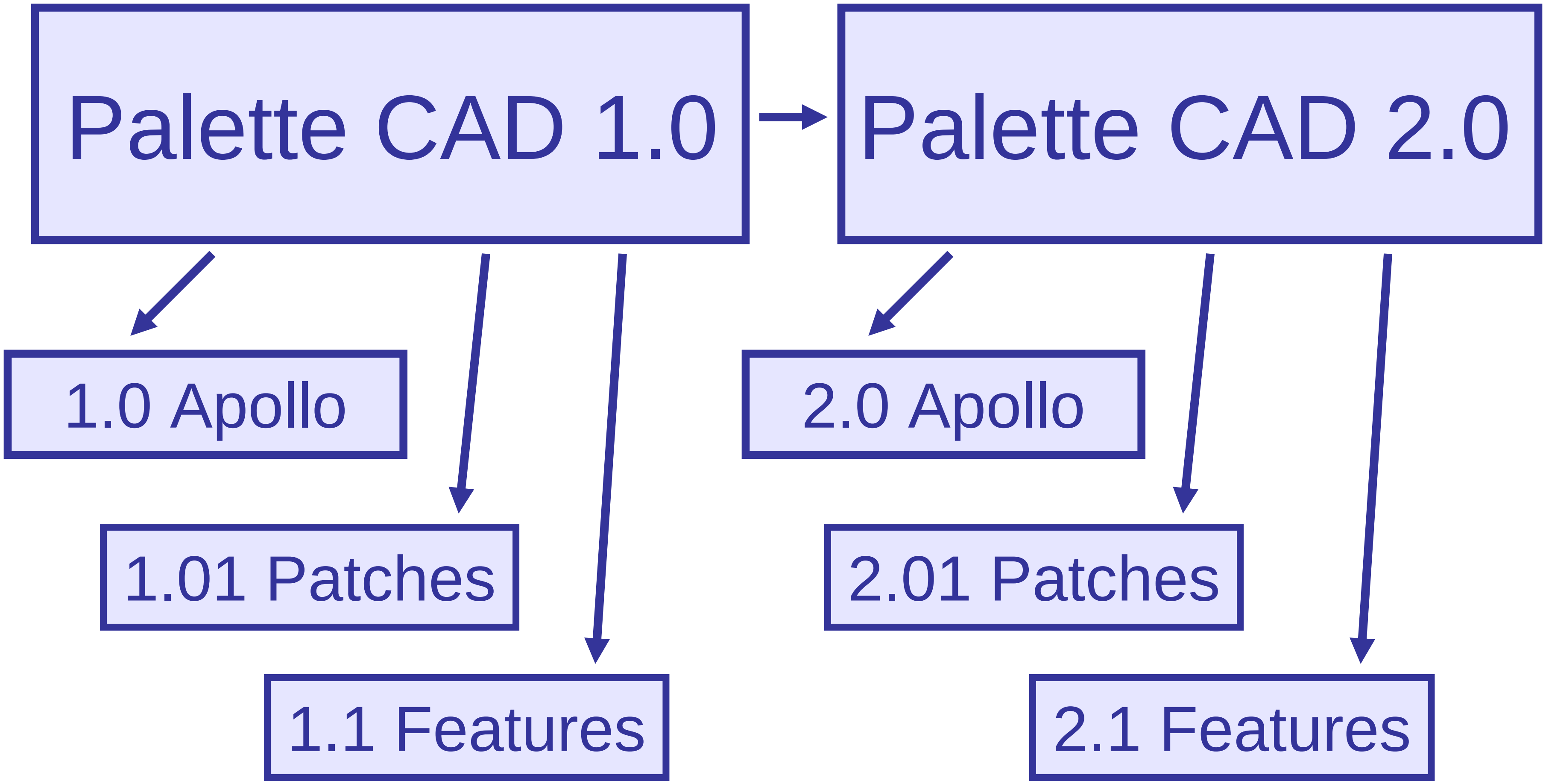
Software Verification: An Evolution-Centric Perspective

Gregg Rothermel

Dept. of Computer Science and Engineering
University of Nebraska - Lincoln

Supported by the National Science Foundation,
Microsoft, Lockheed Martin,
and Boeing Commercial Aircraft Group

Evolving Software



An Evolution-Centric Perspective on Software Verification

- Focus on evolution first
- Harness evolution
- Design for incremental validation

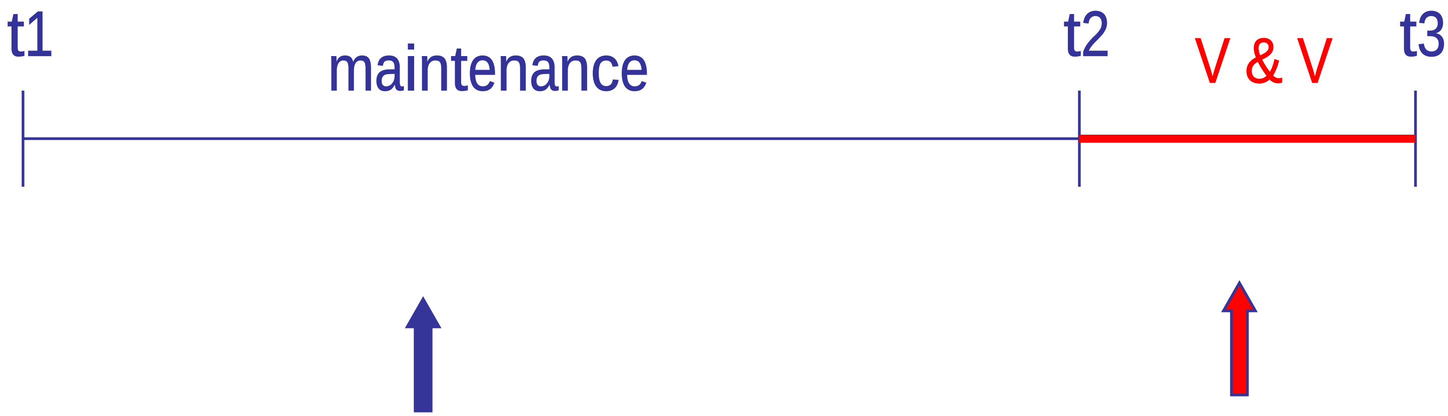
Overview of Presentation

- Evolving software
- Regression test selection
  - Dejavu algorithm
  - Analytical and empirical evaluation
- Regression model checking
  - Algorithm
  - Empirical evaluation
- Ongoing and Future Work

