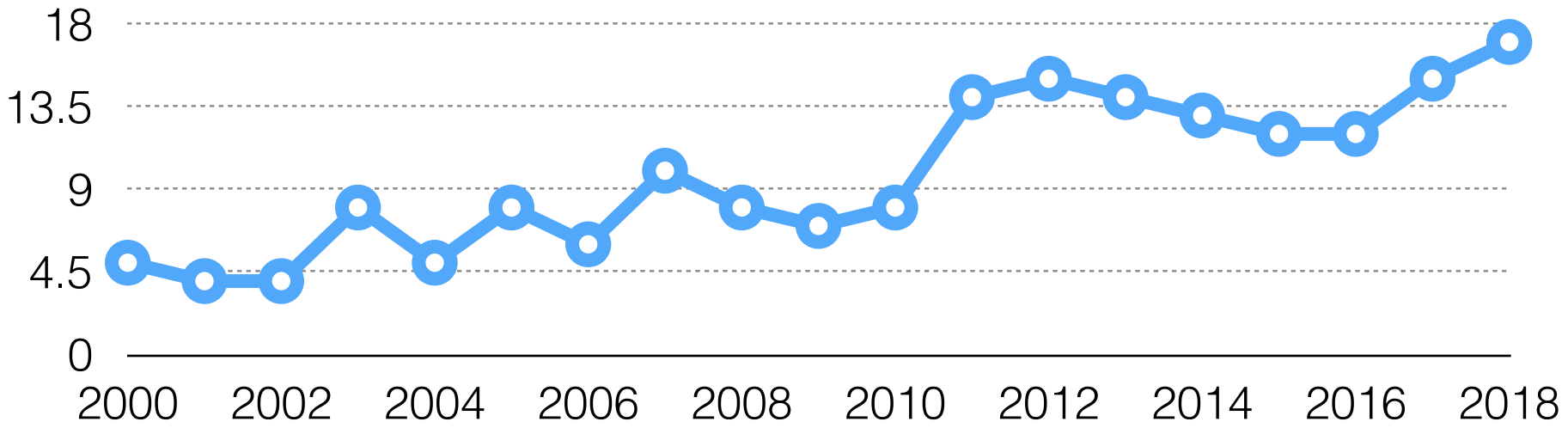
State of the Art and Open Issues in Software Testing


SOFTWARE
IS BUGGY!
Report...
Reopen
Ignore
Report...
Reopen

Software Testing

and investigated

Most used approach

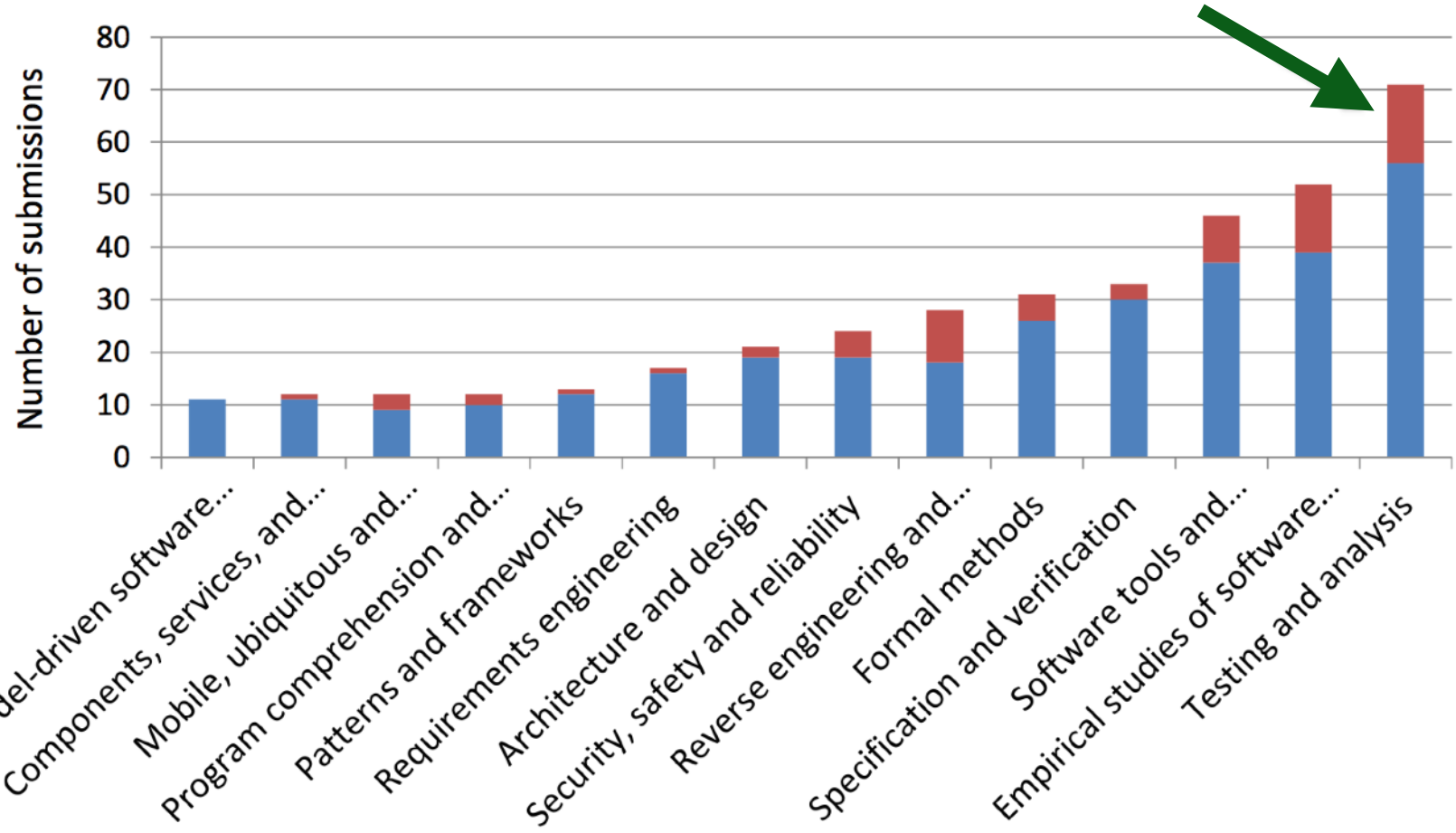


Number of papers on testing at ICSE from 2000 to 2018

Software Testing

and investigated

Most used approach



Software Testing: A Research Travelogue (2000–2014)

Alessandro Orso
College of Computing
School of Computer Science
Georgia Institute of Technology
Atlanta, GA, USA
orso@cc.gatech.edu

Gregg Rothermel
Department of Computer Science
and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
grother@cse.unl.edu

ABSTRACT

Despite decades of work by researchers and practitioners on numerous software quality assurance techniques, testing remains one of the most widely practiced and studied approaches for assessing and improving software quality. Our goal, in this paper, is to provide an accounting of some of the most successful research performed in software testing since the year 2000, and to present what appear to be some of the most significant challenges and opportunities in this area. To be more inclusive in this effort, and to go beyond our own personal opinions and biases, we began by contacting over 50 of our colleagues who are active in the testing research area, and asked them what they believed were (1) the most significant contributions to software testing since 2000 and (2) the greatest open challenges and opportunities for future research in this area. While our colleagues' input (consisting of about 30 responses) helped guide our choice of topics to cover and ultimately the writing of this paper, we by no means claim that our paper represents all the relevant and noteworthy research performed in the area of software testing in the time period considered—a task that would require far more space and time than we have available. Nevertheless, we hope that the approach we followed helps this paper better reflect not only our views, but also those of the software testing community in general.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Algorithms, Experimentation, Verification

Keywords: Software testing

PROLOGUE

In 2000, the International Conference on Software Engineering held its first "Future of Software Engineering" track, and featured a number of papers described as "Roadmaps". These papers offered assessments, by authors expert in various areas of software engineering, of the directions we could expect research in those areas to take. Among the papers and presentations was one entitled, "Testing: A Roadmap", authored by Mary Jean Harrold.

In the spring of 2013, we (Alex and Gregg) were contacted by the organizers of the FOSE track for ICSE 2014, Matthew Dwyer and James Herbsleb. They told us that, in addition to the usual "Roadmaps", they wanted to include in the FOSE track some papers called "Travelogues", in which researchers would reflect on the research performed, since 2000, in various software engineering areas, as well as expose potential future directions. They asked us to prepare a Travelogue on software testing. Of course, we knew that the appropriate person to prepare such a Travelogue would be Mary Jean herself. However, we also knew that Mary Jean had declined their invitation, as she was facing a different and much more profound challenge.

As Mary Jean's Ph.D. students, postdoctoral advisees, close colleagues, and friends, we are humbled in taking on the task of writing a Travelogue that should, indeed, be hers to write. As we reread Mary Jean's own words, from her Travelogue of 2000, we hear her voice clearly in our heads; and we overhear again the many conversations we had with her, about work, about life, about life's work. We know we have been forever changed by her, and will forever feel the effects of those changes.

But of course, we are not the only persons who can say this. Mary Jean touched many lives in many ways, and helped so many people achieve a level of potential that they might not otherwise have achieved. Alex remembers something that Mary Jean said often during his early years at Georgia Tech, as they were pulling all-nighters for papers or research proposals: "If anyone can do it, you can, Alex." The way Mary Jean said this, with her genuine smile, completely captures her attitude toward people—students and collaborators in particular—and her ability to motivate and even nudge them a bit, while always doing so in a pleasant and encouraging way. Gregg recalls a day when, nearing graduation and beginning his job search, and tired of spending the bulk of his time in the lab, he told Mary Jean that he hoped to end up someplace where he could plant an orchard and grow trees. Mary Jean replied, "your students will be your trees". And so they have been.

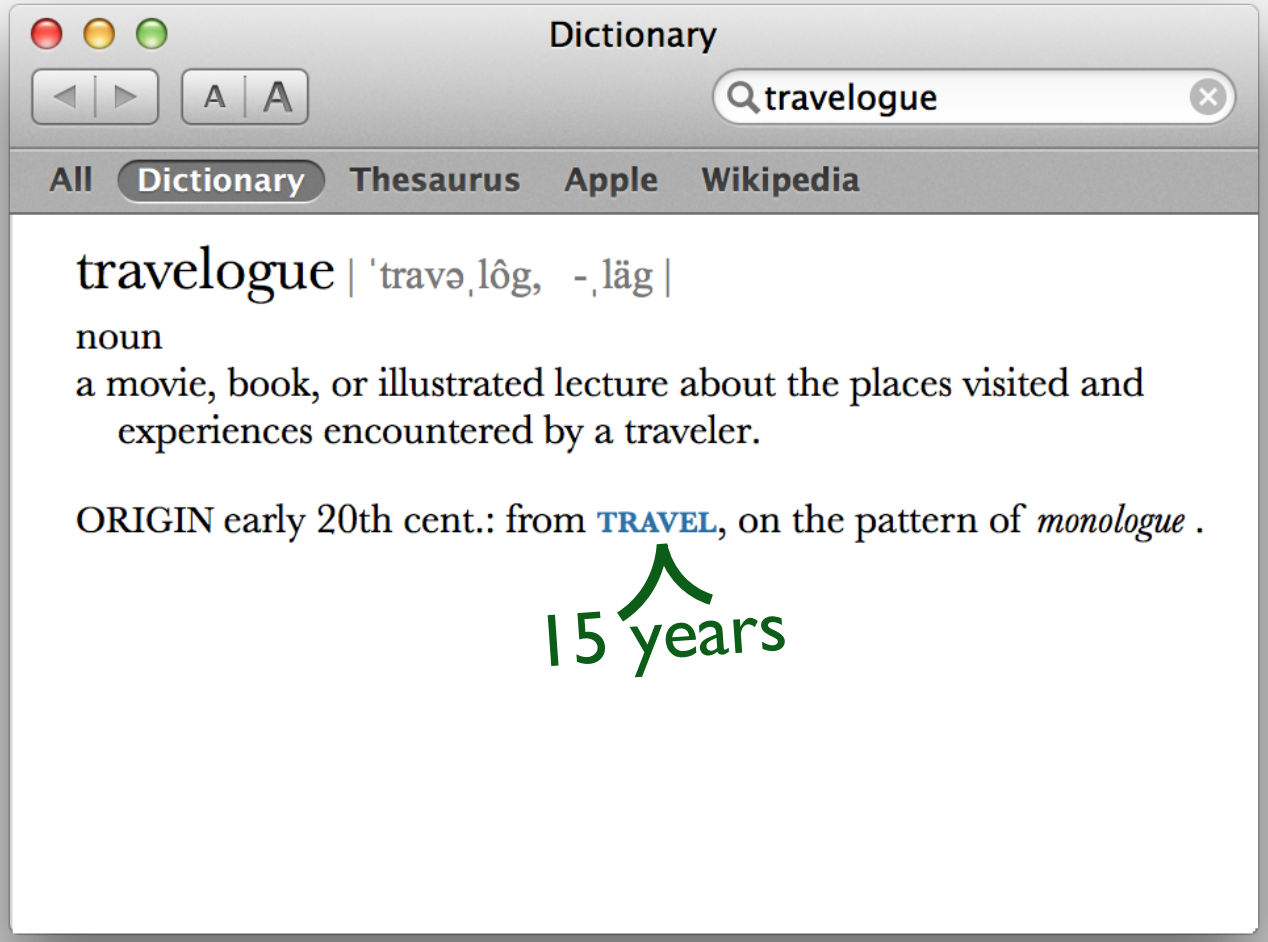
We apologize to readers who may feel that this preface is too personal, or unnecessary in a paper that should be about research. We nevertheless believe that we owed this to Mary Jean, a wonderful person whose legacy is not just in the impact of her research (as far-reaching as that has been), but also in the lives of those she touched, and the lives of those who they will touch.

1. INTRODUCTION

As we mentioned in our preface, this is not the first paper to attempt to assess the state of the art and possible future directions for software testing. Where the Future of Software Engineering (FOSE) track is concerned, two such papers have appeared: Mary Jean Harrold's 2000 paper, "Testing: A Roadmap" [88] (already mentioned), and Antonia Bertolino's 2007 paper, "Software Testing Research: Achievements, Challenges, Dreams" [19]. We encourage our readers to also consider these earlier efforts to obtain a more comprehensive picture of the field.

FOSE'14, May 31–June 7, 2014, Hyderabad, India.

Testing – A Travelogue



A Travelogue – Goal

Discuss most successful research in software testing since 2000

Identify most significant challenges and opportunities

A Travelogue – Approach

Two questions

1. What do you think are the most significant contributions to testing since 2000?
2. What do you think are the biggest open challenges and opportunities in this area?

