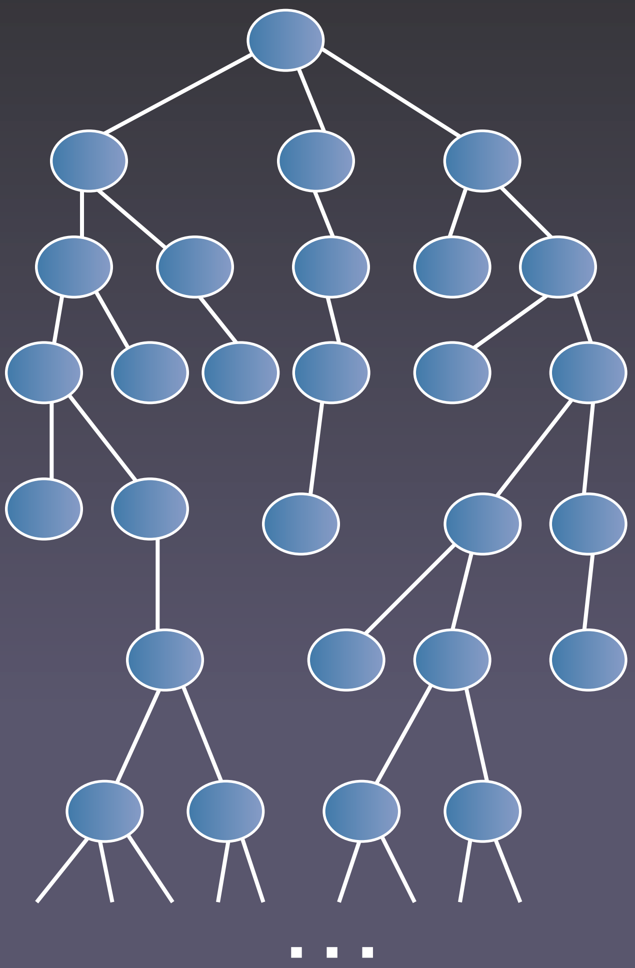
Concurrency Testing with IBM's ConTest



[EFN+02] O. Edelstein, E. Farchi, Y. Nir, G.Ratsaby, and S. Ur. Multithreaded java program test generation. IBM Systems Journal, 41(1):111– 125, 2002.

Model Checking with Java PathFinder (JPF)

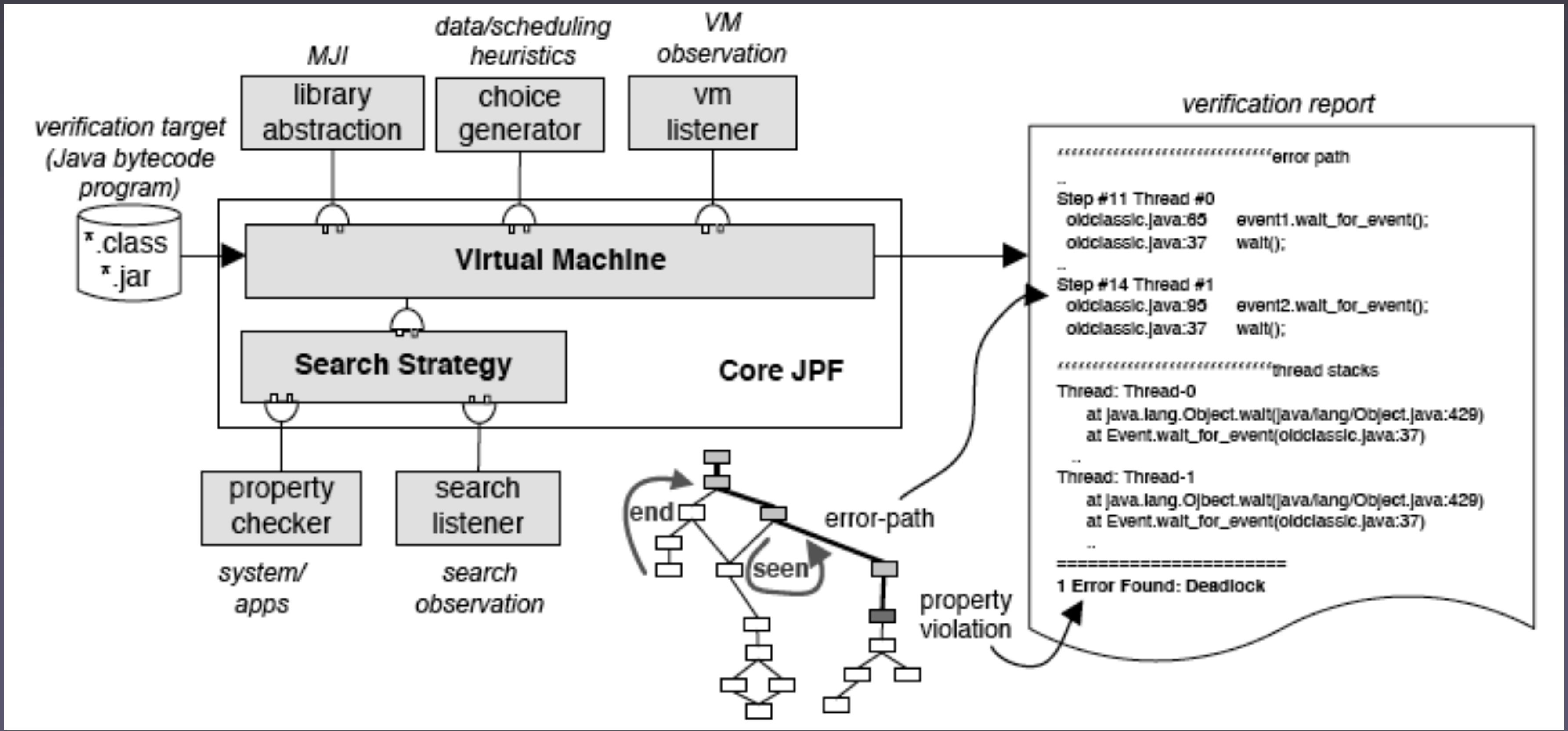
- Model checking exhaustively searches the entire state space of a program (i.e., all interleavings)
- Allows for the analysis of assertions and deadlock detection



[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT), 2(4), Apr. 2000.