Overview of Presentation

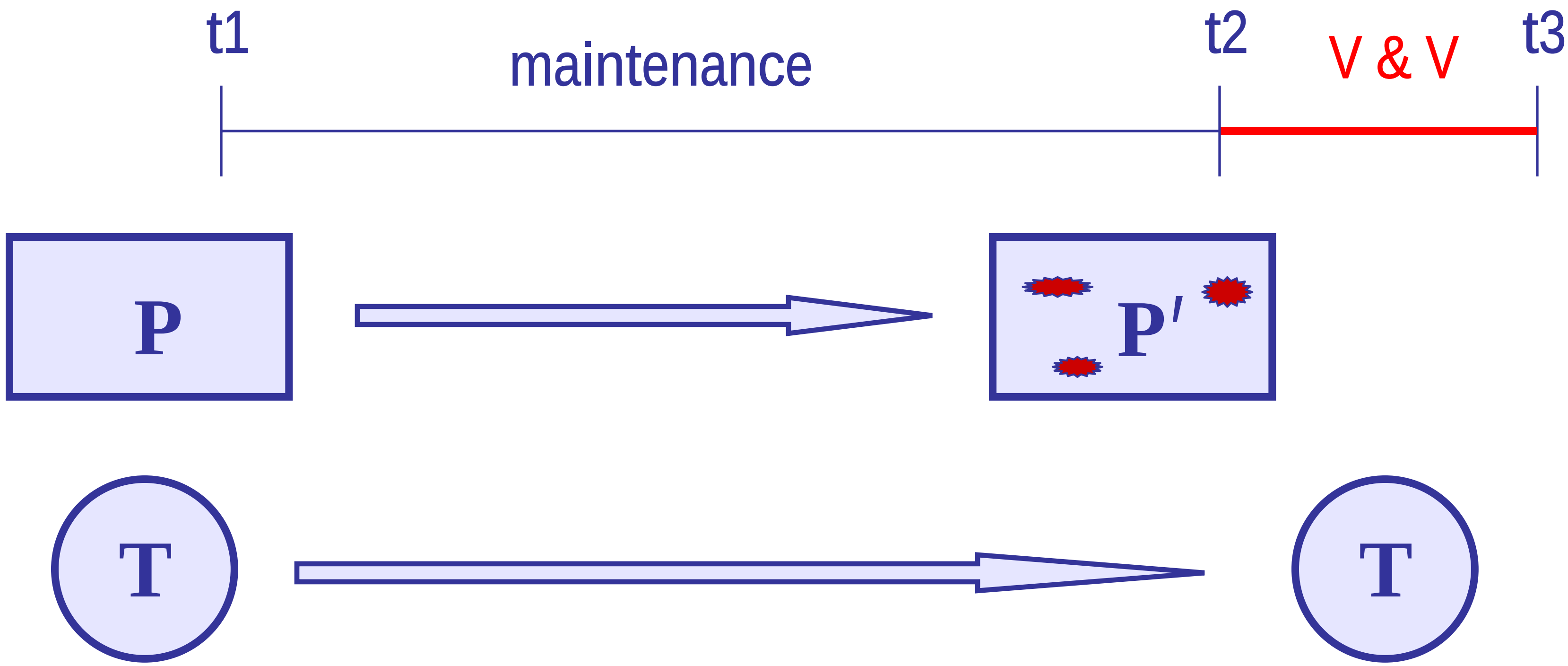
- Evolving software
- Regression test selection
  - Dejavu algorithm
  - Analytical and empirical evaluation
- Regression model checking
  - Algorithm
  - Empirical evaluation
- Ongoing and Future Work

Evolving Software:
A Basic Process Model



Preliminary period   Critical period

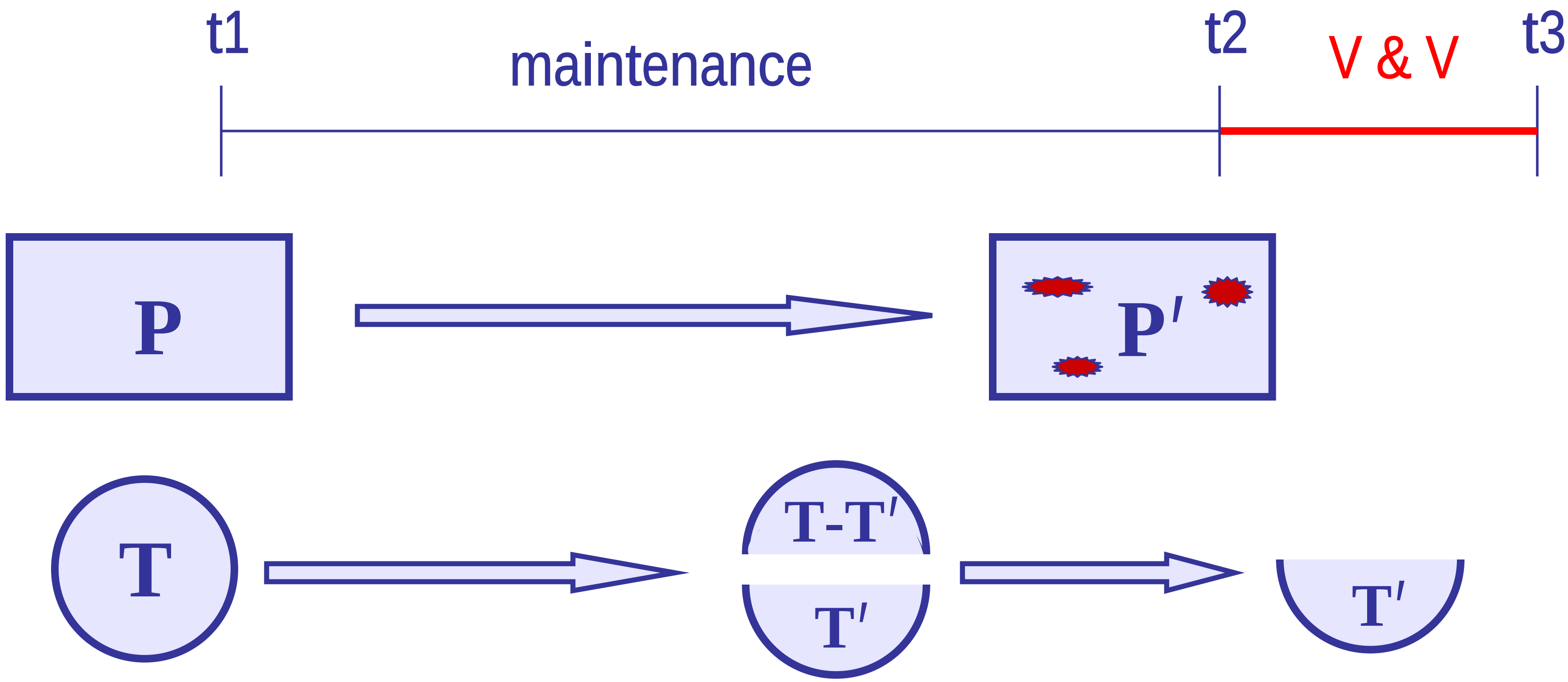
Evaluating Evolving Software with Regression Testing



Overview of Presentation

- Evolving software
- Regression test selection
  - Dejavu algorithm
  - Analytical and empirical evaluation
- Regression Model Checking
  - Algorithm
  - Empirical evaluation
- Ongoing and Future Work