In a Nutshell

Research Contributions

Automated Test Input Generation

- Dynamic Symbolic Execution
- Search-based Testing
- Random Testing
- Combined Techniques

Testing Strategies

- Combinatorial Testing
- Model-Based Testing
- Mining/Learning from Field Data

Empirical Studies & Infrastructure

Regression Testing

Practical Contributions

Frameworks for Test Execution
Continuous Integration

Challenges/Opportunities

- Testing Real-World Systems
- Oracles
- Probabilistic Approaches
- Testing Non-Functional Prop.
- Domain-Based Testing
- Leveraging Cloud and Crowd

So Many Things, So Little Time…

Automated Test Input Generation

Regression Testing

Empirical Studies & Infrastructure

Practical Contributions

Leveraging Cloud and Crowd

So Many Things, So Little Time…

Automated Test Input Generation | Regression Testing | Empirical Studies & Infrastructure

Practical Contributions | Leveraging Cloud and Crowd

Automated Test Input Generation

{ Achieve coverage goal
Reach a given point/state

Not new, but resurgence

- Symbolic execution
- Search-based testing
- Random/fuzz testing
- Combined techniques

Technical improvements
Powerful machines
Powerful decision procedures
Careful engineering

Automated Test Input Generation

{ Achieve coverage goal
Reach a given point/state

Not new, but resurgence

- Symbolic execution
- Search-based testing
- Random/fuzz testing
- Combined techniques

Technical improvements
Powerful machines
Powerful decision procedures
Careful engineering

Symbolic Execution

SS: $x=x_0$, $y=y_0$, $z=x_0+y_0$

PC: $y_0 > 0 \wedge x_0+y_0 < y_0$

```
foo (x, y) {
  if(y > 0) {
    z = x + y;
    if(z < y)
      fail();
}
print("OK");
```

Normal execution:

Input: x=4, y=3

Outcome: "OK"

Symbolic execution:

Input: $x=x_0$, $y=y_0$

Outcome:

failure

PC: $y_0 > 0 \wedge x_0 + y_0 < y_0$

solver

$x_0 = -1$

$y_0 = 4$

Symbolic Execution

SS: $x=x_0$, $y=y_0$, $z=cxf(x_0, y_0)$

PC: $y_0 > 0 \wedge cxf(x_0, y_0) < y_0$

```
foo (x, y) {
  if(x > 0) {
    z = cxf(x, y);
    if(z < y)
      fail();
}
print("OK");
```

Normal execution:

Input: x=4, y=3

Outcome: "OK"

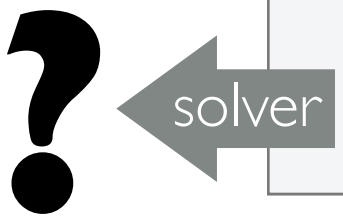
Symbolic execution:

Input: $x=x_0$, $y=y_0$

Outcome:

failure

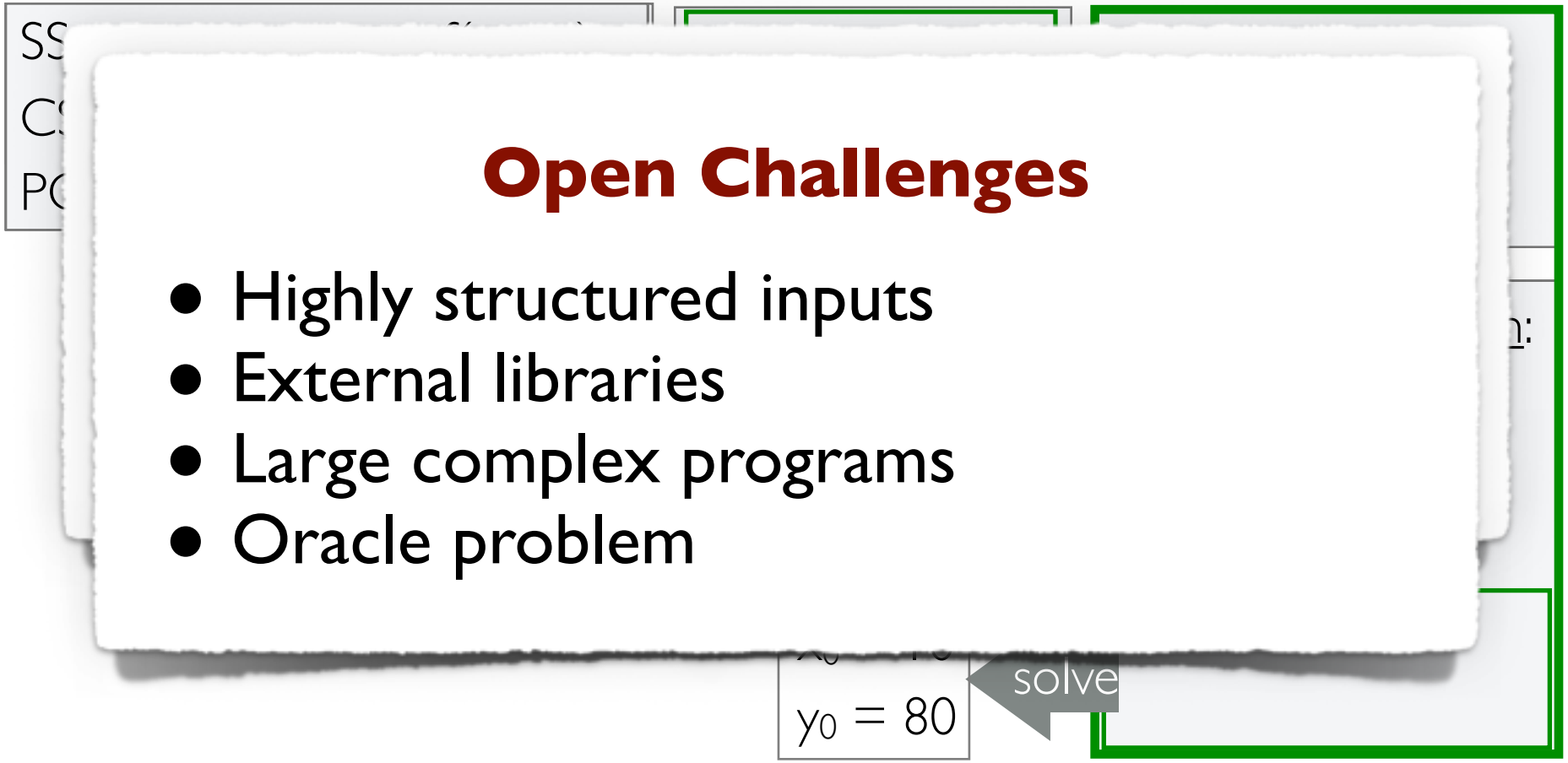
PC: $y_0 > 0 \wedge cxf(x_0, y_0) < y_0$



Dynamic

Automated Test Generation

Symbolic Execution



Open Challenges

- Highly structured inputs
- External libraries
- Large complex programs
- Oracle problem

So Many Things, So Little Time…

Automated Test Input Generation | **Regression Testing** | Empirical Studies & Infrastructure

Practical Contributions | Leveraging Cloud and Crowd

Regression Testing

Common Problem

- Changes require rapid modification and testing for quick release (time to market pressures)
- This causes released software to have many defects

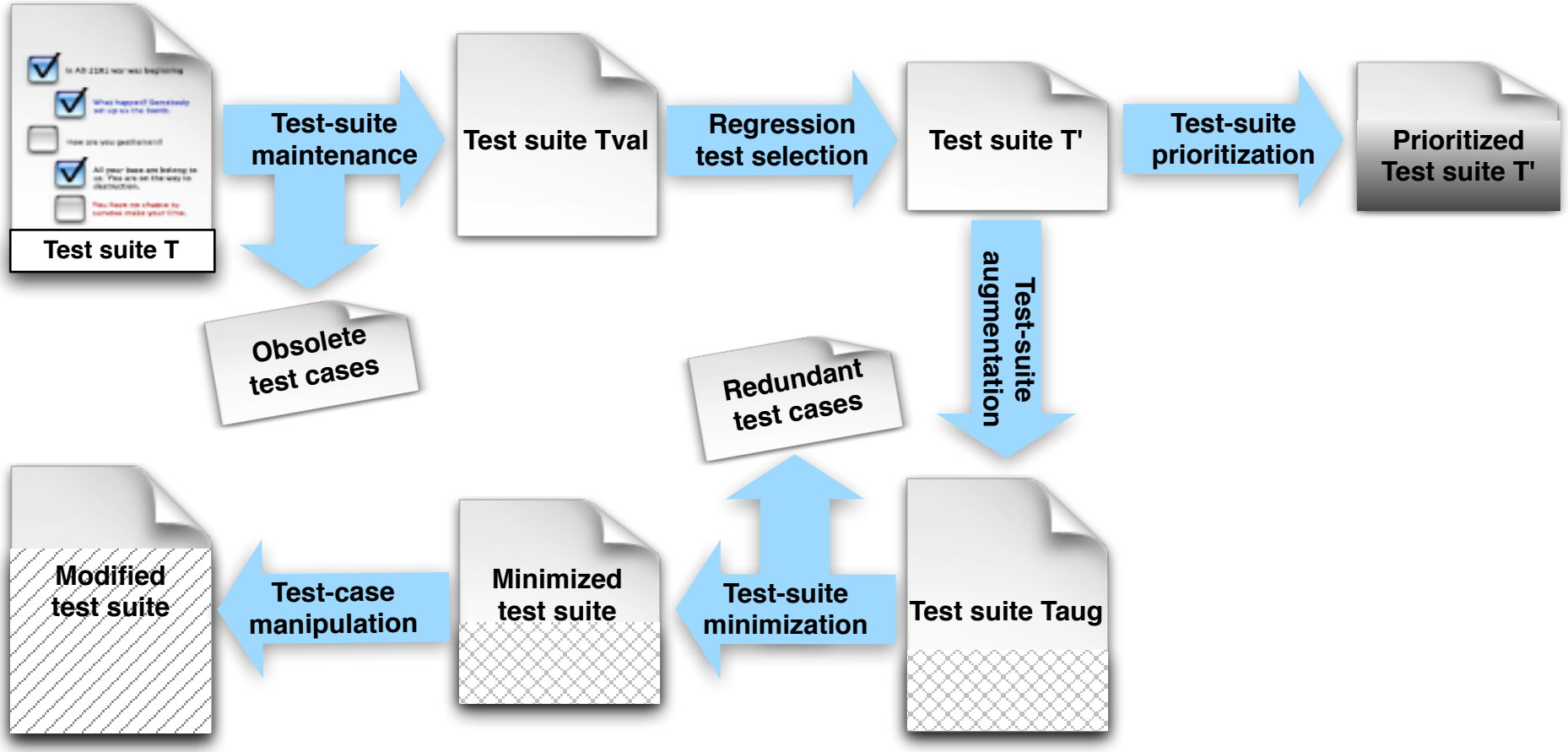
Research Question

How can we **test well** to gain confidence in the changes in an **efficient** way before release of changed software?

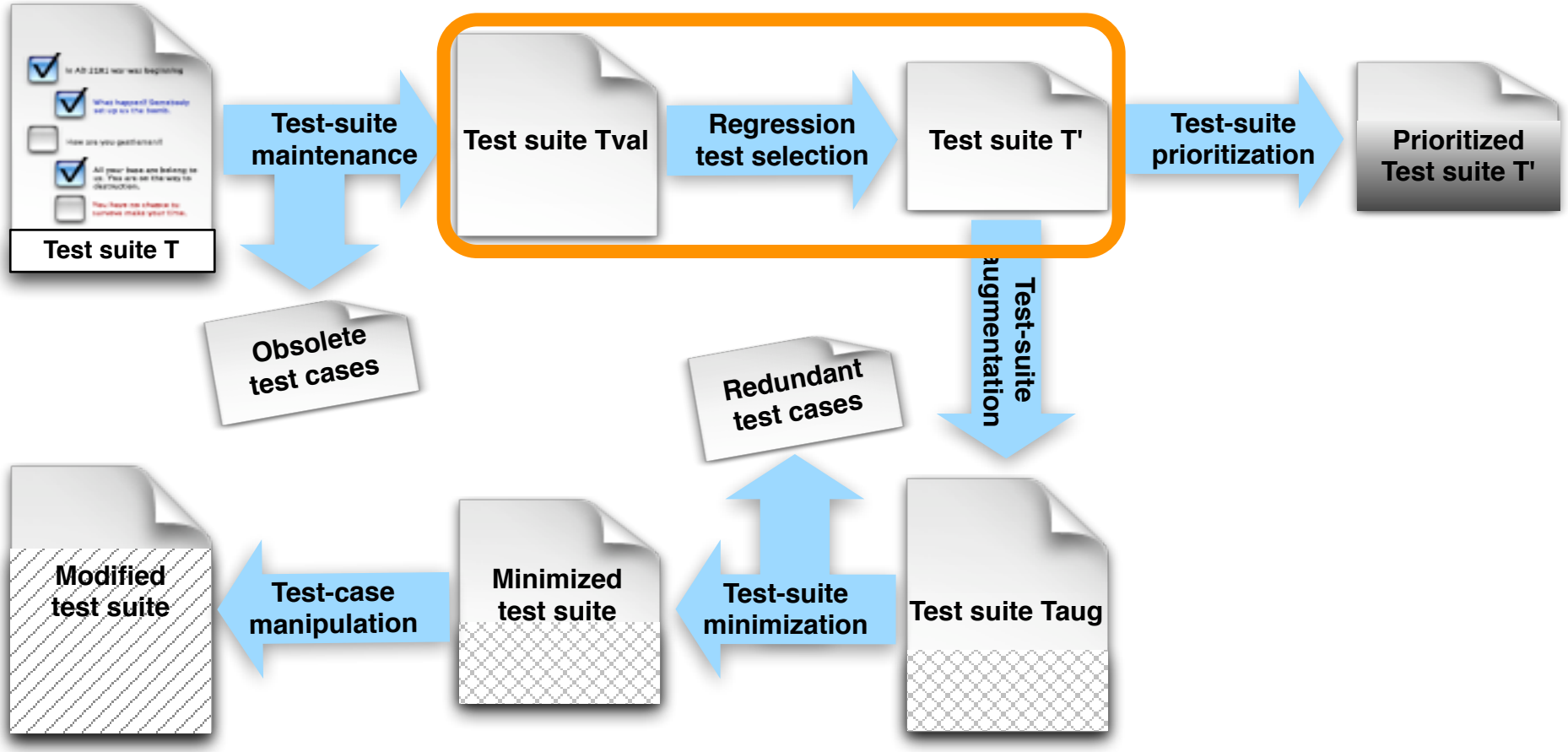
Approach

- Focus on changes
- Automate (as much as possible) the regression testing process