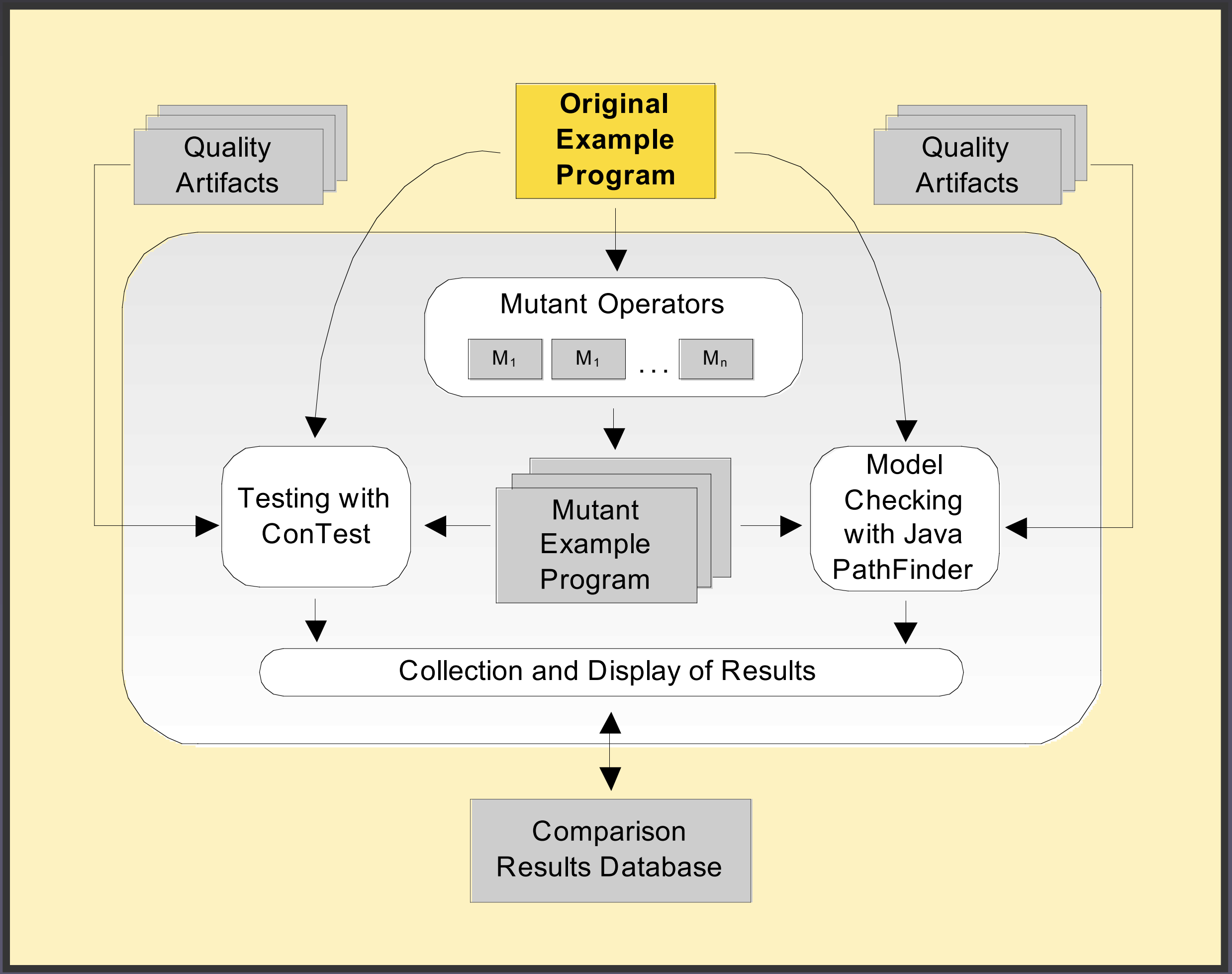
Model Checking with Java PathFinder (JPF)

- Detailed view of JPF architecture



http://javapathfinder.sourceforge.nett

Experimental Setup



Approach Selection

Example Program Selection

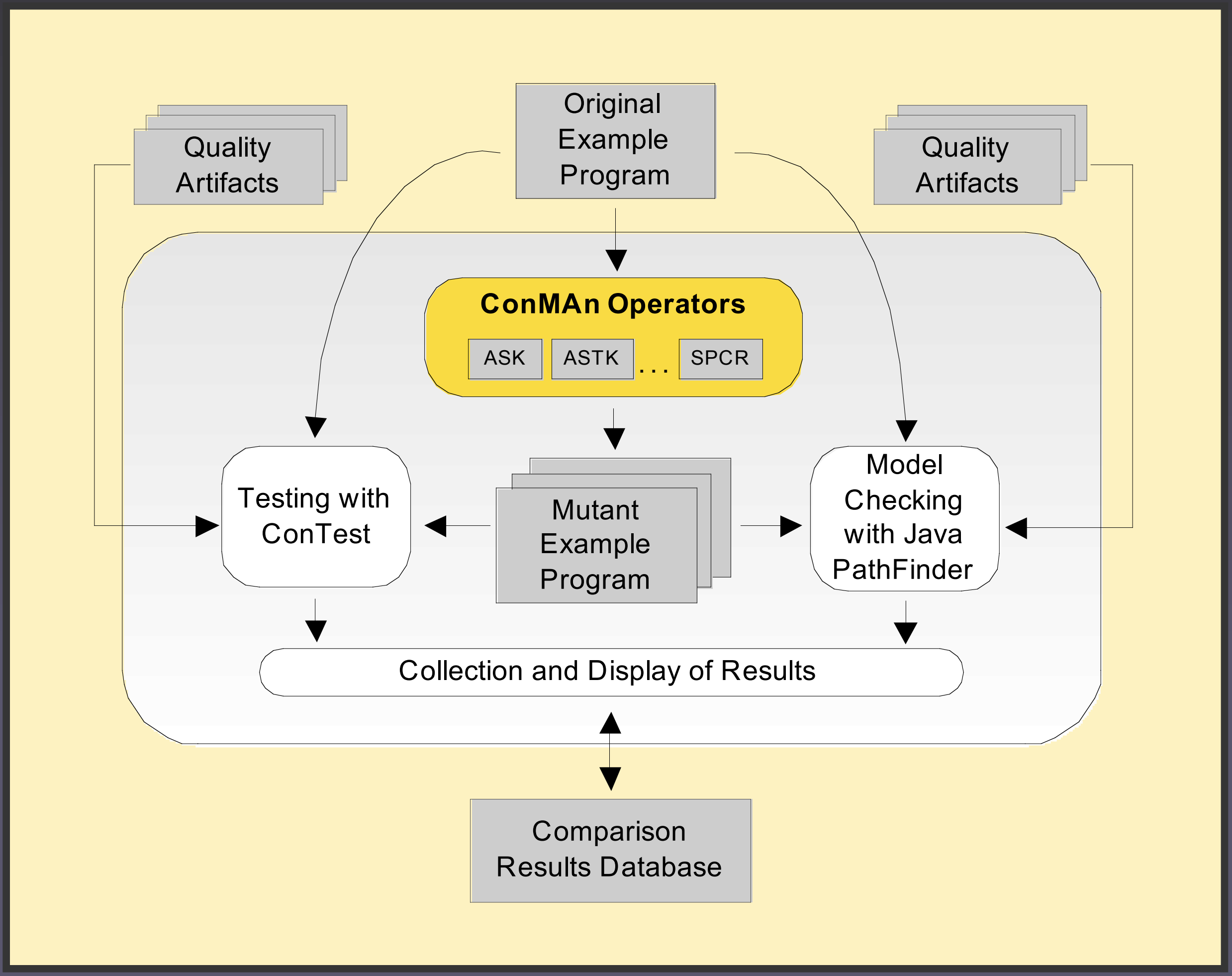
Example Programs

- Ticket Order Simulation
  - Simulates multiple agents selling tickets for a flight
- Linked List
  - Involves storing data in a concurrent linked list (data structure)
- Buffered Writer
  - Two different types of writer threads are updated a buffer that is being read by a reader thread
- Account Management System
  - Manages a series of transactions between a number of accounts