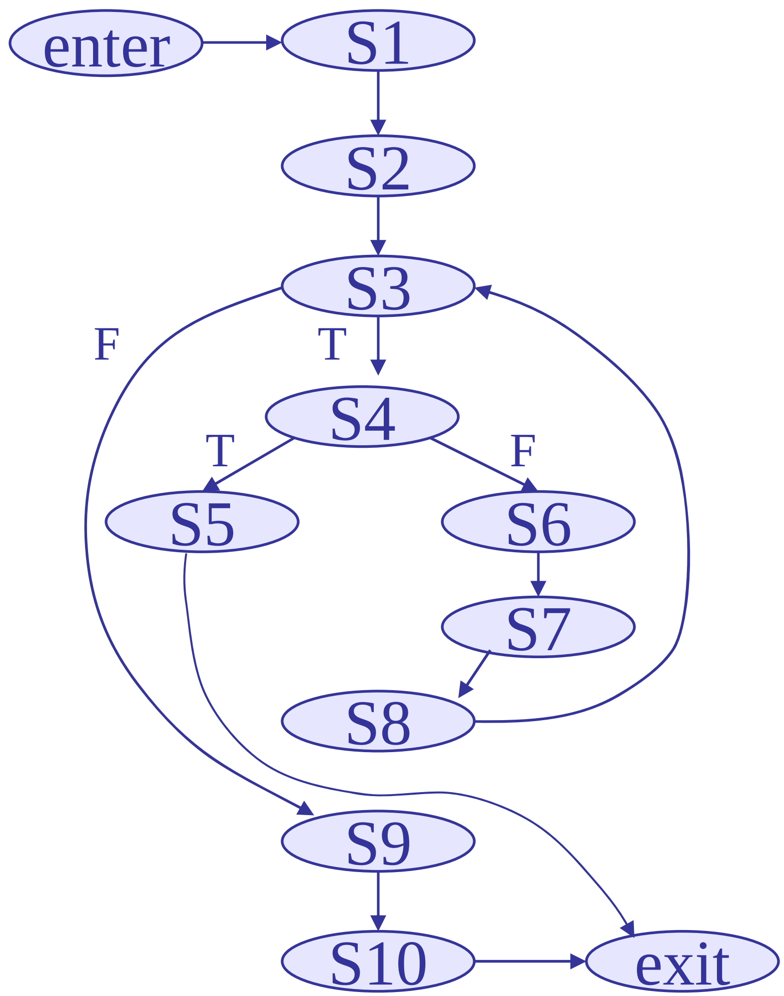
Regression Test Selection



Control Flow Graphs

Procedure Avg

```
S1   count = 0
S2   fread(fptr,n)
S3   while (not EOF) do
S4      if (n<0)
S5         return(error)
        else
S6         nums[count] = n
S7         count++
        endif
S8      fread(fptr,n)
     endwhile
S9   avg = mean(nums,count)
S10  return(avg)
```



Execution Traces

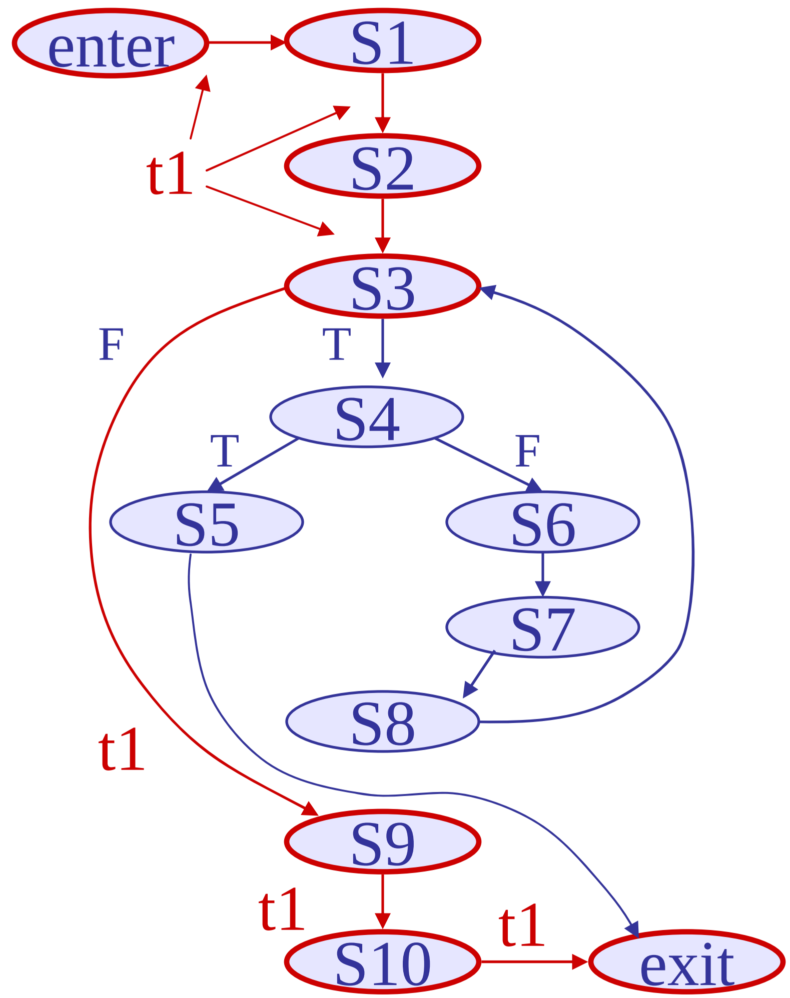
```
Procedure Avg
S1   count = 0
S2   fread(fptr,n)
S3   while (not EOF) do
S4     if (n<0)
S5        return(error)
       else
S6        nums[count] = n
S7        count++
       endif
S8     fread(fptr,n)
     endwhile
S9   avg = mean(nums,count)
S10  return(avg)
```

test	input	output
t1	empty file	0

Execution Traces

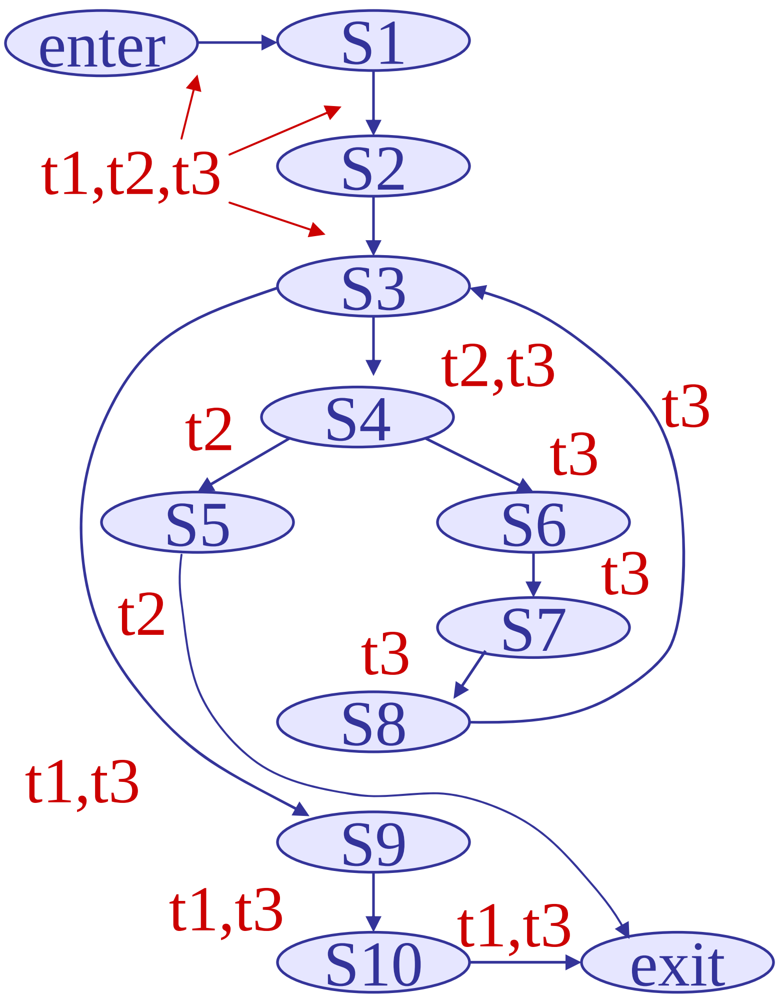
Procedure Avg

```
S1   count = 0
S2   fread(fptr,n)
S3   while (not EOF) do
S4      if (n<0)
S5         return(error)
        else
S6         nums[count] = n
S7         count++
        endif
S8      fread(fptr,n)
     endwhile
S9   avg = mean(nums,count)
S10  return(avg)
```