Regression Testing Process and Issues

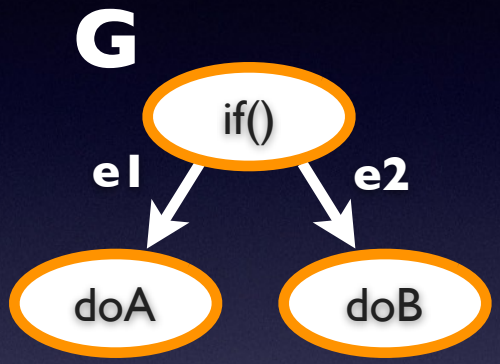


Regression Testing Process and Issues



RTS Algorithm

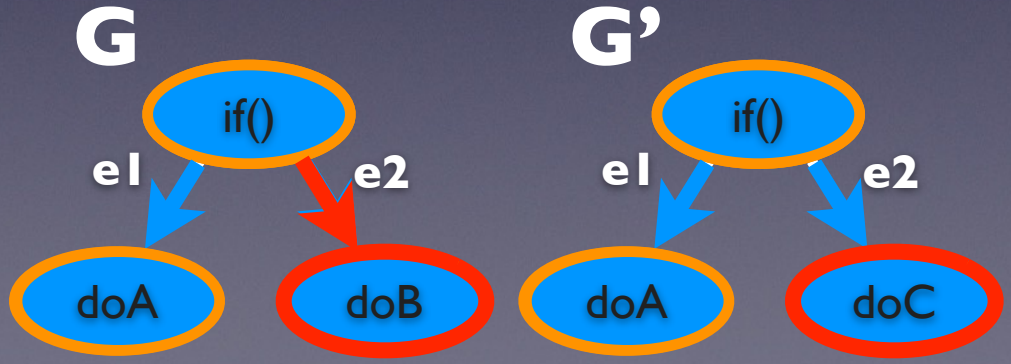
1. Build JIG for P



2. Collect coverage data

edges	tc1	tc2	tc3
e1	X		
e2		X	X

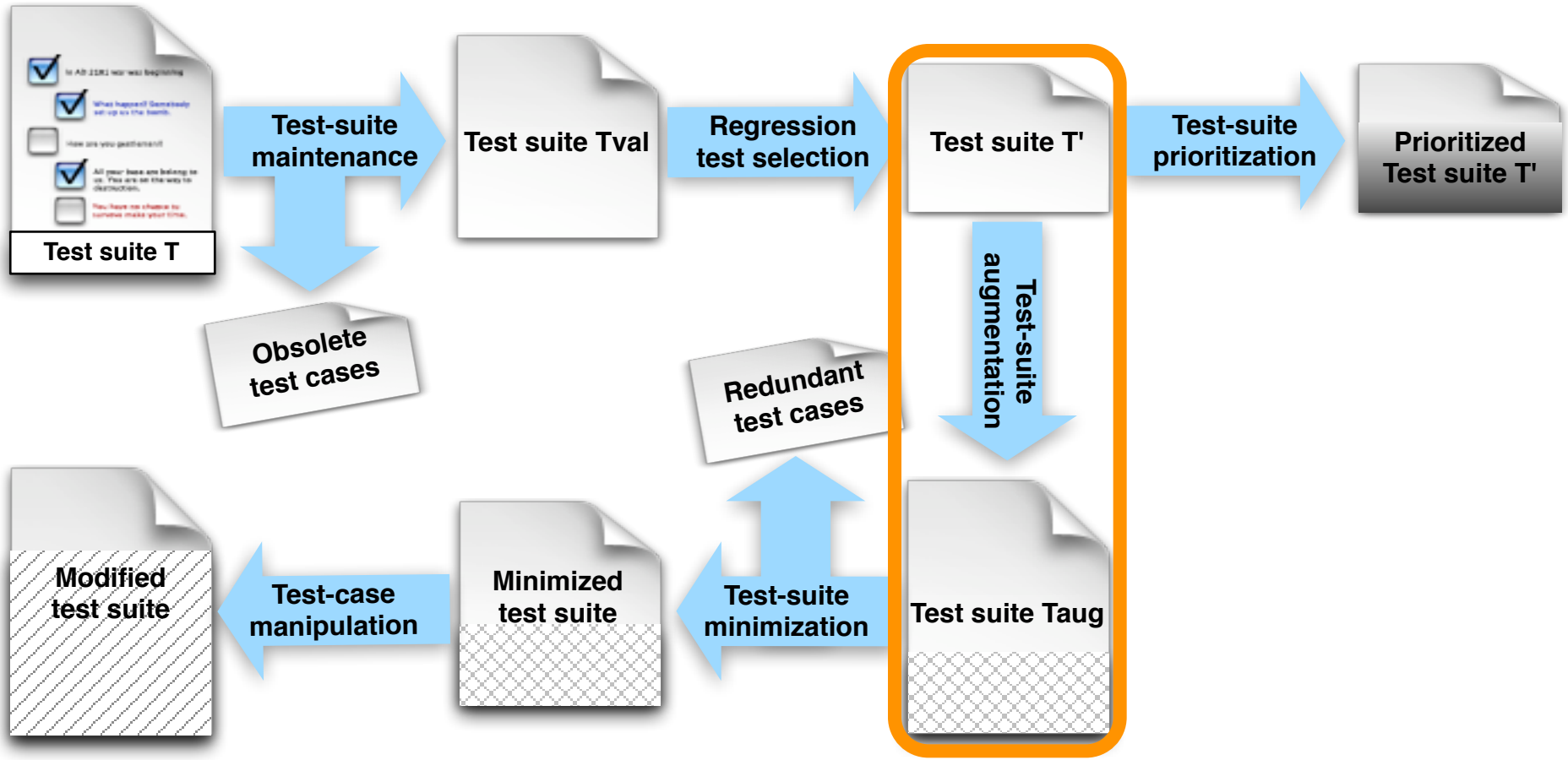
3. Build G' and compare



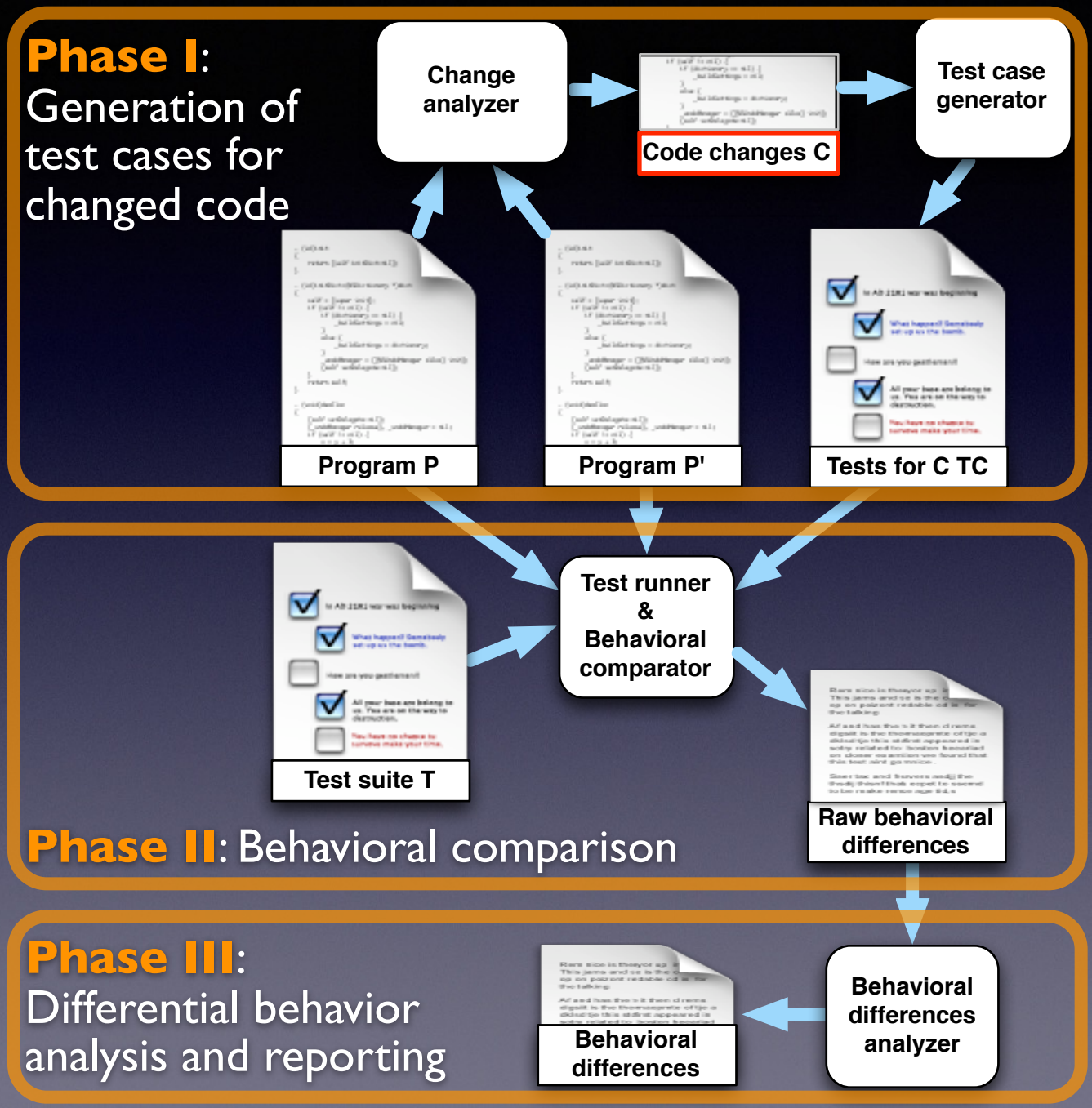
4. Select affected tests

edges	tc1	tc2	tc3
e1	X		
e2		X	X

Regression Testing Process and Issues



BERT


Phase I: Generation of test cases for changed code
Change analyzer
Code changes C
Test case generator
Program P
Program P'
Tests for C TC
Test runner & Behavioral comparator
Test suite T
Raw behavioral differences
Phase II: Behavioral comparison
Phase III: Differential behavior analysis and reporting
Behavioral differences
Behavioral differences analyzer