Metrics for the Example Programs

Example Program	Lines of Code	Statements	Critical Regions	Critical Region Statements
TicketsOrderSim	75	21	1	6 (28.5%)
LinkedList	303	70	2	4 (5.7%)
BufWriter	213	55	3	20 (36.4%)
AccountProgram	145	40	5	8 (20%)

Experimental Setup



Approach Selection

Example Program Selection

Mutation Selection

The ConMAn Operators [BCD06a]

- **ConMAn** = **Con**currency **M**utation **An**alysis
- What are the ConMAn operators?
  - *"…a comprehensive set of 24 operators for Java that are representative of the kinds of bugs that often occur in concurrent programs."*
  - based on an existing fault model for Java concurrency [FNU03]
- Can be used as a comparative metric
- In this experiment we used a subset of the operators for Java 1.4.

[BCD06a] J.S. Bradbury, J.R. Cordy, J. Dingel. Mutation operators for concurrent Java (J2SE 5.0). In. *Proc. of Mutation 2006.*
[FNU03] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. of IPDPS 2003.*

Example ConMAn Mutation

SKCR – Shrink Critical Region

```
Object lock1  = new Object();
...
public void m1 () {
   <statement n1>
  synchronized (lock1) {
      //critical region
      <statement c1>
      <statement c2>
      <statement c3>
   }
  <statement n2>
...
```

Example ConMAn Mutation

SKCR – Shrink Critical Region

```
Object lock1  = new Object();
...
public void m1 () {
   <statement n1>
  synchronized (lock1) {
      //critical region
      <statement c1>
      <statement c2>
      <statement c3>
   }
  <statement n2>
...
```

```
Object lock1  = new Object();
...
public void m1 () {
   <statement n1>
   //critical region
   <statement c1>
   synchronized (lock1) {
      <statement c2>
   }
   <statement c3>
   <statement n2>
...
```

Example ConMAn Mutation
SKCR – Shrink Critical Region

```
Object lock1  = new Object();
...
public void m1 () {
   <statement n1>
  synchronized (lock1) {
      //critical region
      <statement c1>
      <statement c2>
      <statement c3>
  }
  <statement n2>
...
```

```
Object lock1  = new Object();
...
public void m1 () {
   <statement n1>
   //critical region
   <statement c1>
   synchronized (lock1) {
      <statement c2>
   }
   <statement c3>
   <statement n2>
...
```

No Lock Bug!

Example ConMAn Mutation

ESP – Exchange Synchronized Block Parameters

```
Object lock1  = new Object();
Object lock2  = new Object();
...
synchronized (lock1) {
 <statement c1>
  ...
  synchronized (lock2) {
 <statement c2>
     ...
  }
}
...
```

Example ConMAn Mutation

ESP – Exchange Synchronized Block Parameters

```
Object lock1  = new Object();
Object lock2  = new Object();
...
synchronized (lock1) {
 <statement c1>
  ...
  synchronized (lock2) {
 <statement c2>
     ...
  }
}
...
```

```
Object lock1  = new Object();
Object lock2  = new Object();
...
synchronized (lock2) {
 <statement c1>
  ...
  synchronized (lock1) {
 <statement c2>
     ...
  }
}
...
```

Example ConMAn Mutation

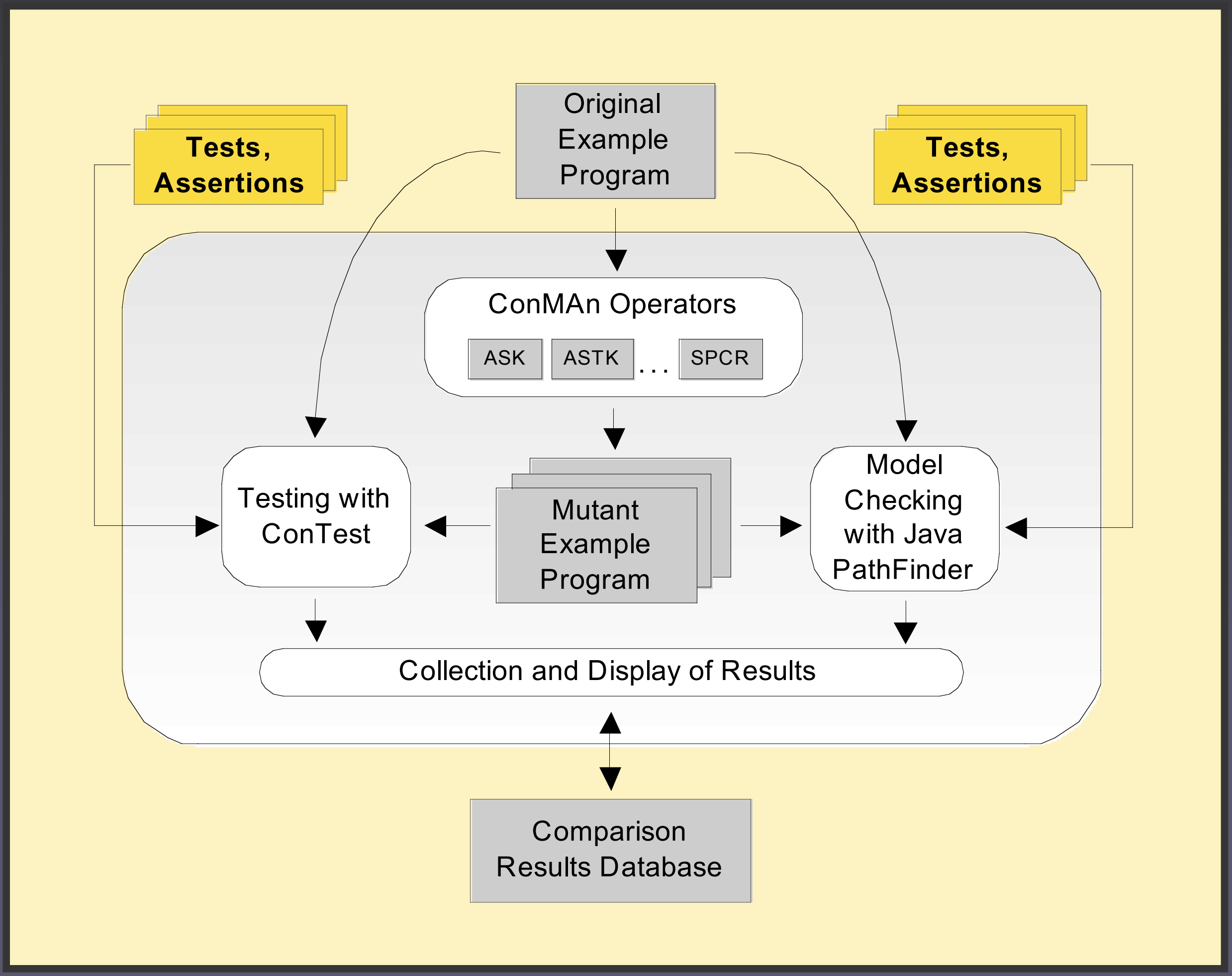
ESP – Exchange Synchronized Block Parameters

```
Object lock1  = new Object();
Object lock2  = new Object();
...
synchronized (lock1) {
 <statement c1>
  ...
   synchronized (lock2) {
 <statement c2>
      ...
   }
}
...
```

```
Object lock1  = new Object();
Object lock2  = new Object();
...
synchronized (lock2) {
 <statement c1>
  ...
   synchronized (lock1) {
 <statement c2>
      ...
   }
}
...
```

Deadlock bug!