Test History Information

test	input	output
t1	empty file	0
t2	-1	error
t3	1 2 3	2



Program and Modified Version

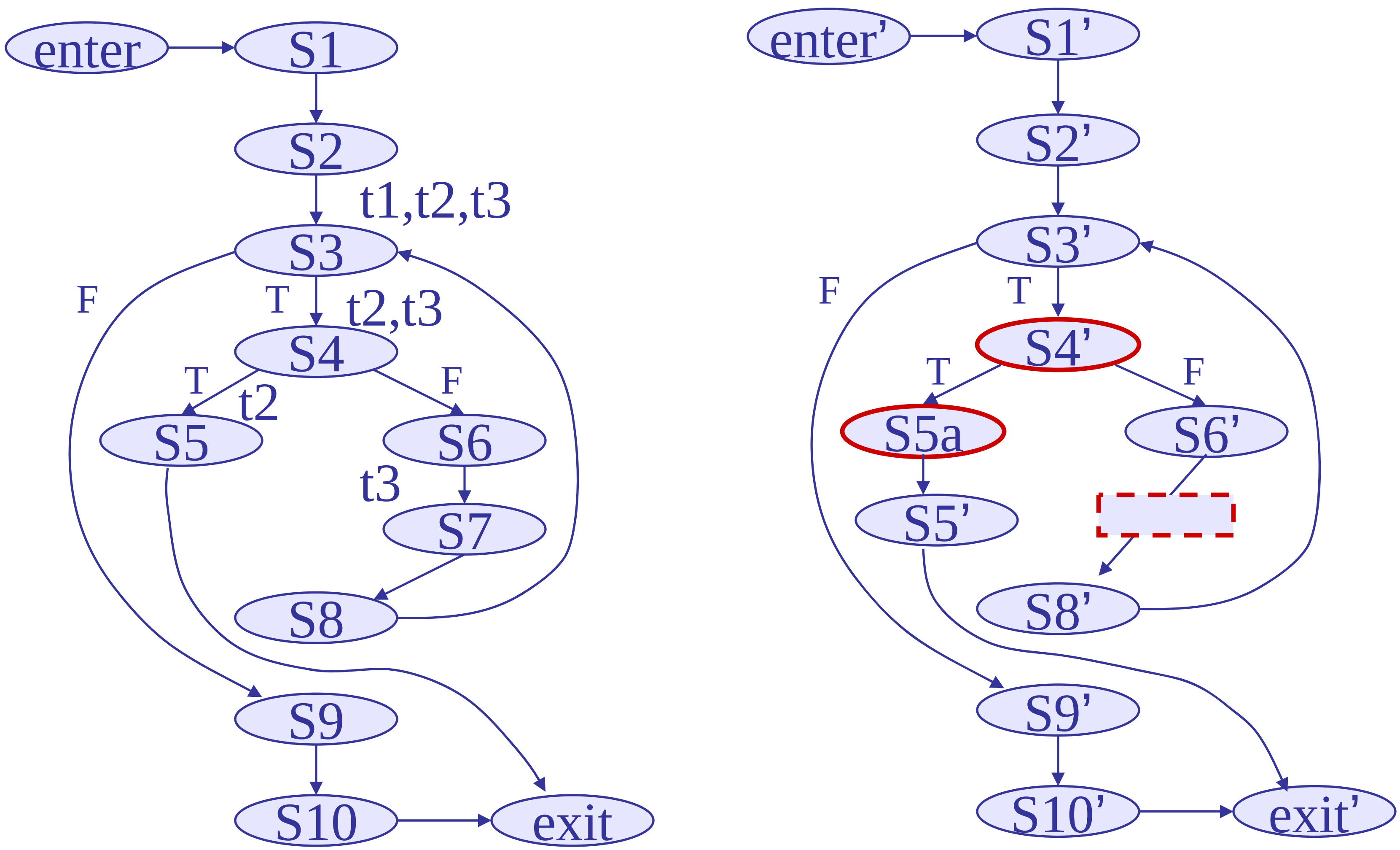
Procedure Avg

```
S1   count = 0
S2   fread(fptr,n)
S3   while (not EOF) do
S4      if (n<0)

S5         return(error)
        else
S6         nums[count] = n
S7         count++
        endif
S8      fread(fptr,n)
     endwhile
S9   avg = mean(nums,count)
S10  return(avg)
```

Procedure Avg′

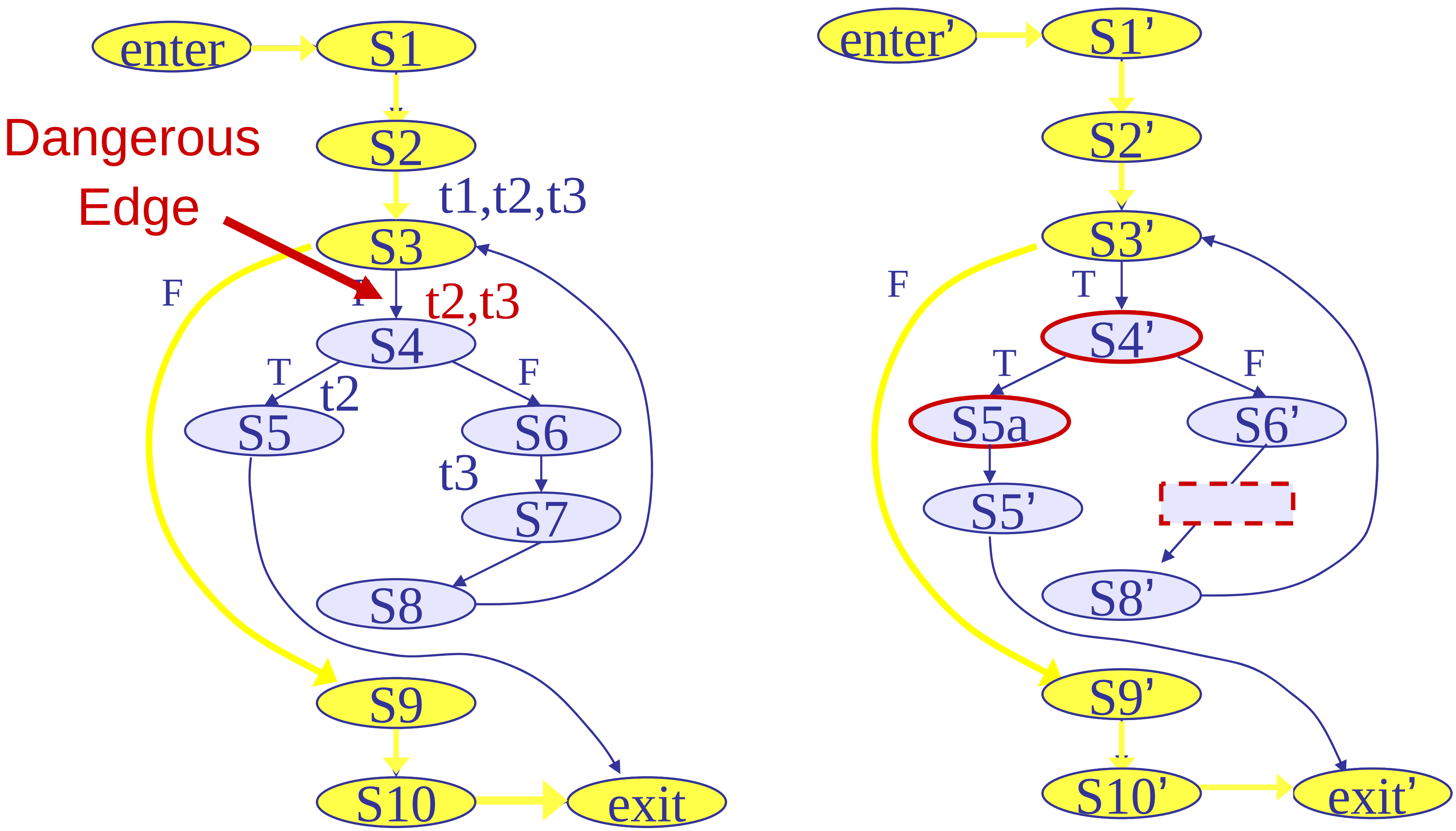
```
S1'  count = 0
S2'  fread(fptr,n)
S3'  while (not EOF) do
S4'     if (n<=0)
S5a        print("input error")
S5'        return(error)
        else
S6'        nums[count] = n

        endif
S8'     fread(fptr,n)
     endwhile
S9'  avg = mean(nums,count)
S10' return(avg)
```