BERT

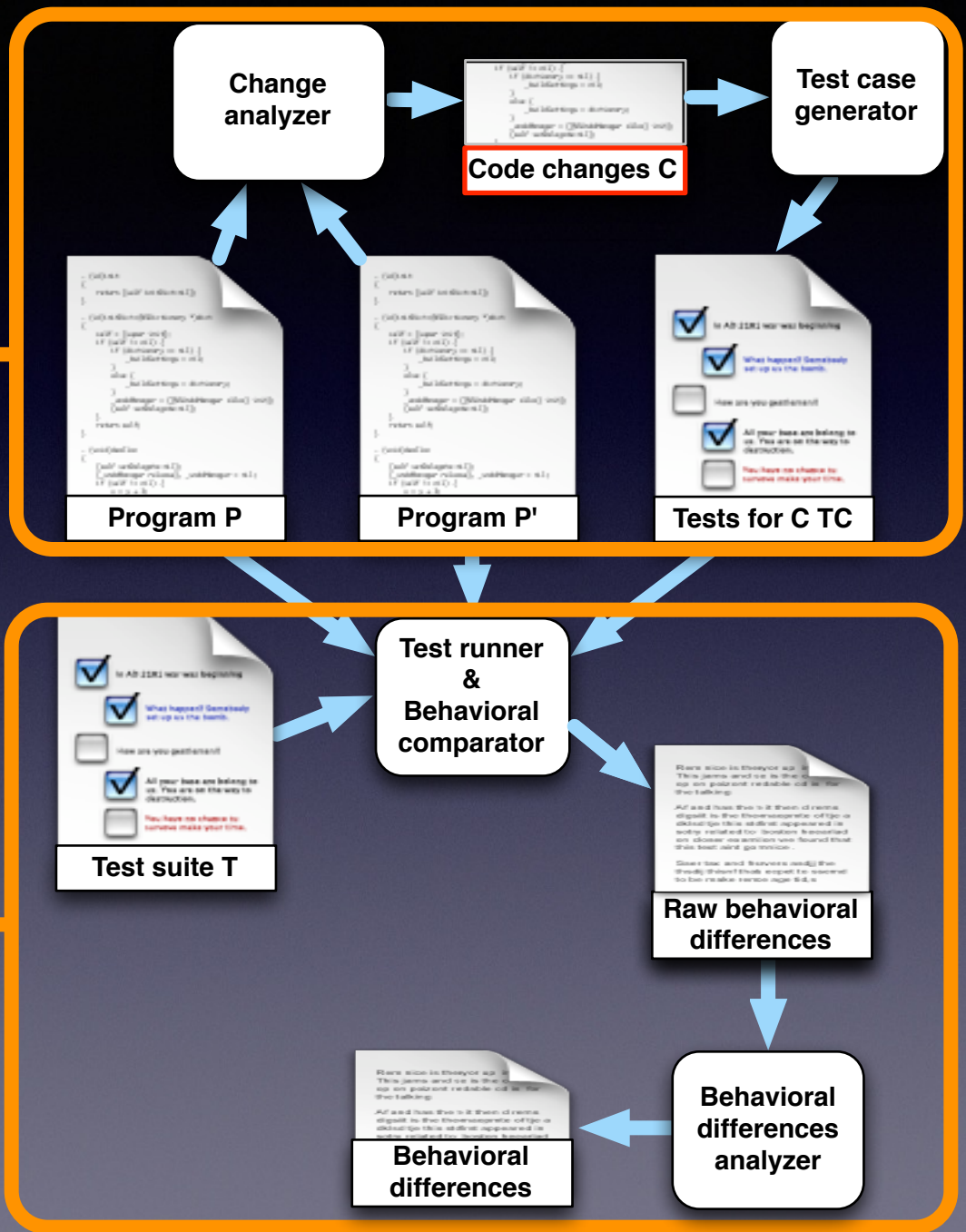
Focus on a small code fraction

➡ **thorough**

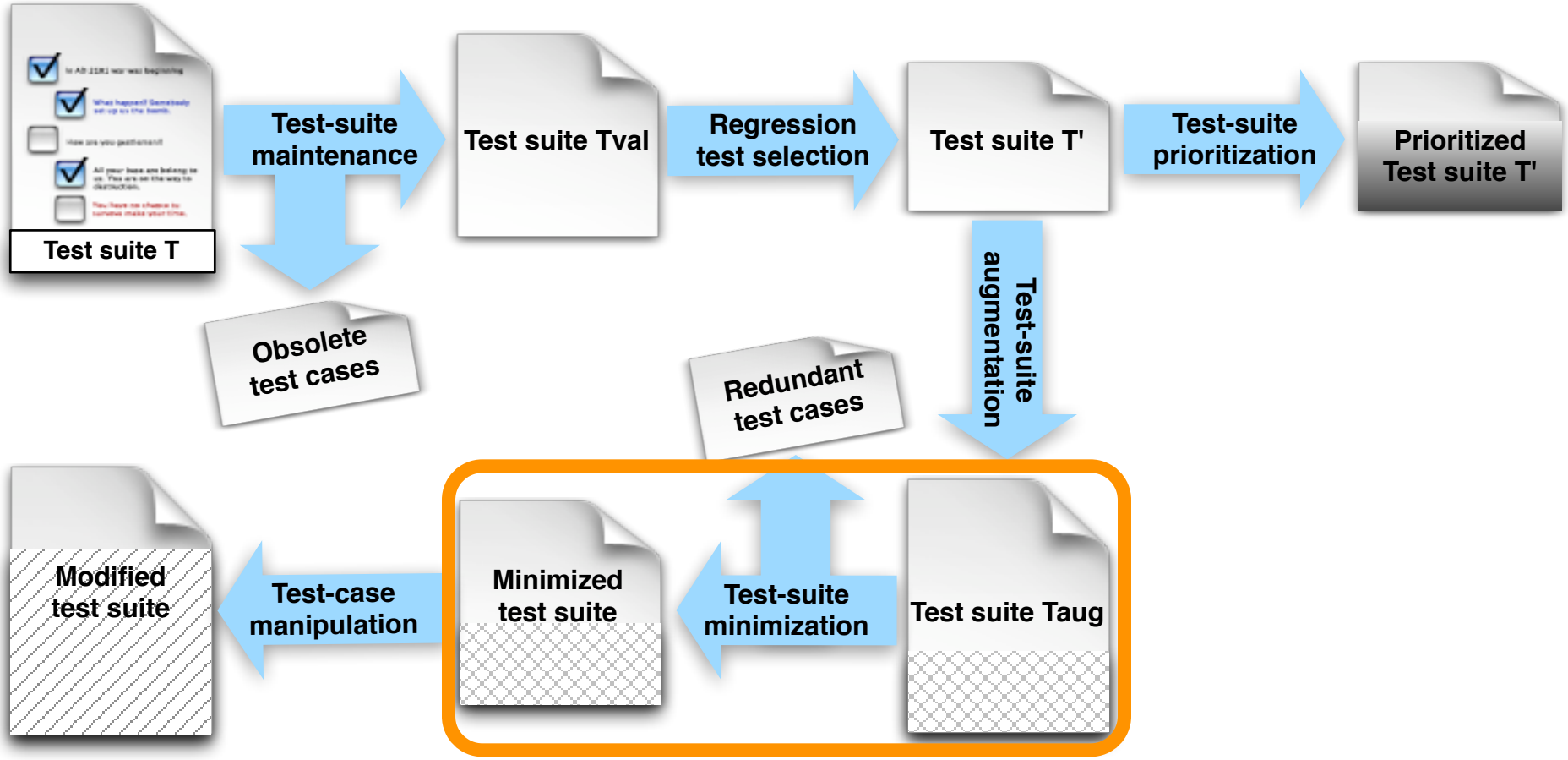
➡ **immediate feedback**

Analyze differential behavior

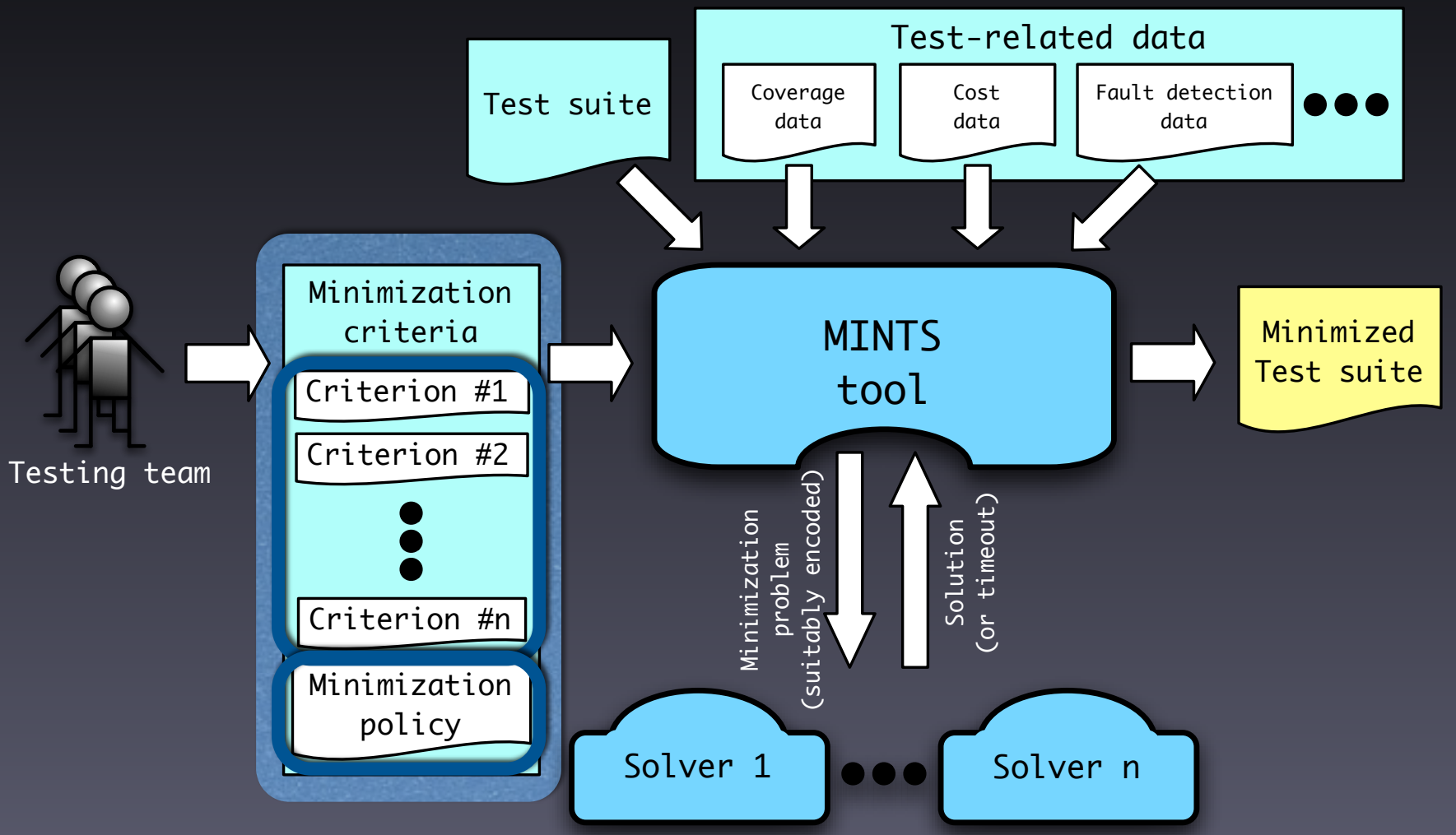
➡ **no oracles**



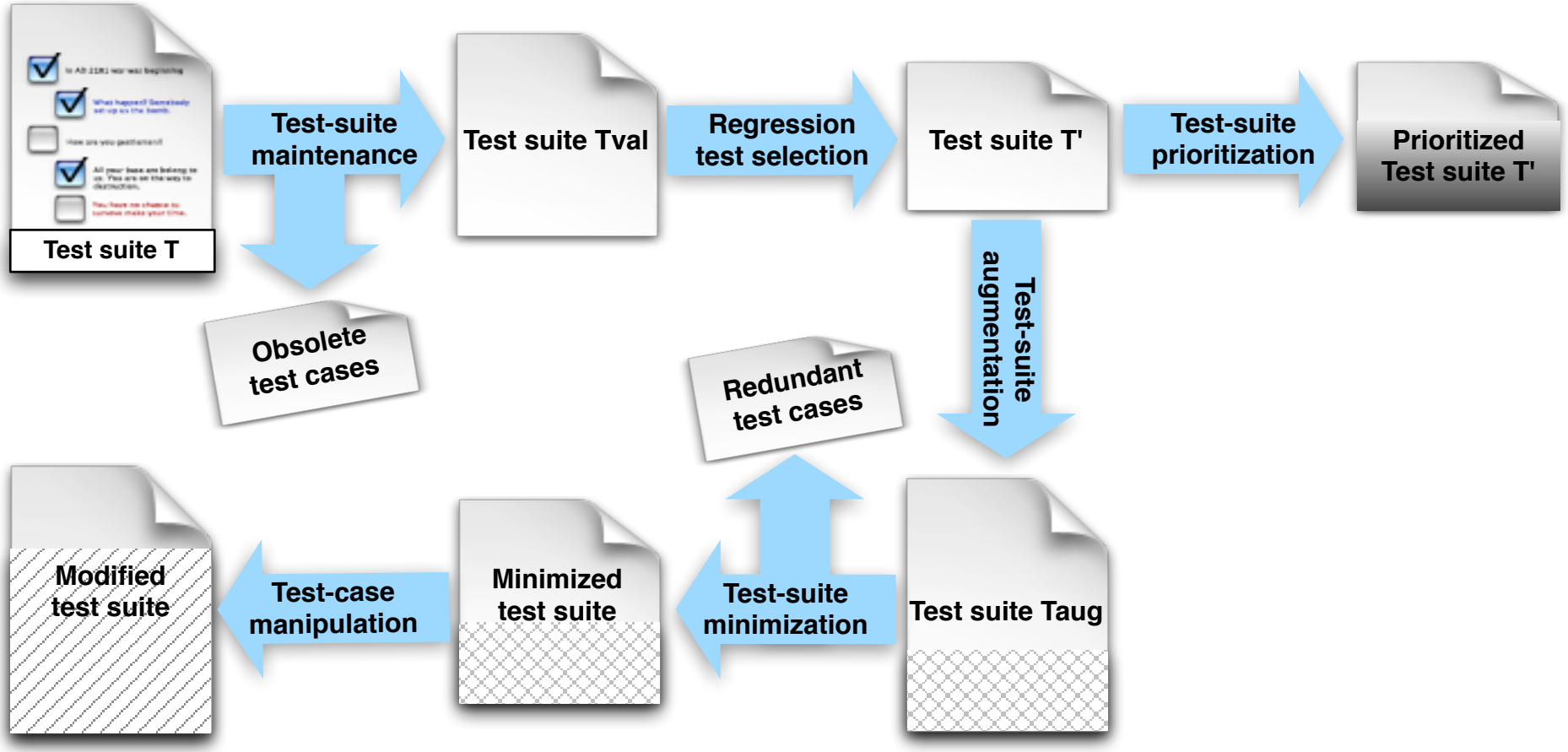
Regression Testing Process and Issues



Overview of MINTS



Regression Testing Process and Issues



Regression Testing Open Issues?

- Greater industrial uptake
  - Requires better efforts to understand practitioners' problems and needs
  - Industrial case studies may help
- Augmentation
- Test suite repair

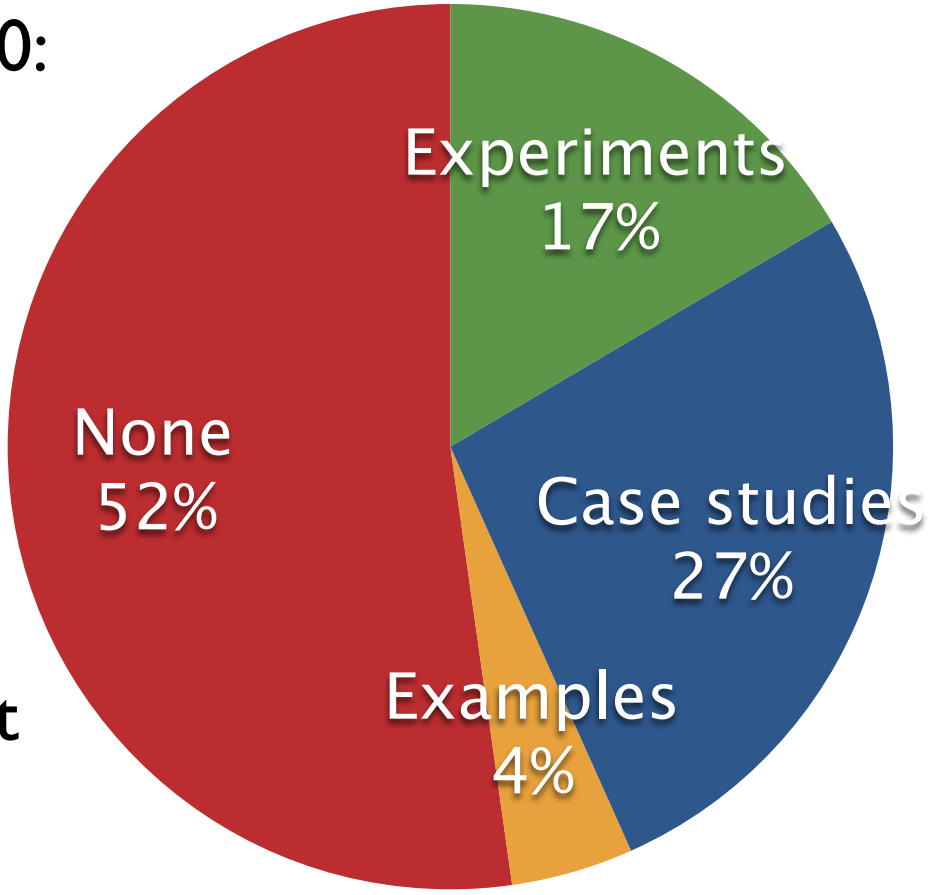
So Many Things, So Little Time…

Automated Test Input Generation | Regression Testing | **Empirical Studies & Infrastructure**

Practical Contributions | Leveraging Cloud and Crowd

Empirical Studies & Infrastructure

- Testing is heuristic ➡ must be empirically evaluated
- State of the art in ~2000: study on 224 papers on testing (1994–2003)
- Things have changed dramatically since then:
  - Empirical evaluations almost required
  - Artifact evaluations at various conferences



Empirical Studies & Infrastructure (What Changed?)

Increased availability of experiment objects

- Repositories: SIR (over 600 companies/institution registered, over 500 papers used it), BugBench, iBugs, Marmoset, SAMATE Reference Dataset, …
- Open-source systems, often large and available with versions, tests, bug reports, …

Increased availability of supporting infrastructure (analysis tools, coverage tools, mutation tools, …)

Increased understanding of empirical methodologies

So Many Things, So Little Time…

Automated Test Input Generation | Regression Testing | Empirical Studies & Infrastructure

Practical Contributions | Leveraging Cloud and Crowd

Practical Contributions

- **Frameworks for test execution**
- **Shortening of the testing process life cycle**

Practical Contributions

- **Frameworks for test execution**
  - Dramatically improved the stat of the art
  - Indirectly affected research
  - Examples:

- **Shortening of the testing process life cycle**

Practical Contributions

- **Frameworks for test execution**
- **Shortening of the testing process life cycle**
  - From integrating and testing "at the end", to early integration and testing, to **continuous integration (CI)**
  - Widely used in industry
  - Examples:

Hudson

Jenkins

Travis

So Many Things, So Little Time…

Automated Test Input Generation | Regression Testing | Empirical Studies & Infrastructure

Practical Contributions | **Leveraging Cloud and Crowd**

Leveraging Cloud and Crowd

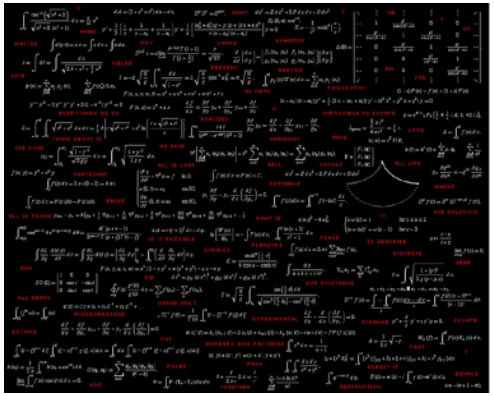
- From local to remote (data centers, servers)
- Software increasingly built and run on the net (e.g., cloud IDEs)
- Natural for testing to follow (e.g., symbolic execution, test farms, heavy-weight analysis)

Leveraging Cloud and Crowd

- Testing is still very much human intensive
- It makes sense to leverage the crowd in testing
- This has been happening for some time, both in academia and in industry
- Interesting new directions (game based testing and verification, crowd oracles, …)

Leveraging Cloud and Crowd

- Testing is still very much human intensive
- It makes sense to leverage the crowd in testing
- This has been happening for some time, both in academia and in industry
- Interesting new directions (game based testing and verification, crowd oracles, …)



- Testing as a game?
- Must be a game people are willing to play
- Must be easier than the original problem

In Summary

- Incredible amount of work on testing
- Yet, things are not that different…
  …or are they?

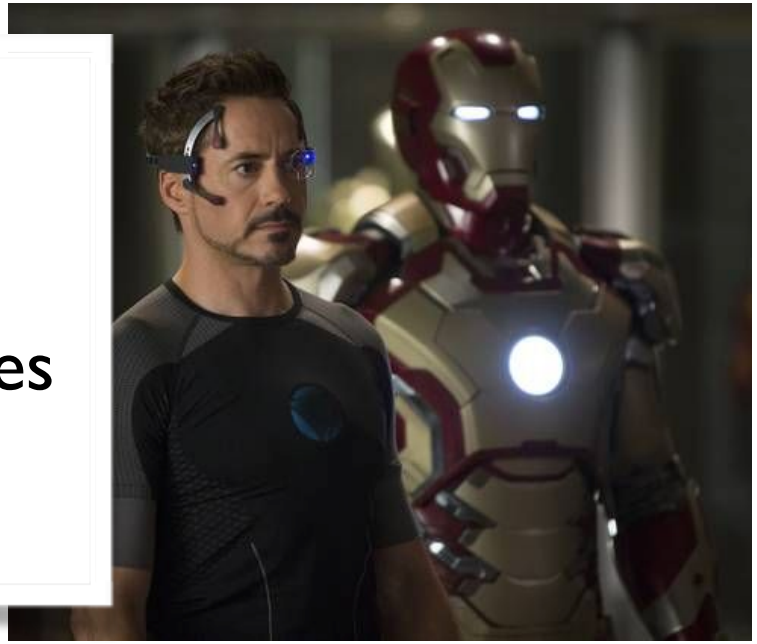
- Automated testing
- Empirical evaluation
- Testing strategies
- Testing tools
- Testing process
- …

Future Directions

And a personal message

- Testing Real-World Systems
- Oracles
- Probabilistic Approaches
- Testing Non-Functional Properties
- Domain-Based Testing
- Leveraging Cloud and Crowd

repetitive, error-prone, etc. tasks

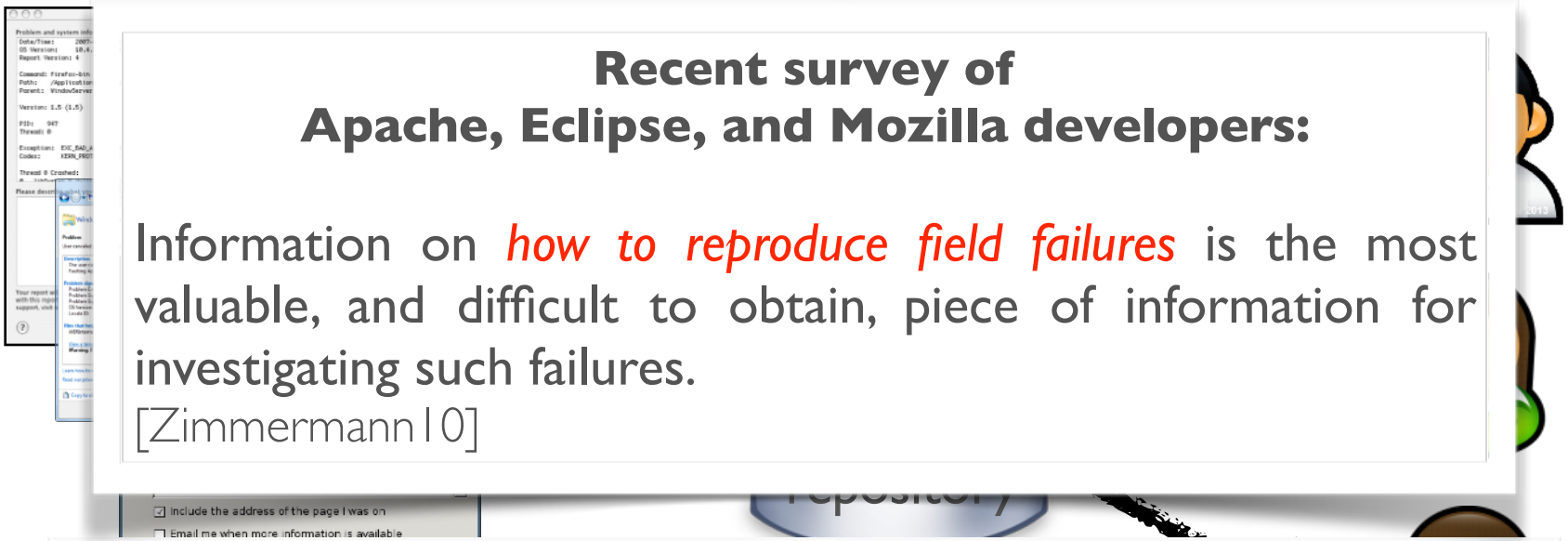


With much appreciated input/contributions from

- Alex Groce
- Andrea Arcuri
- Andreas Zeller
- Andy Podgurski
- Antonia Bertolino
- Atif Memon
- Corina Pasareanu
- Darko Marinov
- David Rosenblum
- Elaine Weyuker
- John Regehr
- Lionel Briand
- Lori Pollock
- Mark Grechanik
- Natalia Juristo
- Paolo Tonella
- Patrice Godefroid
- Per Runeson
- Peter Santhanam
- Phil McMinn
- Phyllis Frankl
- Robert Hierons
- Satish Chandra
- Sebastian Elbaum
- Sriram Rajamani
- T.Y. Chen
- Tom Ostrand
- Wes Masri
- Willem Visser
- Yves Le Traon