Experimental Setup



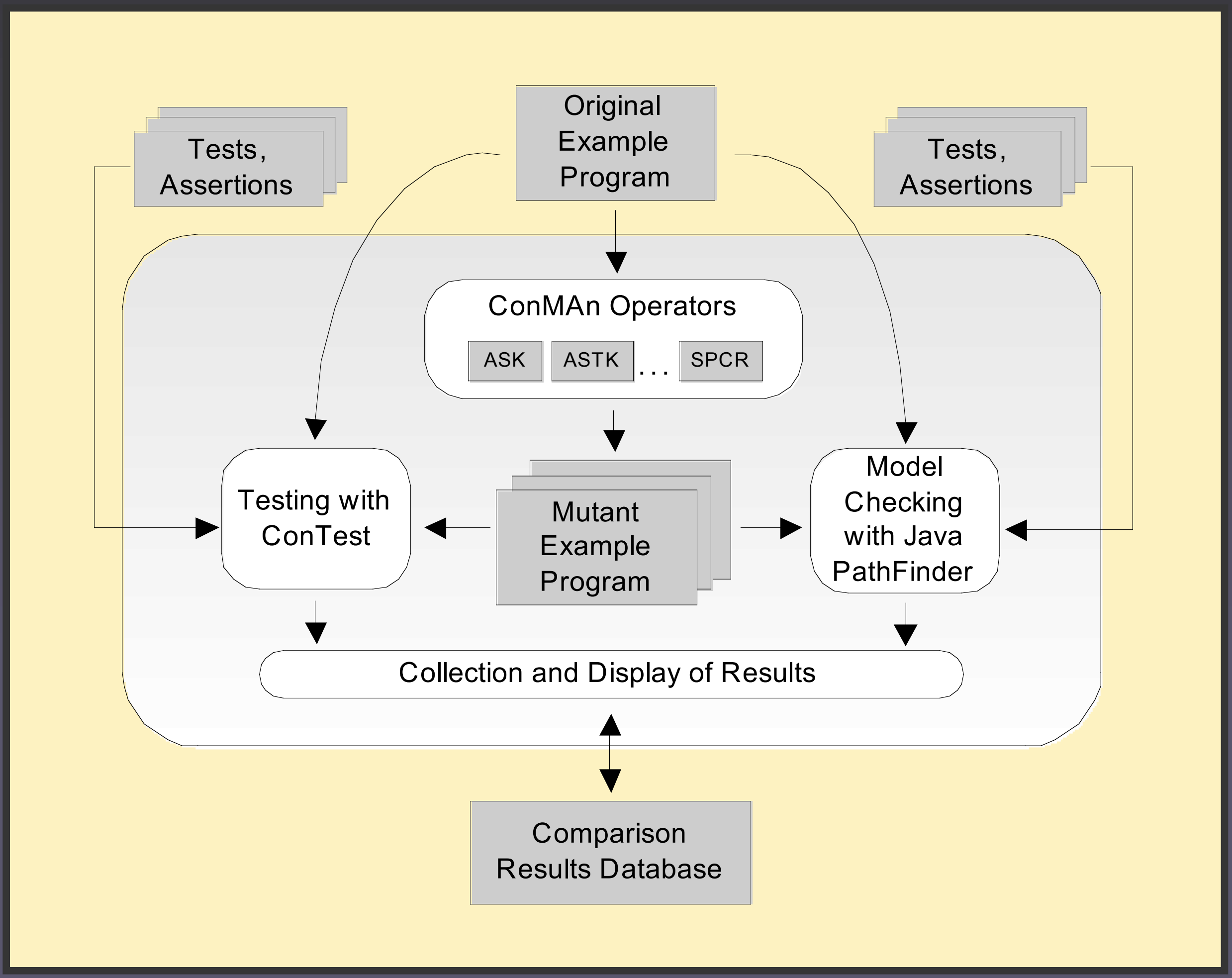
Approach Selection

Example Program Selection

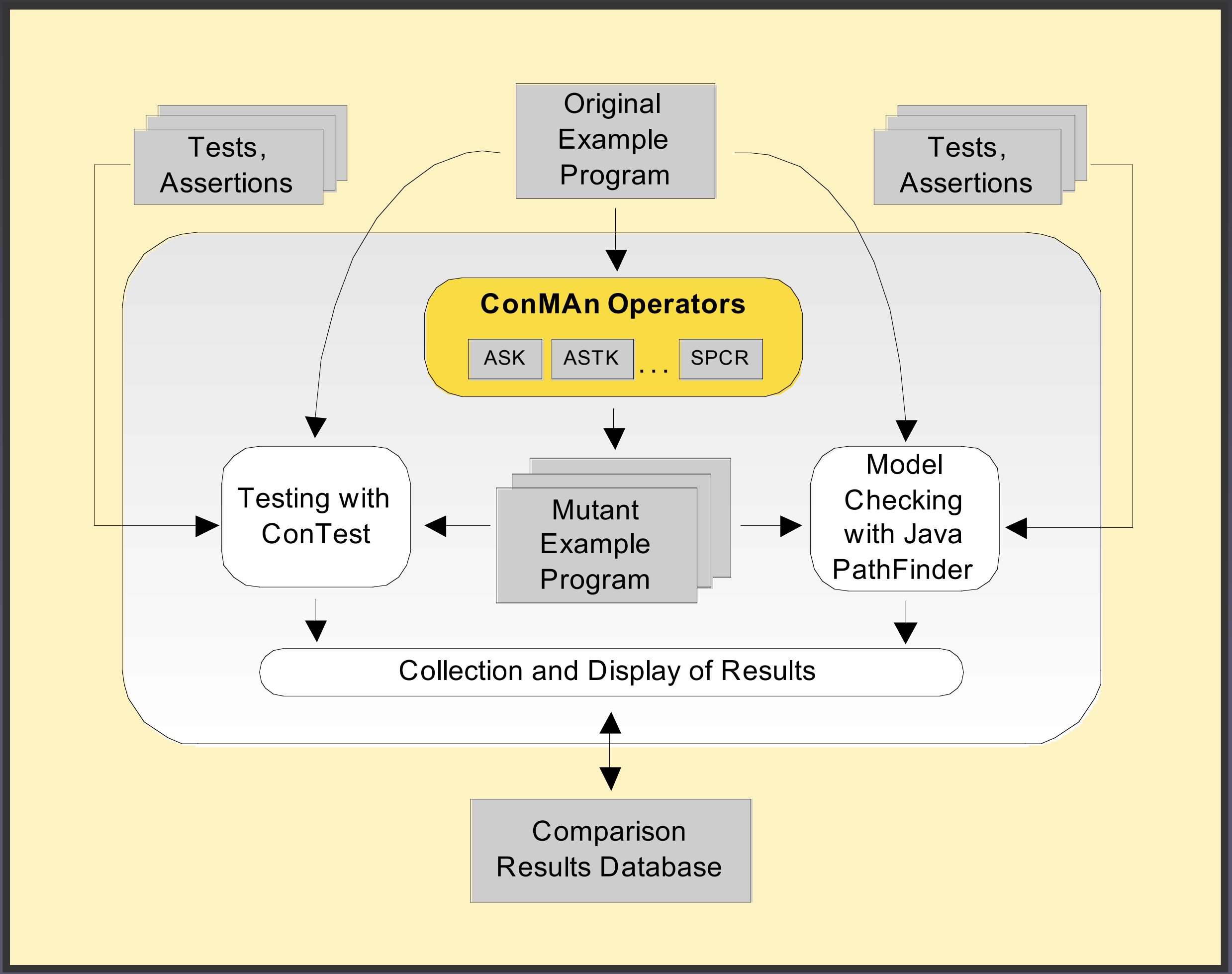
Mutation Selection

Program Artifact Selection

Experimental Procedure