CFG and Modified CFG



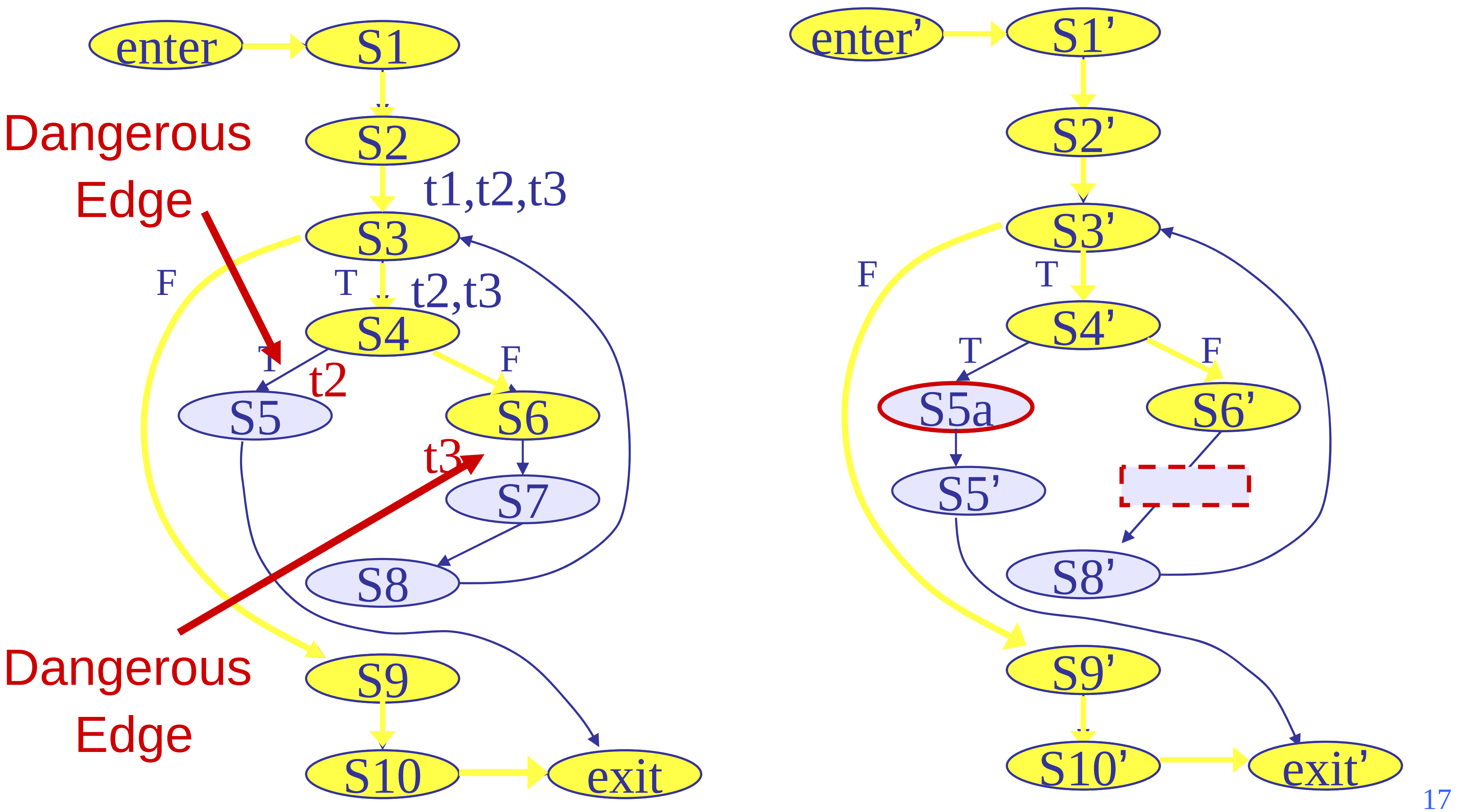
Example 1

T' = {t2, t3}



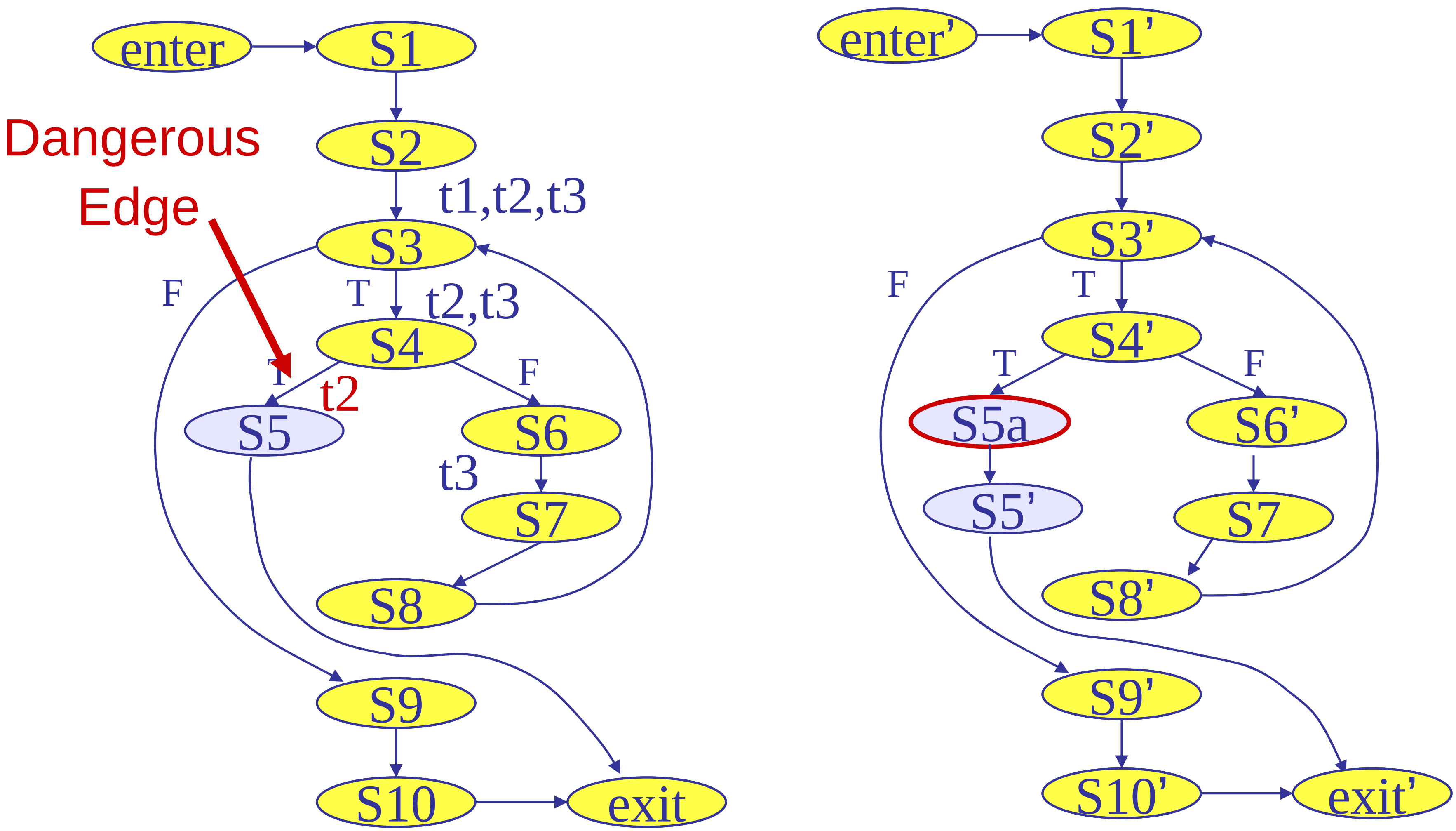
Example 2

T' = {t2, t3}



Example 3

T’ = {t2}



Algorithm Dejavu

Input: P, P', T   Output: T'

1. Build CFGs G and G' for P and P'
2. Compare(G.EntryNode,G'.EntryNode)

3. Compare(N,N')
4.   mark N "N'-visited"
5.   for each pair of successors C and C' of N and N'
6.   on equivalently labeled edges do
7.   if C is not marked "C'-visited"
8.   if C and C' are not lexically identical
9. $T' = T' \cup$ TestsOnEdge(N,C,T)
10.   else
11.   Compare(C,C')

Interprocedural Methodologies

1. Compare all pairs of procedures
2. Create & walk interprocedural representation
3. Compare all pairs of procedures identified by configuration management system

Overview of Presentation

- Evolving software
- Regression test selection
  - Dejavu algorithm
  - Analytical and empirical evaluation
- Regression model checking
  - Algorithm
  - Empirical evaluation
- Ongoing and Future Work

Algorithm Efficiency

CFG construction: linear in program size

Graph walk (graph sizes n, n'; test set size t):

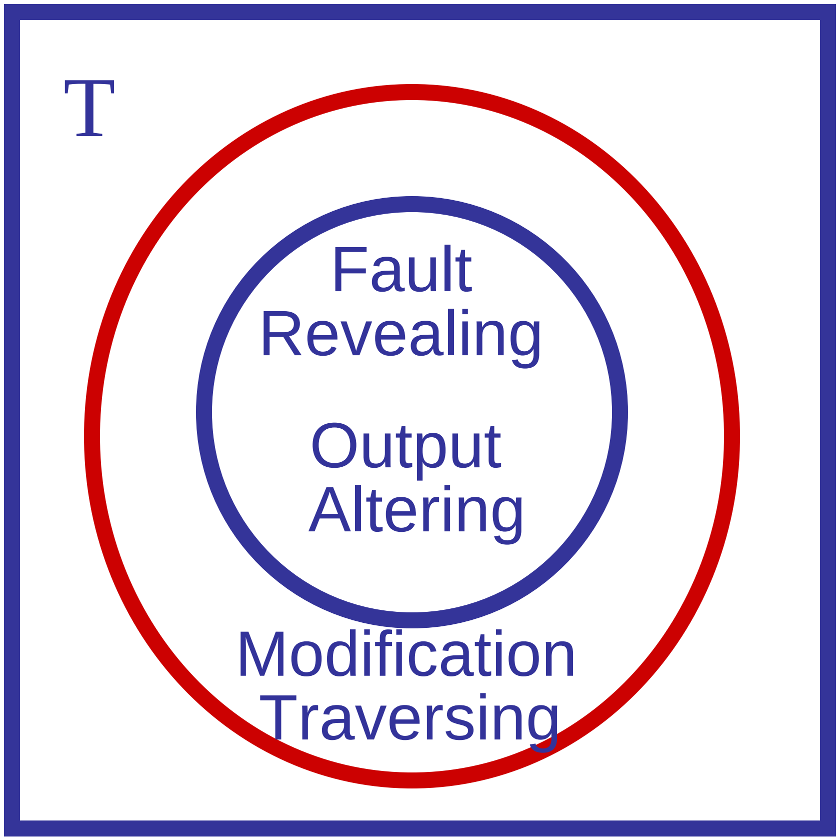
$O(t * n * n')$

(with *multiply-visited nodes*)

$O(t * \min(n,n'))$

(with no *multiply-visited nodes*)