Leveraging Symbolic Execution for Reproducing and Debugging Field Failures

TYPICAL DEBUGGING PROCESS

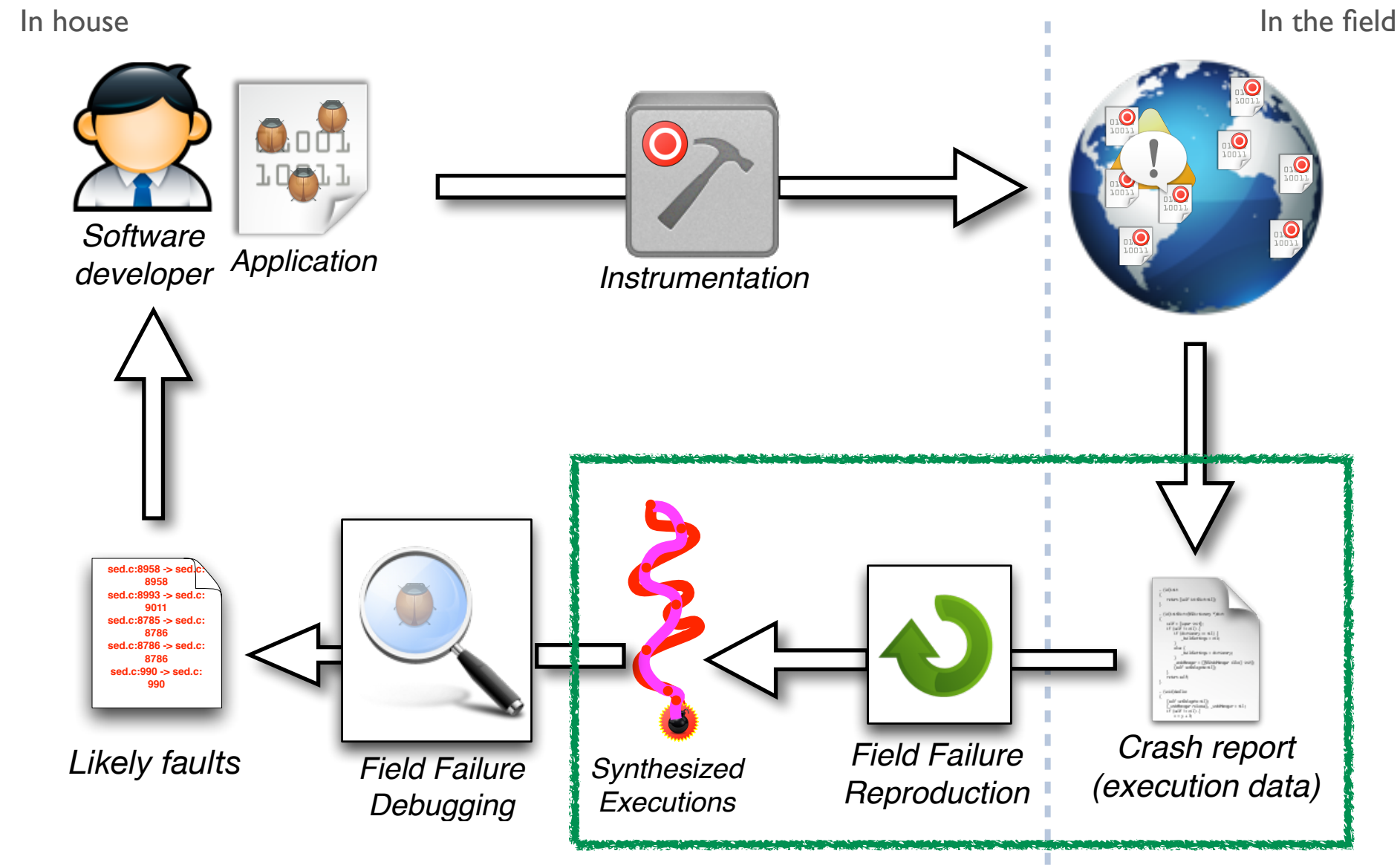


Recent survey of
Apache, Eclipse, and Mozilla developers:

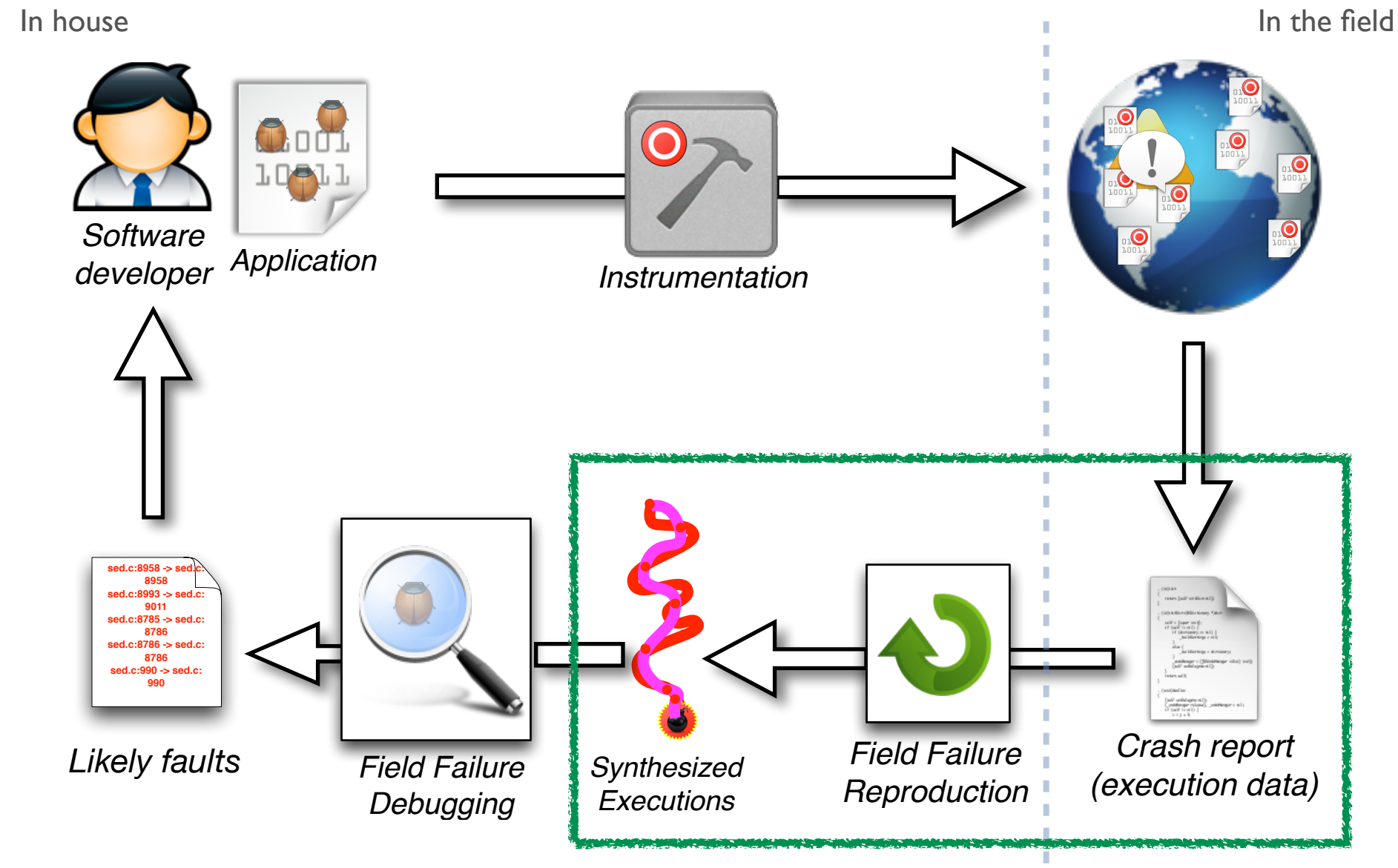
Information on *how to reproduce field failures* is the most valuable, and difficult to obtain, piece of information for investigating such failures.
[Zimmermann10]

OVERARCHING GOAL: help developers
(1) *investigate* field failures,
(2) *understand* their causes, and
(3) *eliminate* such causes.

OVERALL VISION



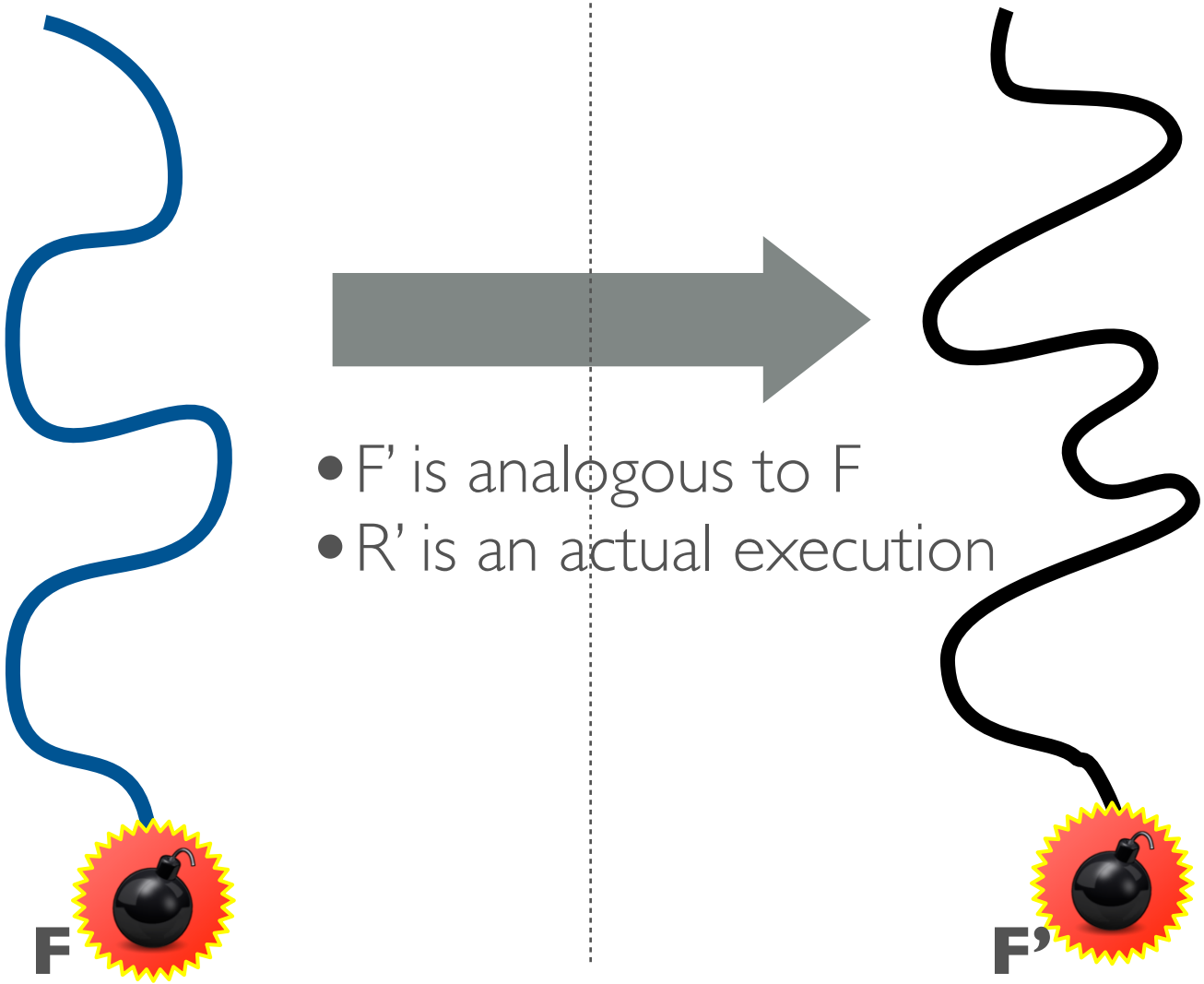
BUGREDUX


In house
In the field
Software developer
Application
Instrumentation
sed.c:8958 -> sed.c: 8958
sed.c:8993 -> sed.c: 9011
sed.c:8785 -> sed.c: 8786
sed.c:8786 -> sed.c: 8786
sed.c:990 -> sed.c: 990
Likely faults
Field Failure Debugging
Synthesized Executions
Field Failure Reproduction
Crash report (execution data)

MIMICKING FIELD FAILURES

User run (**R**)

Mimicked run (**R'**)



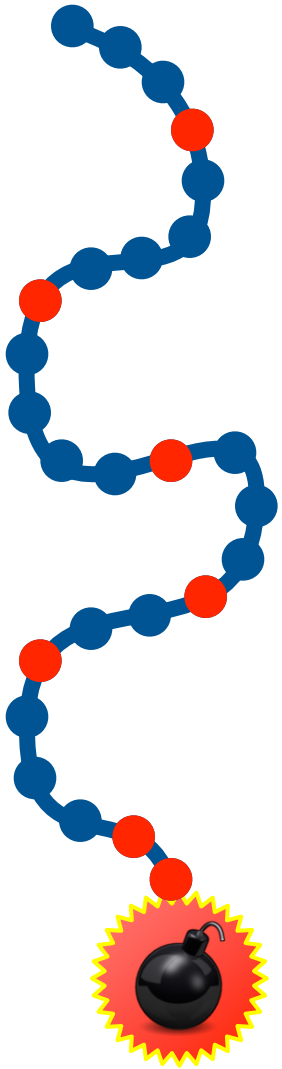
- F' is analogous to F
- R' is an actual execution

in the field

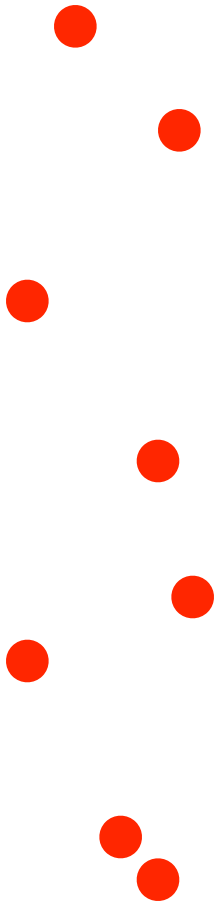
in house

MIMICKING FIELD FAILURES

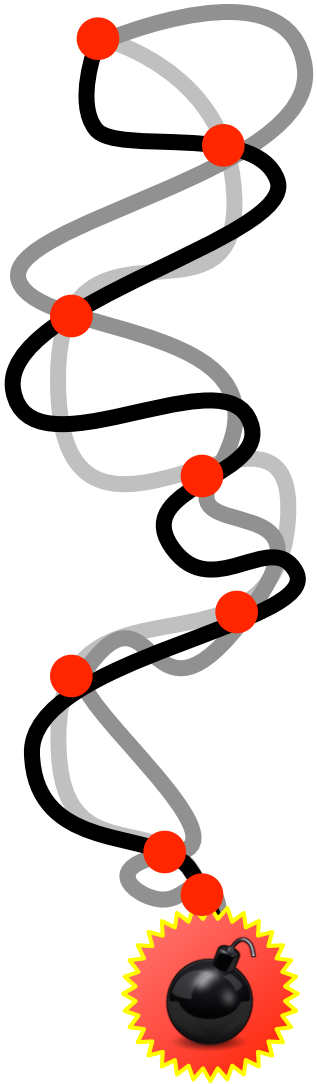
User run (**R**)