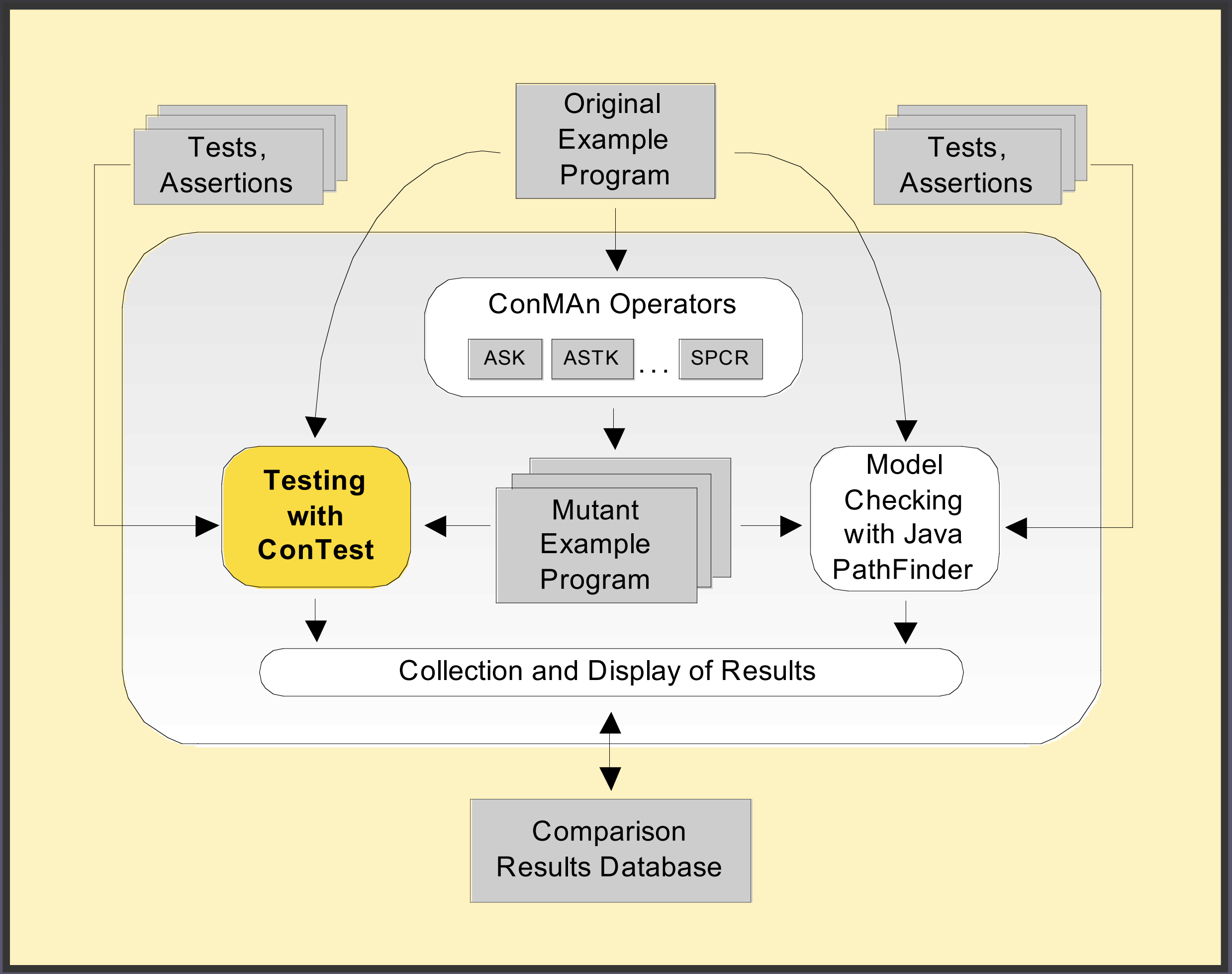


Experimental Procedure



Mutant Generation

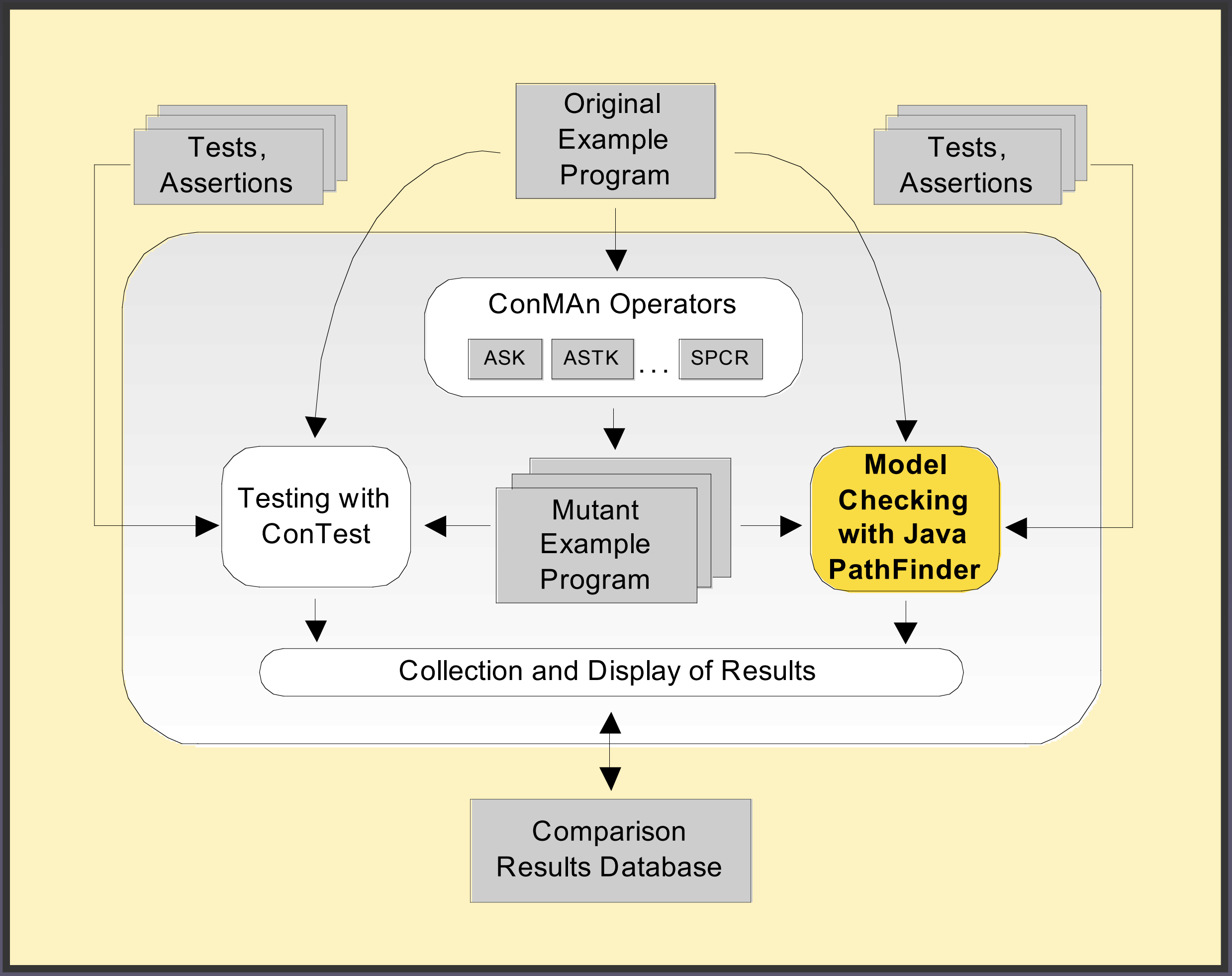
Experimental Procedure



Mutant Generation

Testing

Experimental Procedure