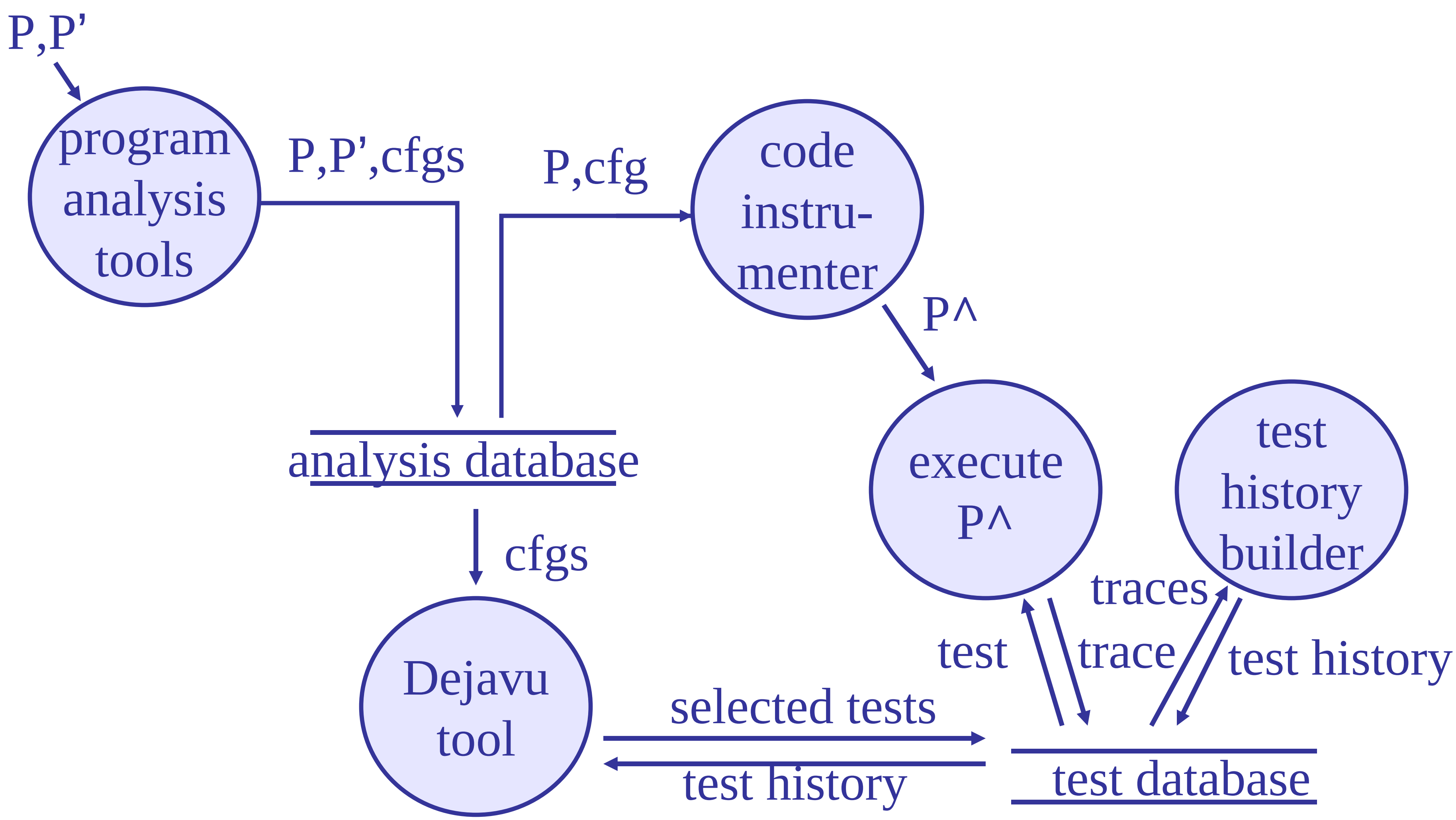
Precision and Safety



Conditions:

1. P was correct for all tests in T
2. T contains no obsolete tests
3. Controlled regression testing

Regression Test Selection System

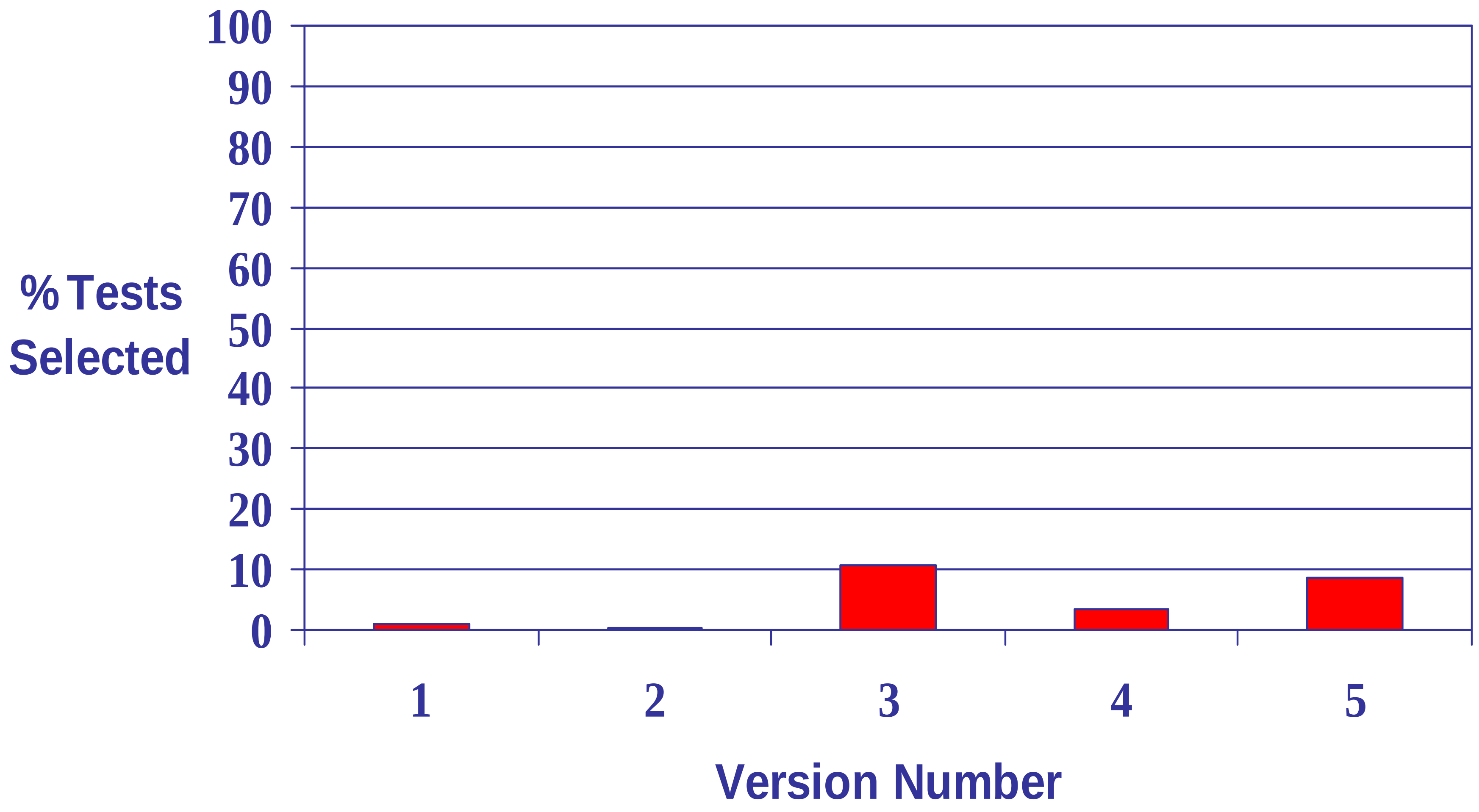


Study 1: Empire

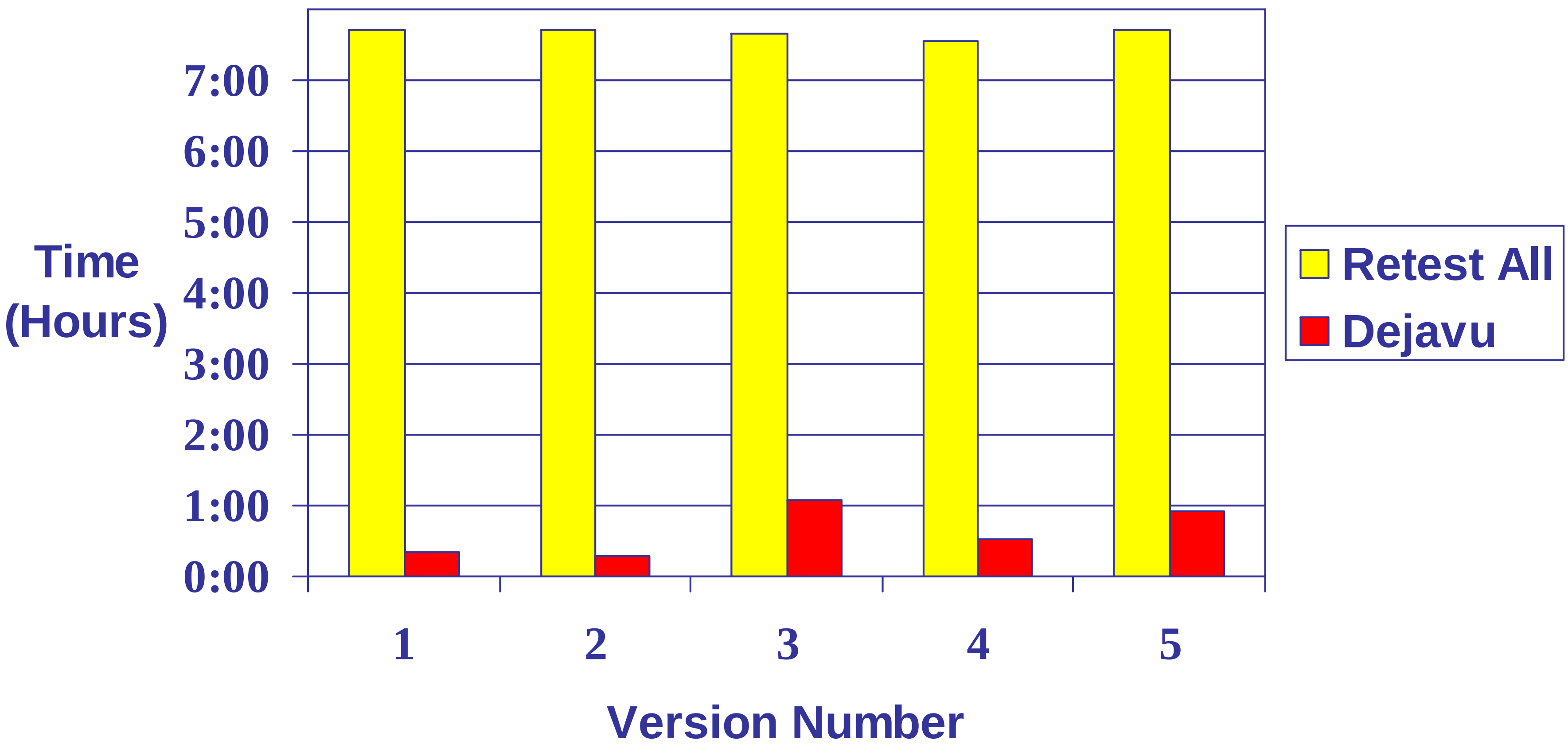
Program	Procs	LOC	Vers	Tests
server	766	49316	5	1033

Version	Functions Modified	LOC Modified
1	3	114