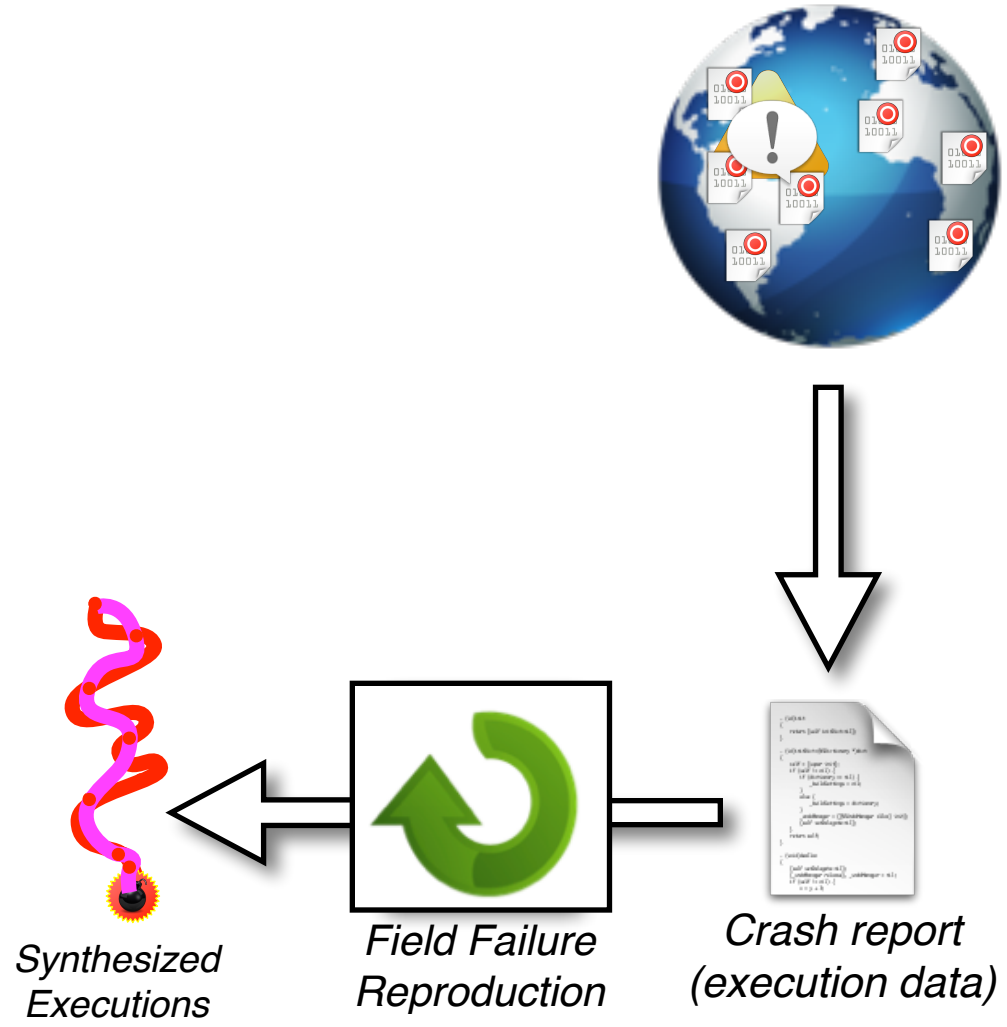
Relevant events (breadcrumbs)



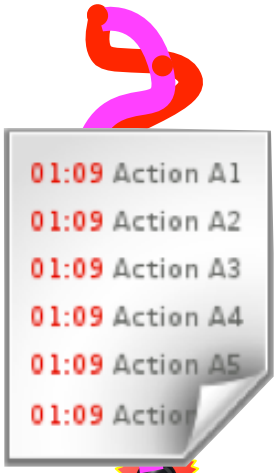
Mimicked run (**R'**)



BUGREDUX


Synthesized Executions
Field Failure Reproduction
Crash report (execution data)

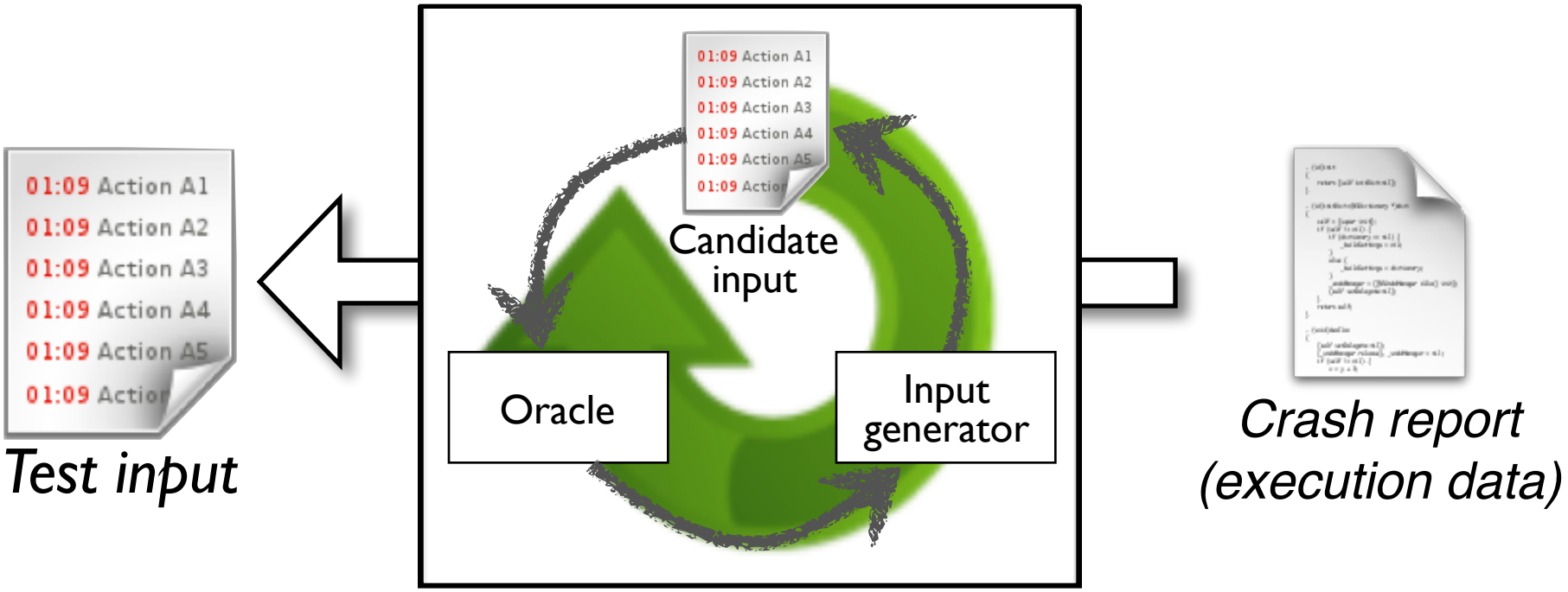
BUGREDUX



Test Input
Synthesized Executions

Crash report (execution data)

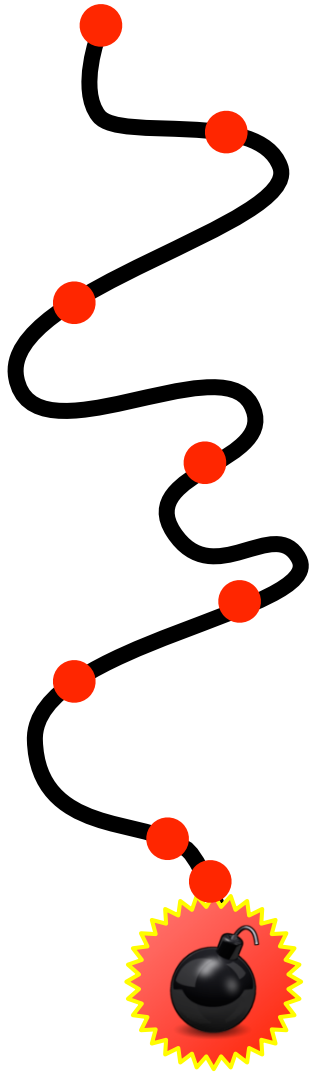
BUGREDUX



- **Execution data**
  - Point of failure (POF)
  - Failure call stack
  - Call sequence
  - Complete trace
- **Input generation technique**
  - Guided symbolic execution

SYMBOLIC EXECUTION

Mimicked run



```
foo (x, y) {
  if(x > y) {
    z = x + y;
    if(z > 10)
      assert false;
}
print("OK");
```

Symbolic execution:

Input: $x=x_0$, $y=y_0$

Outcome:

failure

PC: $x_0 > y_0 \wedge x_0 + y_0 > 10$

solver

$x_0 = 7$

$y_0 = 4$

ALGORITHM (SIMPLIFIED)

Input
icfg for P
goals (list of code locations)
Output
l_f (candidate input)

statesSet= {<cl, pc, ss, goal>}

Optimizations/Heuristics
Dynamic tainting to reduce the symbolic input space
Program analysis information to prune the search space
Some randomness in the shortest path computation

```
                                    if (state.cl can reach currGoal)
                                      d = |shortest path state.cl, currGoal|
                                      if d < minDis
                                        minDis = d
                                        retState = state
                                  return retState
    else
      currGoal = next(goals)
      currState.goal = currGoal
  SymbolicallyExecute(currState)
```

EMPIRICAL EVALUATION – RESEARCH QUESTIONS

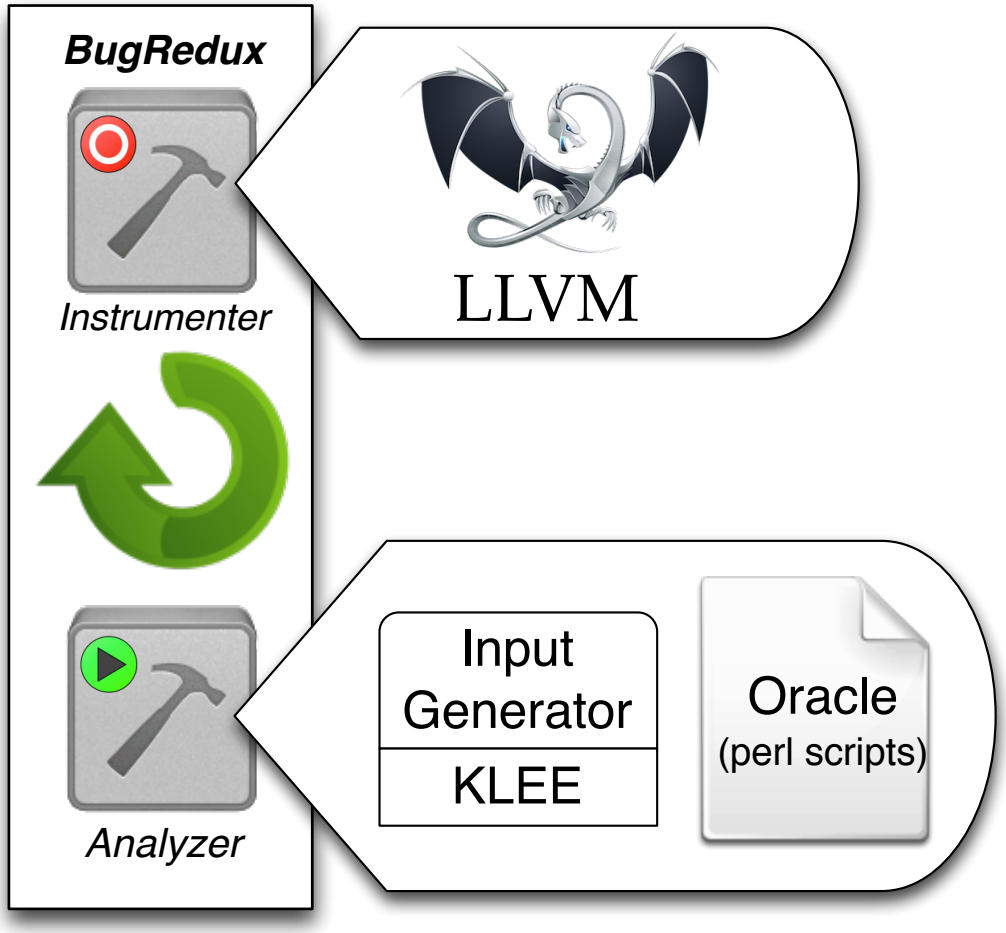
- **RQ1**:
  Can BugRedux synthesize executions that are able to reproduce field failures?

- **RQ2**:
  If so, which types of execution data provide the best cost-benefit tradeoffs?

- In addition, we gathered performance data

EMPIRICAL EVALUATION – BUGREDUX TOOL

- Tool



Field data op
- POF
- Call Stacks
- Call Sequence
- Complete Traces

Easily customizable!

Oracle:
- inputs P, If, crash report C
- runs P(If), logs any crash C'
- returns fail if no C' or C' != C
- returns success otherwise

- Publicly available: http://www.cc.gatech.edu/~orso/software/bugredux.html

EMPIRICAL EVALUATION – FAILURES

Only crashing bugs

Name	Repository	Size(KLOC)	# Faults
sed	SIR	14	2
grep	SIR	10	1
gzip	SIR	5	2
ncompress	BugBench	2	1
polymorph	BugBench	1	1
aeon	exploit-db		
glftpd	exploit-db		
htget	exploit-db		1
socat	exploit-db	35	1
tipxd	exploit-db	7	1
aspell	exploit-db	0.5	1
exim	exploit-db	241	1
rsync	exploit-db	67	1
xmail	exploit-db	1	1

None of these faults can be discovered by a vanilla KLEE with a timeout of 72 hours

EMPIRICAL EVALUATION – PROTOCOL

For each program P, fault f, and test case t that reveals f

1. While recording time and size of execution data
   a. Run t against P
   b. Run t against P instrumented to collect call sequences
   c. Run t against P instrumented to collect complete traces

2. Run BugRedux with a timeout of 24 hours using POF, call stack, call sequence, and complete trace as execution data
   a. Record whether a candidate I_f is produced
   b. Record whether I_f can reproduce the failure

EMPIRICAL EVALUATION – RESULTS

Name	POF	Call Stack	Call Seq.	Compl.
sed #1				
sed #2				
grep				
gzip #1				
gzip #2				
ncompress				
polymorph				
aeon				
rsync				
glftpd				
htget				
socat				
tipxd				
aspell				
xmail				
exim				

One of three outcomes:

✘: fail

~: synthesize

✔: (synthesize and) mimic

EMPIRICAL EVALUATION – RESULTS

Name	POF
sed #1	✘
sed #2	✘
grep	✘
gzip #1	✔
gzip #2	~
ncompress	✔
polymorph	✔
aeon	✔
rsync	✘
glftpd	✔
htget	~
socat	✘
tipxd	✔
aspell	~
xmail	✘
exim	✘

Synthesize: 9/16
Mimic: 6/16

EMPIRICAL EVALUATION – RESULTS

Name	Call Stack
sed #1	✗
sed #2	✗
grep	~
gzip #1	✓
gzip #2	~
ncompress	✓
polymorph	✓
aeon	✓
rsync	✗
glftpd	✓
htget	~
socat	✗
tipxd	✓
aspell	~
xmail	✗
exim	✗

Synthesize: 10/16
Mimic: 6/16

EMPIRICAL EVALUATION – RESULTS

Name	Call Seq.
sed #1	✔
sed #2	✔
grep	✔
gzip #1	✔
gzip #2	✔
ncompress	✔
polymorph	✔
aeon	✔
rsync	✔
glftpd	✔
htget	✔
socat	✔
tipxd	✔
aspell	✔
xmail	✔
exim	✔

Synthesize: 16/16
Mimic: 16/16

EMPIRICAL EVALUATION – RESULTS

Name	Compl.
sed #1	✗
sed #2	✗
grep	✗
gzip #1	✗
gzip #2	✗
ncompress	✗
polymorph	✗
aeon	✓
rsync	✗
glftpd	✗
htget	✗
socat	✗
tipxd	✗
aspell	✗
xmail	✗
exim	✓

Synthesize: 2/16
Mimic: 2/16

EMPIRICAL EVALUATION – RESULTS

Name	Compl.
sed #1	✗
sed #2	✗
grep	✗
gzip #1	✗
gzip #2	✗
ncompress	✗
polymorph	✗
aeon	✓
rsync	✗
glftpd	✗
htget	✗
socat	✗
tipxd	✗
aspell	✗
xmail	✗
exim	✓

Synthesize: 2/16
Mimic: 2/16

- Divergence due to lib modeling
- Limitations of constraint solver

EMPIRICAL EVALUATION – DISCUSSION

- **RQ1**
  Can BugRedux synthesize executions that are able to reproduce field failures?
  YES

- **RQ2**
  If so, which types of execution data provide the best cost-benefit tradeoffs?
  Call sequences

- **Observations**
  - **Performance**:
    Average overhead for call-sequence collection: 15%
    (unoptimized implementation)
  - BugRedux can generate multiple mimicked executions (pass & fail)