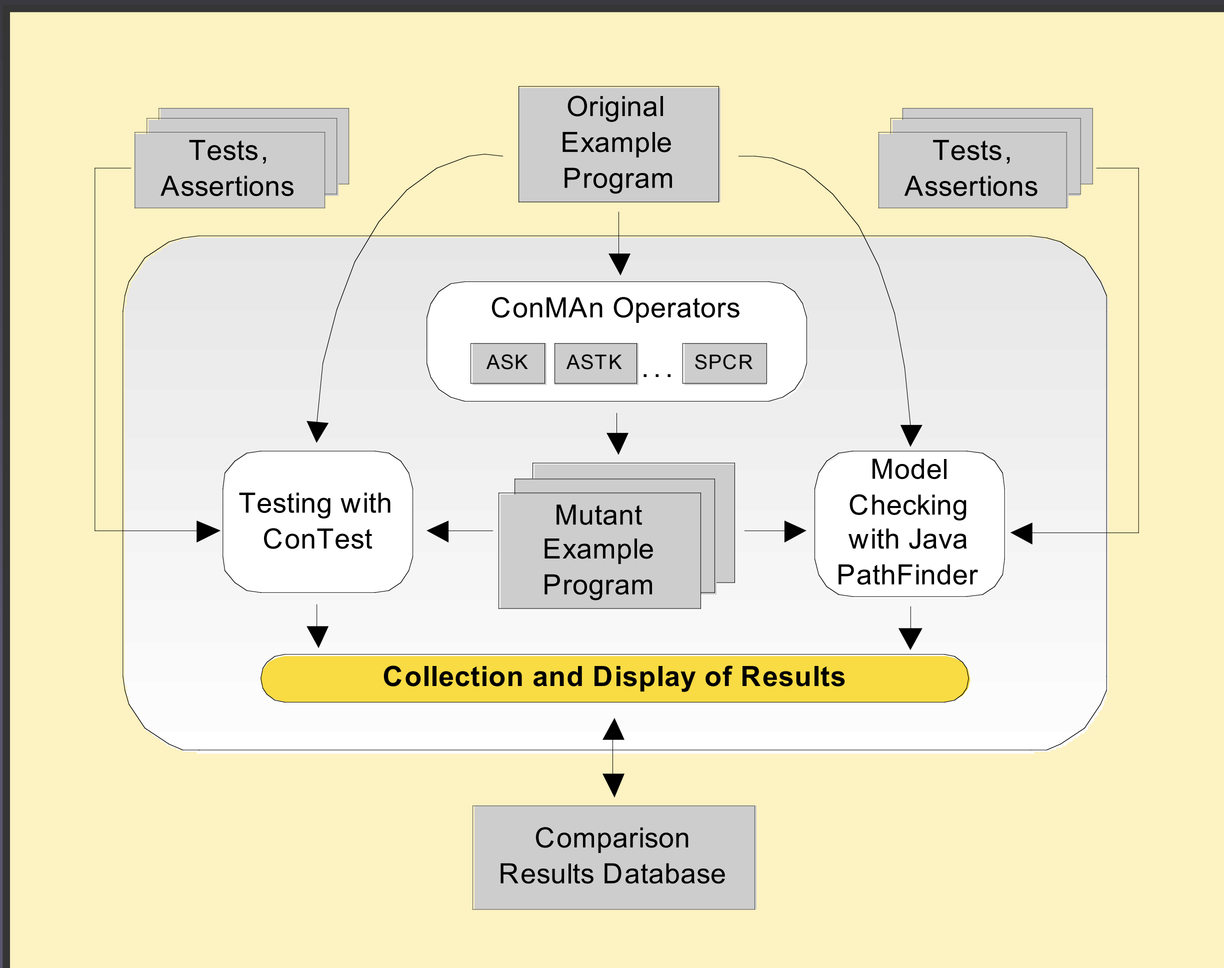


Mutant Generation

Testing

Model Checking

Experimental Procedure



Mutant Generation

Testing

Model Checking

Collection and Display of Result

The ExMAn Framework [BCD06b]

- **ExMAn** = **Ex**perimental **M**utation **An**alysis

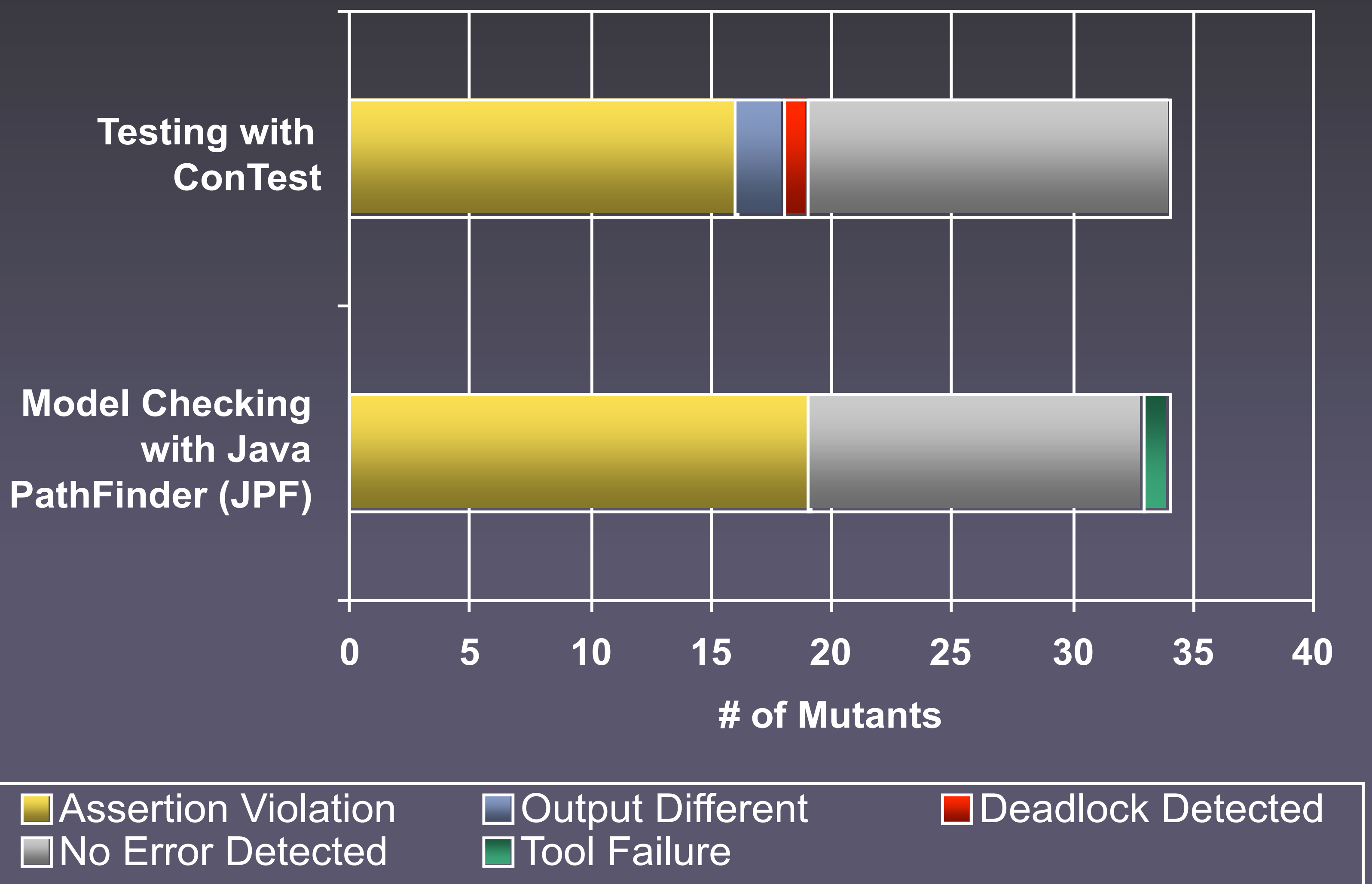
- What is ExMAn?
  – *"ExMAn is a reusable implementation for building different customized mutation analysis tools for comparing different quality assurance techniques."*
  – ExMAn automates the experimental procedure

[BCD06b] J.S. Bradbury, J.R. Cordy, J. Dingel. ExMAn: A generic and customizable framework for experimental mutation analysis. In. *Proc. of Mutation 2006.*

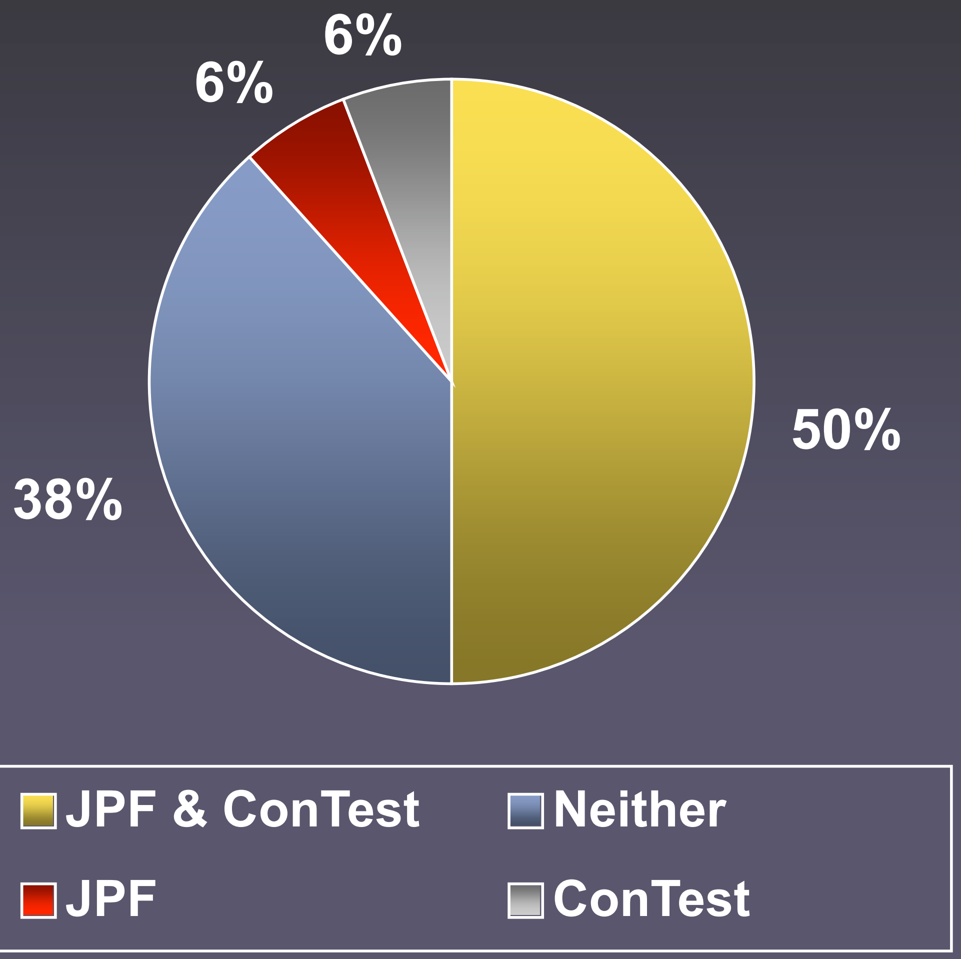
ConTest vs. Java PathFinder

- How do we better understand the **effectiveness** of each technique?
  - We measure the mutant score for each technique (dependent variable)
  - We vary the analysis technique (factor)
  - We fix all other independent variables
    - quality artifacts (tests and properties), example programs …