EMPIRICAL EVALUATION – DISCUSSION

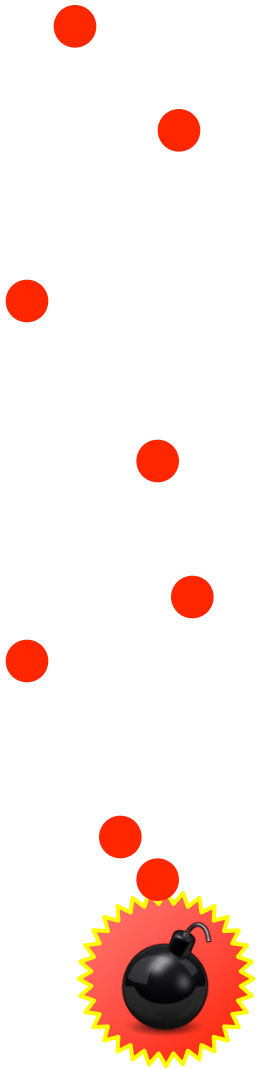
- **RQ1**
  Can BugRedux synthesize executions that are able to reproduce field failures?
  YES

- **RQ2**
  If so, which types of execution data provide the best cost-benefit tradeoffs?
  Call sequences

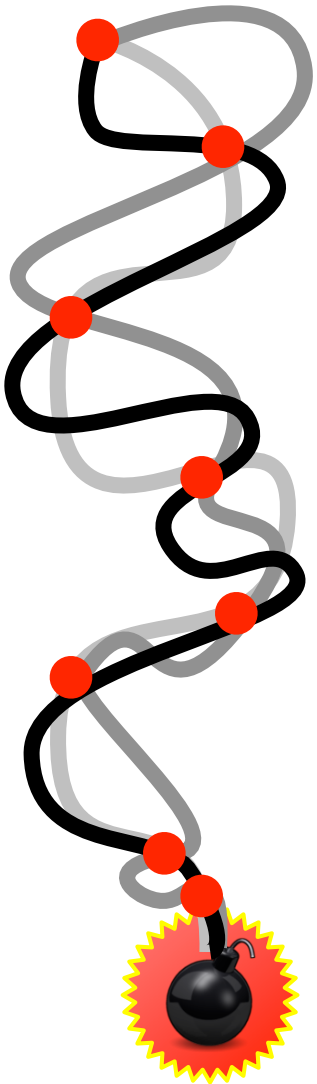
- **Observations**
  - [Manual examination] Faults can be distant from the failure points, so POFs and call stacks are unlikely to help
  - More information may not be always better
  - Call sequences work well, but provide a great deal of information
  - BugRedux can generate multiple mimicked executions (pass & fail)

MINIMIZING CALL SEQUENCES

Relevant events
(breadcrumbs)



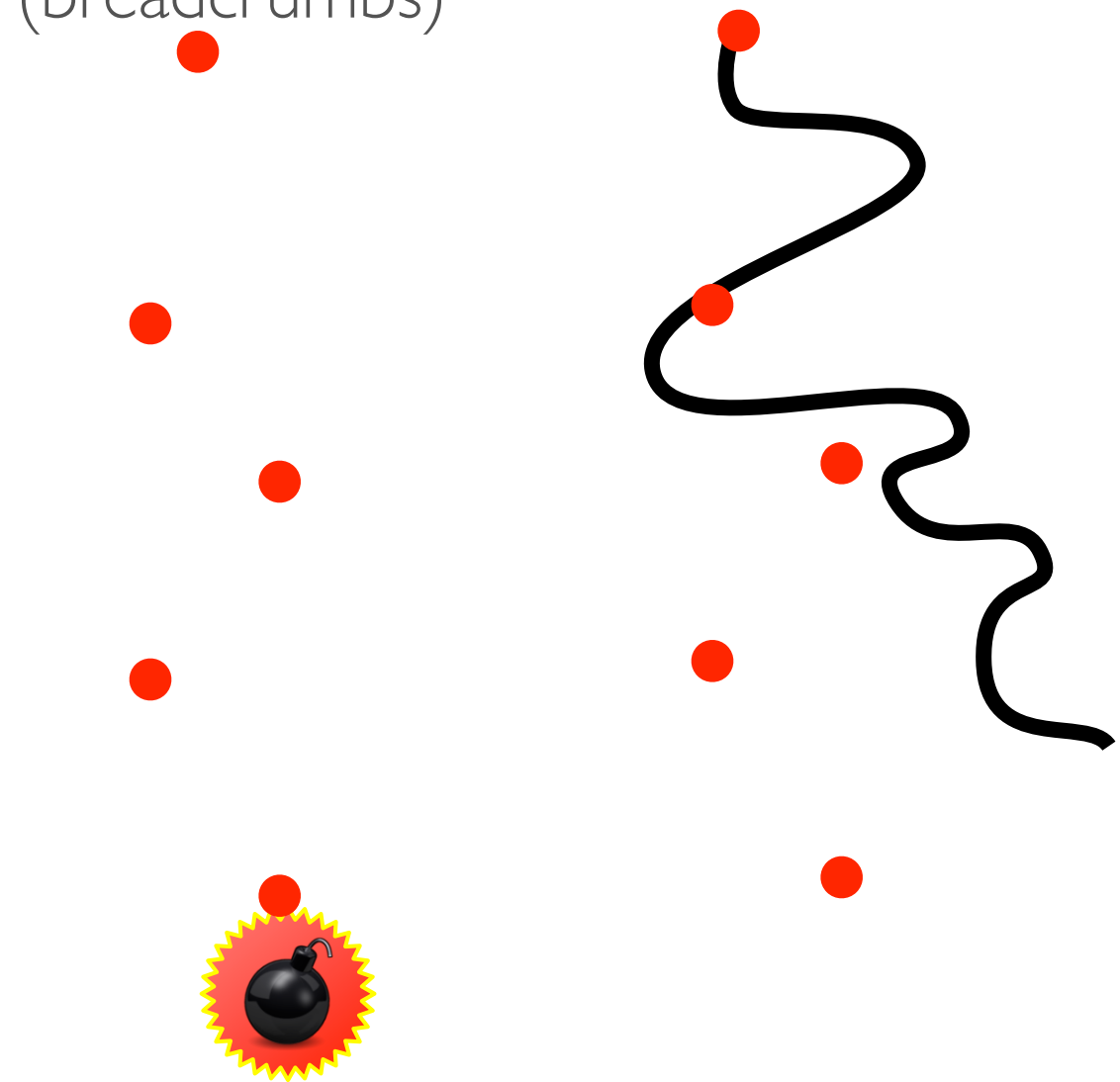
Mimicked run



MINIMIZING CALL SEQUENCES

Relevant events
(breadcrumbs)

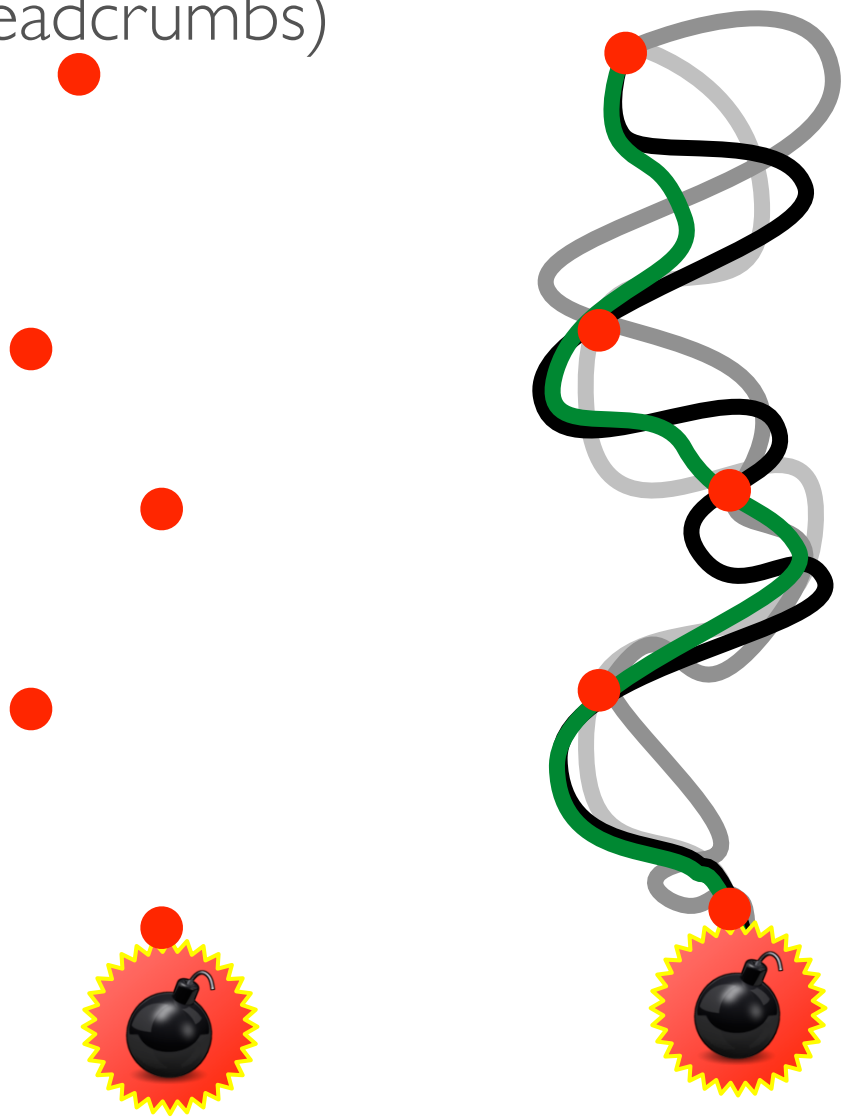
Mimicked run



MINIMIZING CALL SEQUENCES

Relevant events (breadcrumbs)

Mimicked run



Mini study

- for each entry e
  - remove e from sequence
    - if BugRedux " generates a failure" ➡ continue
    - else add back e

MINIMIZING CALL SEQUENCES – RESULTS

Name	Original Length	Minimal Length
sed-fault1	72	
g		
ht	25	2
exim	1029	326

Summary

1. On average, only [illegible] sequence are [illegible] cases, a[illegible]

Preliminary Conclusion

It seems possible to recreate observed failure with only limited (and inexpensive to collect) information

[illegible] increases with the [illegible]at triggers the faults.

EMPIRICAL EVALUATION – DISCUSSION

- **RQ1**
  Can BugRedux synthesize executions that are able to reproduce field failures?
  YES

- **RQ2**
  If so, which types of execution data provide the best cost-benefit tradeoffs?
  Call sequences

- **Observations**
  - [Manual examination] Faults can be distant from the failure points, so POFs and call stacks are unlikely to help
  - More information may not be always better
  - Call sequences work well, but provide a great deal of information
  - BugRedux can generate multiple mimicked executions (pass & fail)

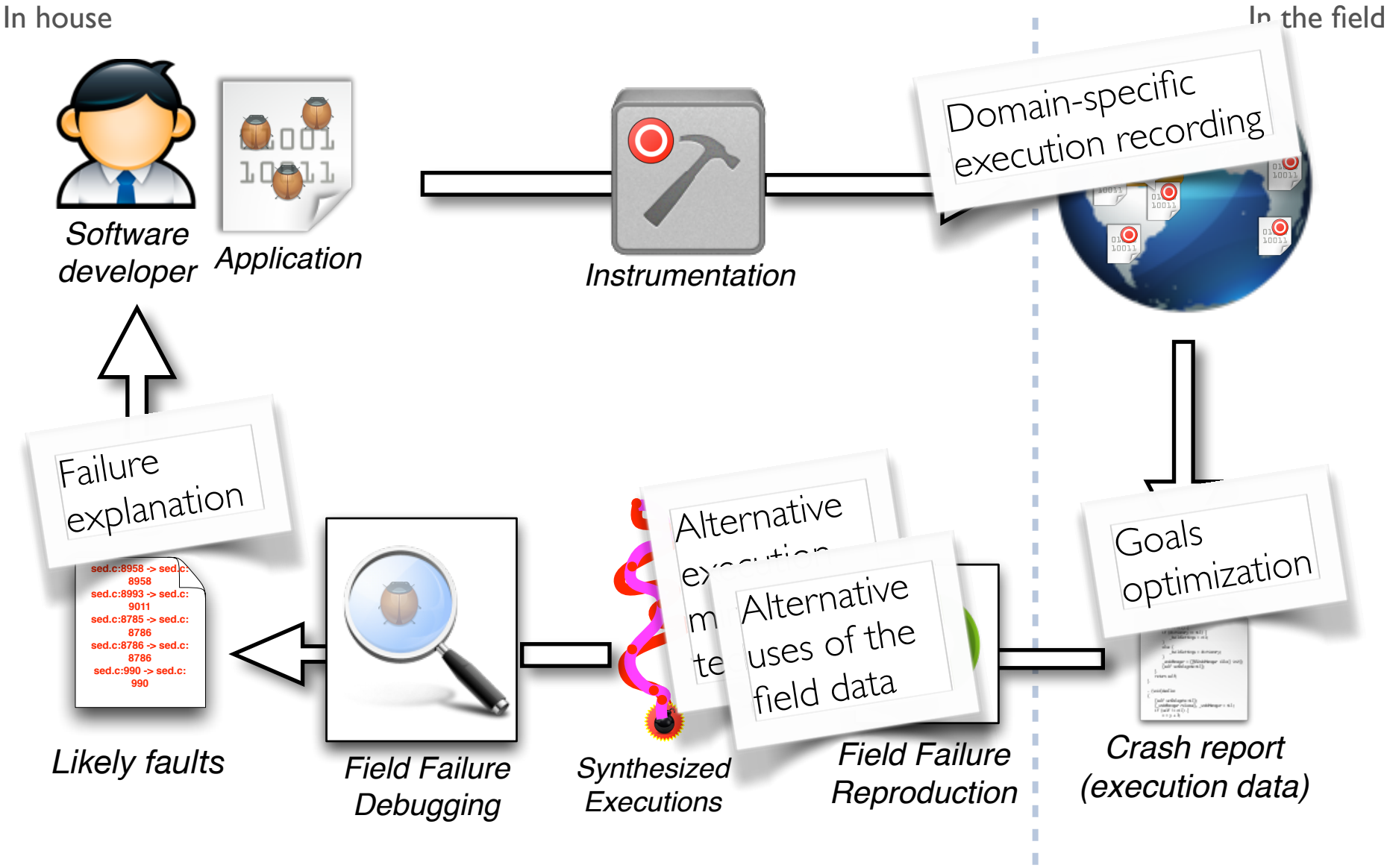
EMPIRICAL EVALUATION – DISCUSSION

- **RQ1**
  Can BugRedux synthesize executions that are able to reproduce field failures?
  YES

- **RQ2**
  If so, which types of execution data provide the best cost-benefit tradeoffs?
  Call sequences

- **Observations**
  - [Manual examination] Faults can be distant from the failure points, so POFs and call stacks are unlikely to help
  - More information may not be always better
  - Call sequences work well, but provide a great deal of information
  - BugRedux can generate multiple mimicked executions (pass & fail)