Quantity of Mutants Killed



Detection of Mutants



Mutant Scores of JPF, ConTest and ConTest +JPF

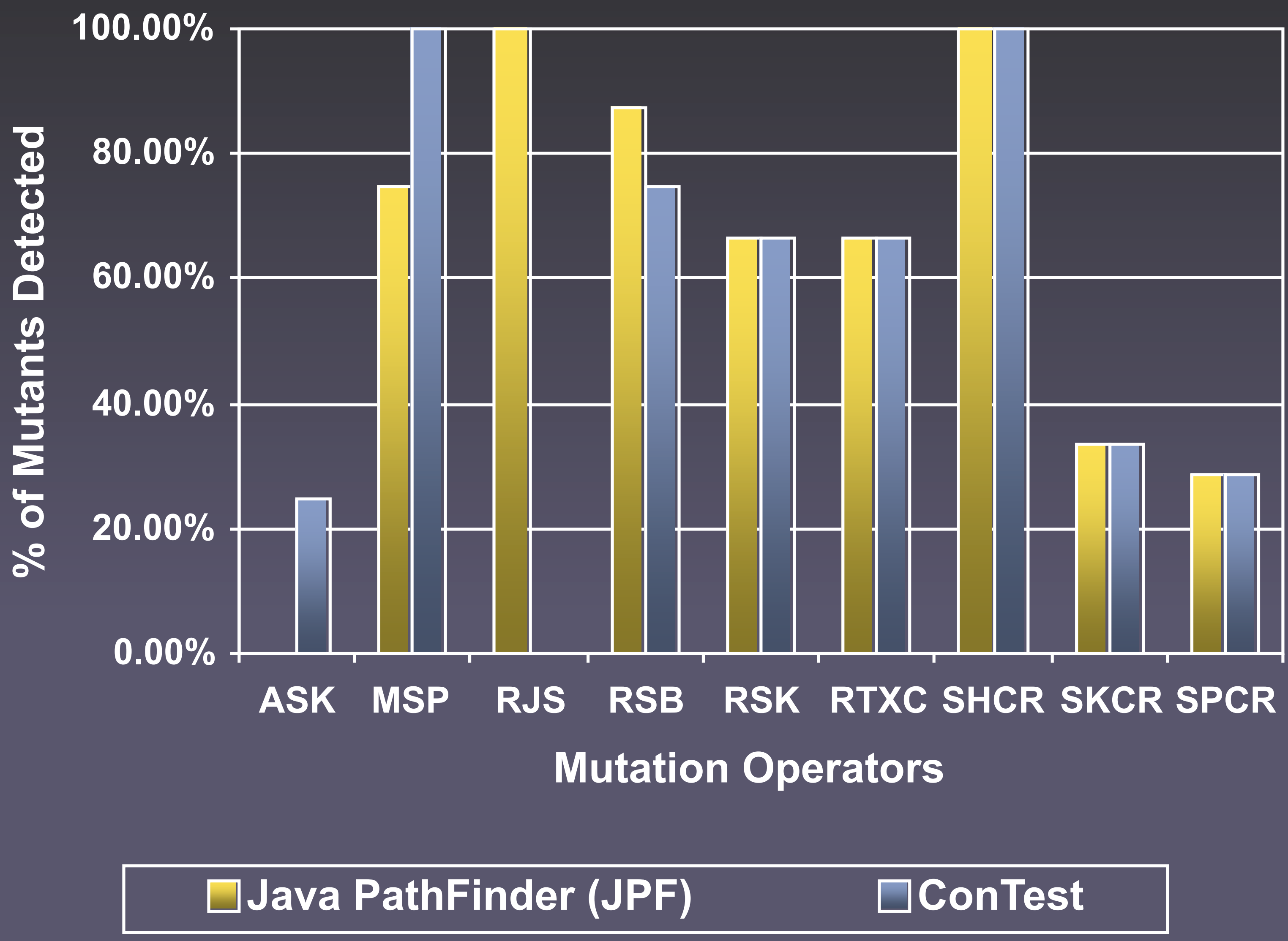
Example Program	ConTest Mutant Score	JPF Mutant Score	ConTest+JPF Mutant Score
BufWriter	38.9%	50%	50%
LinkedList	50%	50%	50%
TicketsOrderSim	100%	100%	100%
AccountProgram	78%	56%	78%
TOTAL	**56%**	**56%**	**62%**

Mutant Scores of JPF, ConTest and ConTest +JPF

Example Program	ConTest Mutant Score	JPF Mutant Score	ConTest+JPF Mutant Score
BufWriter	38.9%	50%	50%
LinkedList	50%	50%	50%
TicketsOrderSim	100%	100%	100%
AccountProgram	78%	56%	78%
TOTAL	**56%**	**56%**	**62%**

ConTest and JPF are most likely **alternative** fault detection techniques with respect to the example programs.

Mutant Score for each Operator



ConTest vs. Java PathFinder

- How do we better understand the **efficiency** of each technique?
  - If ConTest and Java PathFinder are both capable of finding a fault in a program is either of them faster?

ConTest vs. Java PathFinder

- **Experimental Setup**
  - *null hypothesis (H_0):* Time to detect a fault for JPF > Time to detect a fault for ConTest
  - *dependent variable(s):* analysis time
  - *independent variables:*
    - *factor:* analysis technique
    - *fixed:* quality artifacts (tests and properties) software under evaluation

ConTest vs. Java PathFinder

- Time for ConTest (seconds)
  - Mean = 2.0314
  - Median = 1.2030
- Time for Java PathFinder (seconds)
  - Mean = 3.2835
  - Median = 2.3320
- Conducted a paired t-test for n=19
  - P-value = 0.0085 (reject H_O at the 0.05 level)
  - JPF is not more efficient than ConTest for our example programs

Threats to Validity

- internal validity
- external validity:
  - Threats to external validity include:
    - the software being experimented on is not representative of concurrent Java programs in general
    - The configurations of Java PathFinder and ConTest limit our ability to generalize to each approach
- construct validity
- conclusion validity

Conclusions

- For our example programs…
  - Effectiveness: ConTest and Java PathFinder are most likely alternatives (potential to be used with other examples in a complementary way).
  - Efficiency: ConTest is more efficient and can kill a mutant in less time on average than Java PathFinder.
- Future work is further empirical studies in order to generalize our conclusions. ☺

Comparative Assessment of Testing and Model Checking Using Program Mutation

Research Talk

Jeremy S. Bradbury
Faculty of Science • University of Ontario Institute of Technology
Oshawa • Ontario • Canada
jeremy.bradbury@uoit.ca

James R. Cordy, Juergen Dingel
School of Computing • Queen's University
Kingston • Ontario • Canada
{ cordy, dingel }@cs.queensu.ca

Mutation 2007 • Sept. 10-11, 2007

SUPPORTED BY NSERC CRSNG