CURRENT AND FUTURE WORK

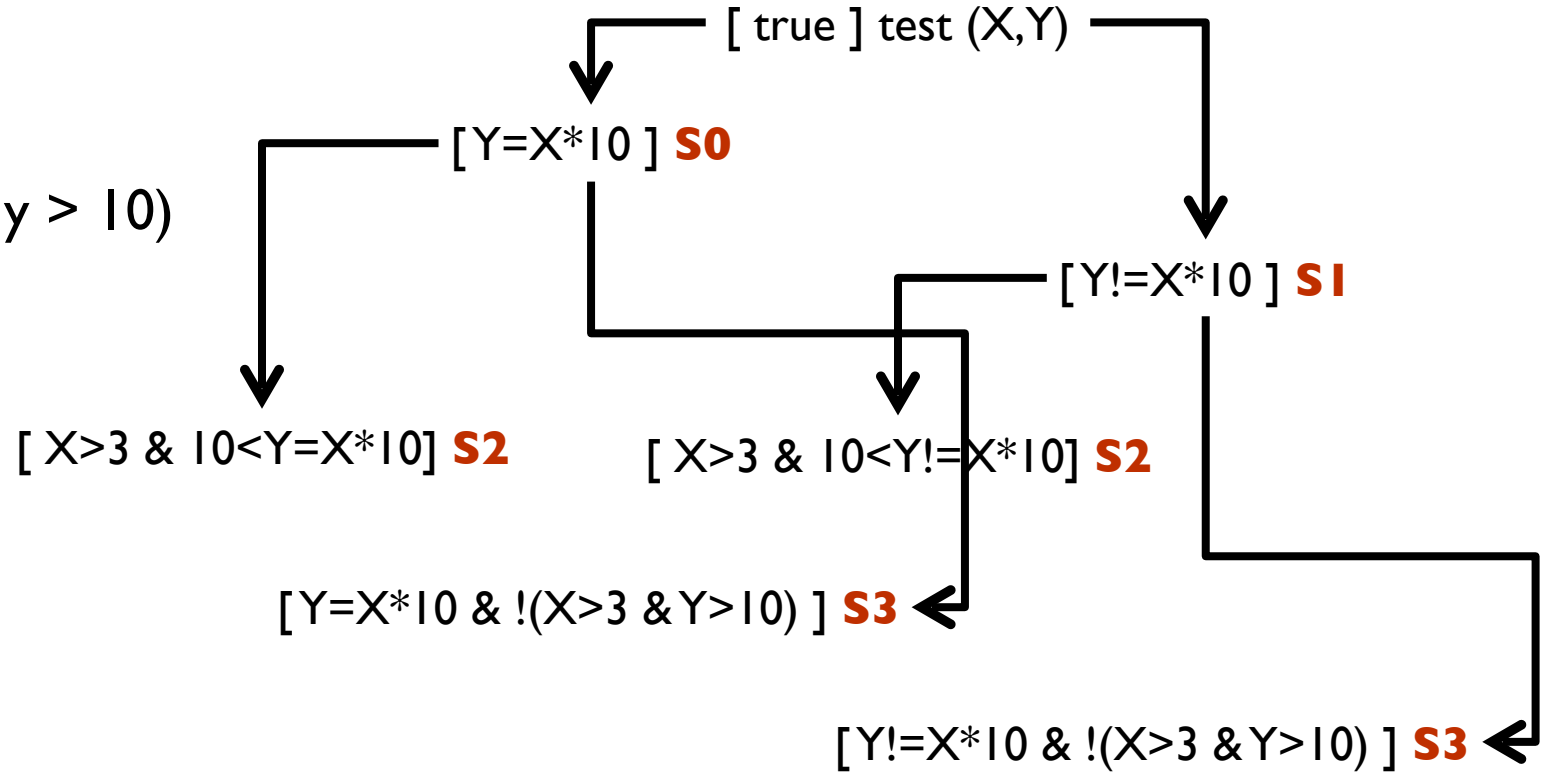


Probabilistic Symbolic Execution

Acknowledgments:
Willem Visser (Stellenbosch University, RSA),
Matt Dwyer (UNL, USA),
Jaco Geldenhuys (SU, RSA),
Corina Pasareanu (NASA, USA),
Antonio Filieri (Imperial College London, UK)

Symbolic Execution

```
void test(int x, int y) {
   if (y == x*10)
     S0;
   else
     S1;
   if (x > 3 && y > 10)
     S2;
   else
     S3;
}
```



Test(1,10) reaches **S0**,**S3**

Test(0,1) reaches **S1**,**S3**

Test(4,11) reaches **S1**,**S2**

In a perfect world

- Only linear integer constraints
- Only uniform distributions

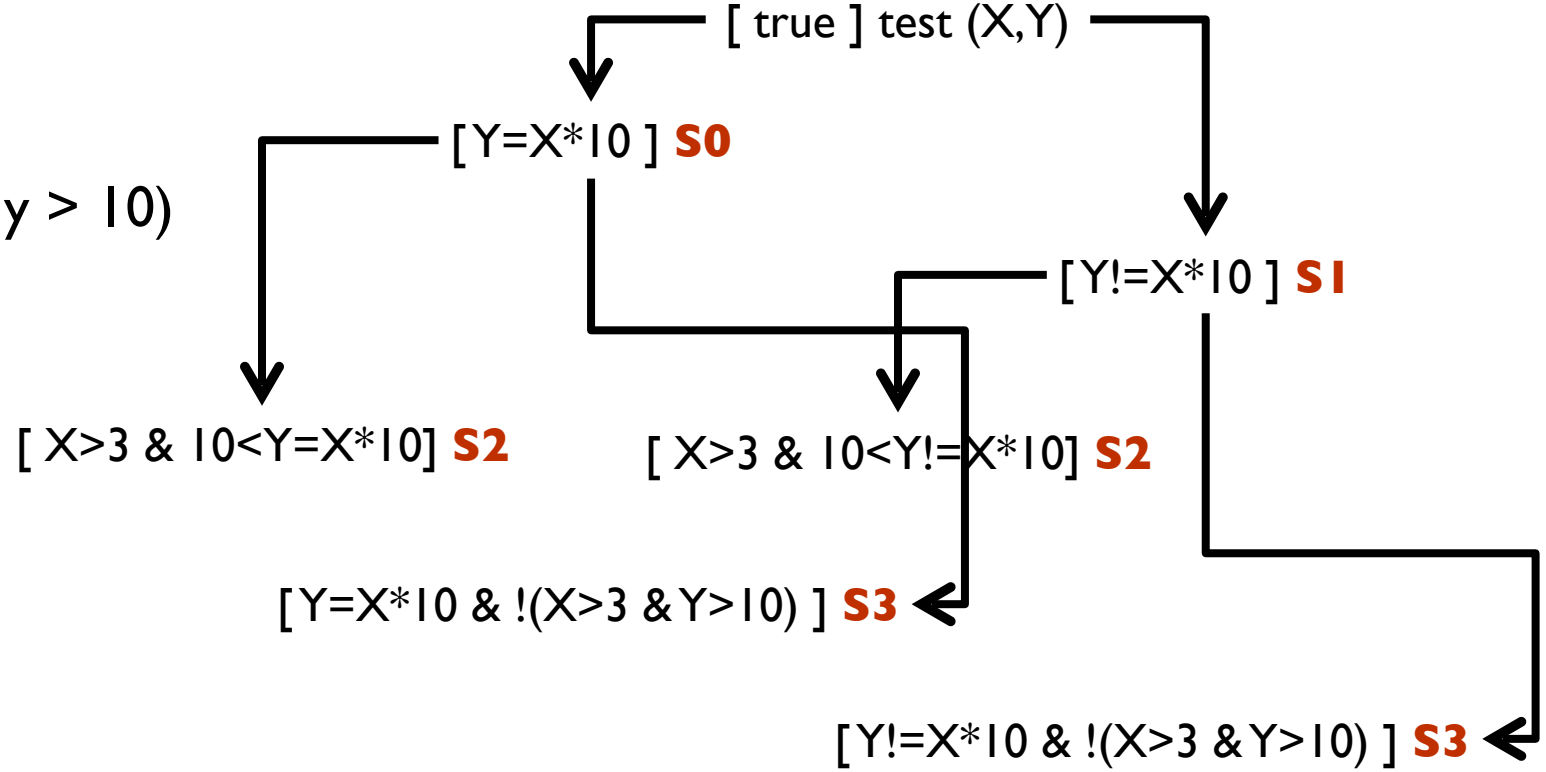
LattE Model Counter

http://www.math.ucdavis.edu/~latte/

Count solutions for
conjunction
of linear inequalities

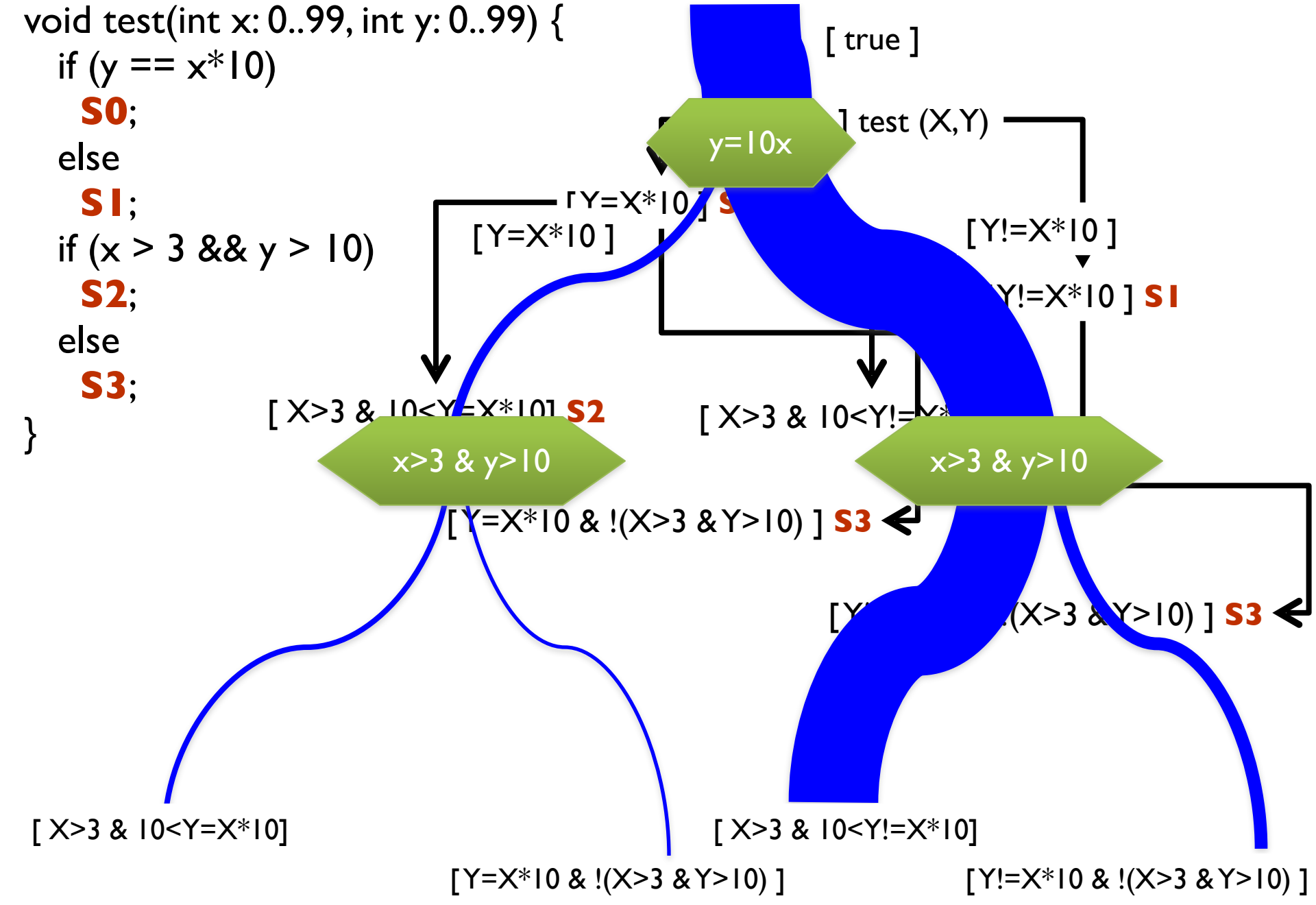
Symbolic Execution

```
void test(int x, int y) {
   if (y == x*10)
     S0;
   else
     S1;
   if (x > 3 && y > 10)
     S2;
   else
     S3;
}
```



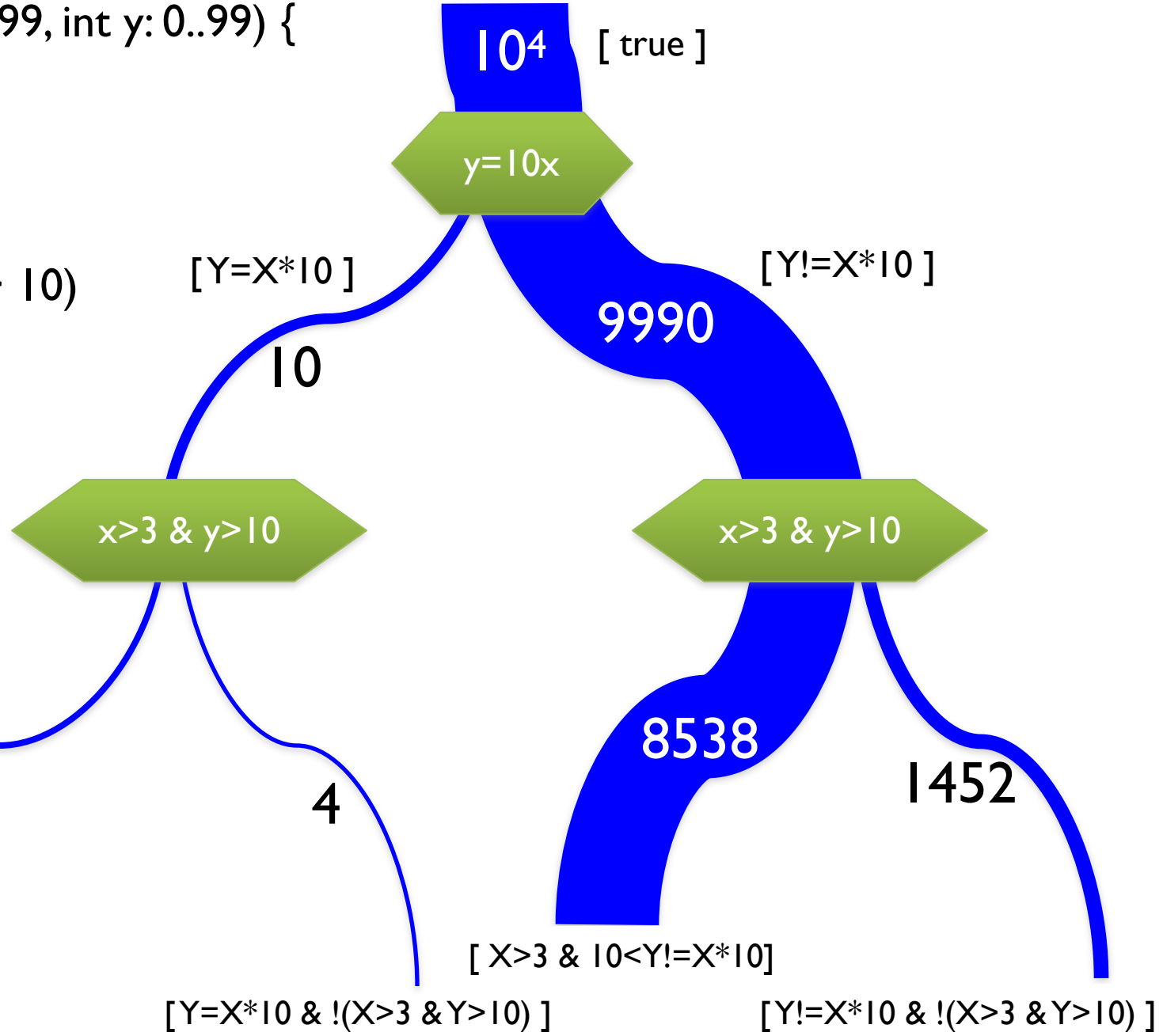
Probabilistic Symbolic Execution

```
void test(int x: 0..99, int y: 0..99) {
   if (y == x*10)
     S0;
   else
     S1;
   if (x > 3 && y > 10)
     S2;
   else
     S3;
}
```



Probabilistic Symbolic Execution

```
void test(int x: 0..99, int y: 0..99) {
   if (y == x*10)
     S0;
   else
     S1;
   if (x > 3 && y > 10)
     S2;
   else
     S3;
}
```



Program Understanding

10^4 [true]

y=10x

[Y=X*10]

10

[Y!=X*10]

9990

x>3 & y>10

x>3 & y>10

6

4

8538

1452

[X>3 & 10<Y=X*10]

[Y=X*10 & !(X>3 & Y>10)]

[X>3 & 10<Y!=X*10]

[Y!=X*10 & !(X>3 & Y>10)]

Probabilistic Analysis

10^4 [true]

y=10x

[Y=X*10] 0.001

[Y!=X*10] 0.999

x>3 & y>10

x>3 & y>10

0.6

0.4

0.854

0.145

0.0006

[X>3 & 10<Y=X*10]

0.0004

[Y=X*10 & !(X>3 & Y>10)]

0.853

[X>3 & 10<Y!=X*10]

0.145

[Y!=X*10 & !(X>3 & Y>10)]

Software Reliability

10^4 [true]

y=10x

[Y=X*10]

0.001

[Y!=X*10]

0.999

0.9996 Reliable

x>3 &

y>10

0.6

0.4

0.854

0.145

0.0006

[X>3 & 10<Y=X*10]

0.0004

[Y=X*10 & !(X>3 & Y>10)]

0.853

[X>3 & 10<Y!=X*10]

0.145

[Y!=X*10 & !(X>3 & Y>10)]

Future Directions

- Incorporate usage profiles
- Extend model counting to other types
- Reduce, reuse and recycle constraints
- Use informed sampling for statistical SE
- Target non-deterministic programs
- …