2	2	55
3	11	726
4	11	62
5	42	221

Study 1: Test Selection Percentages



Study 1:
Cost Effectiveness

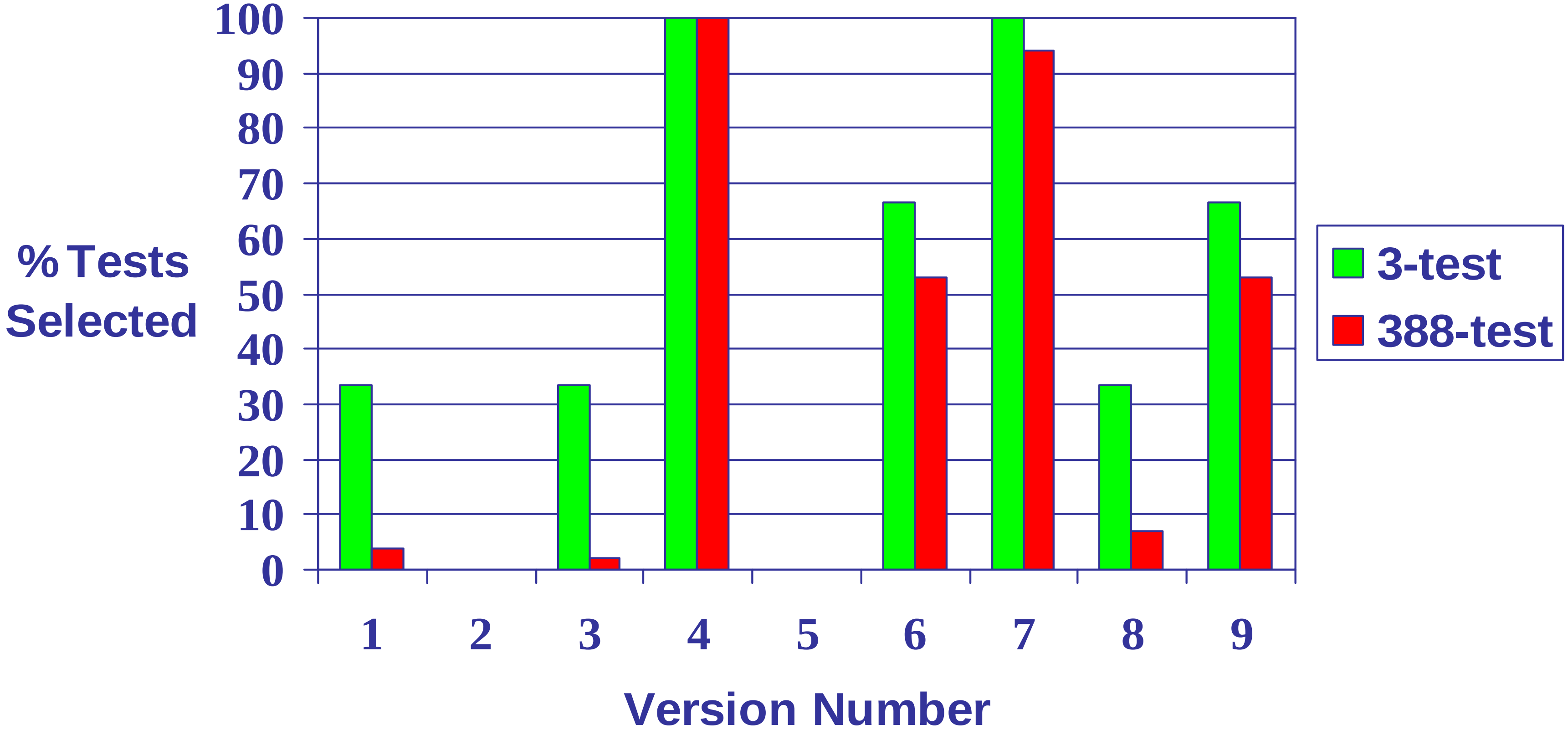


Study 2:
Windows NT Calculator

Program	Funcs	LOCs	Vers	Tests
calculator	27	2145	9	3/388

Version	Functions Modified	LOCs Modified
1	1	1
2	2	12
3	1	1
4	12	264
5	1	3
6	3	4
7	3	245
8	2	15
9	2	44

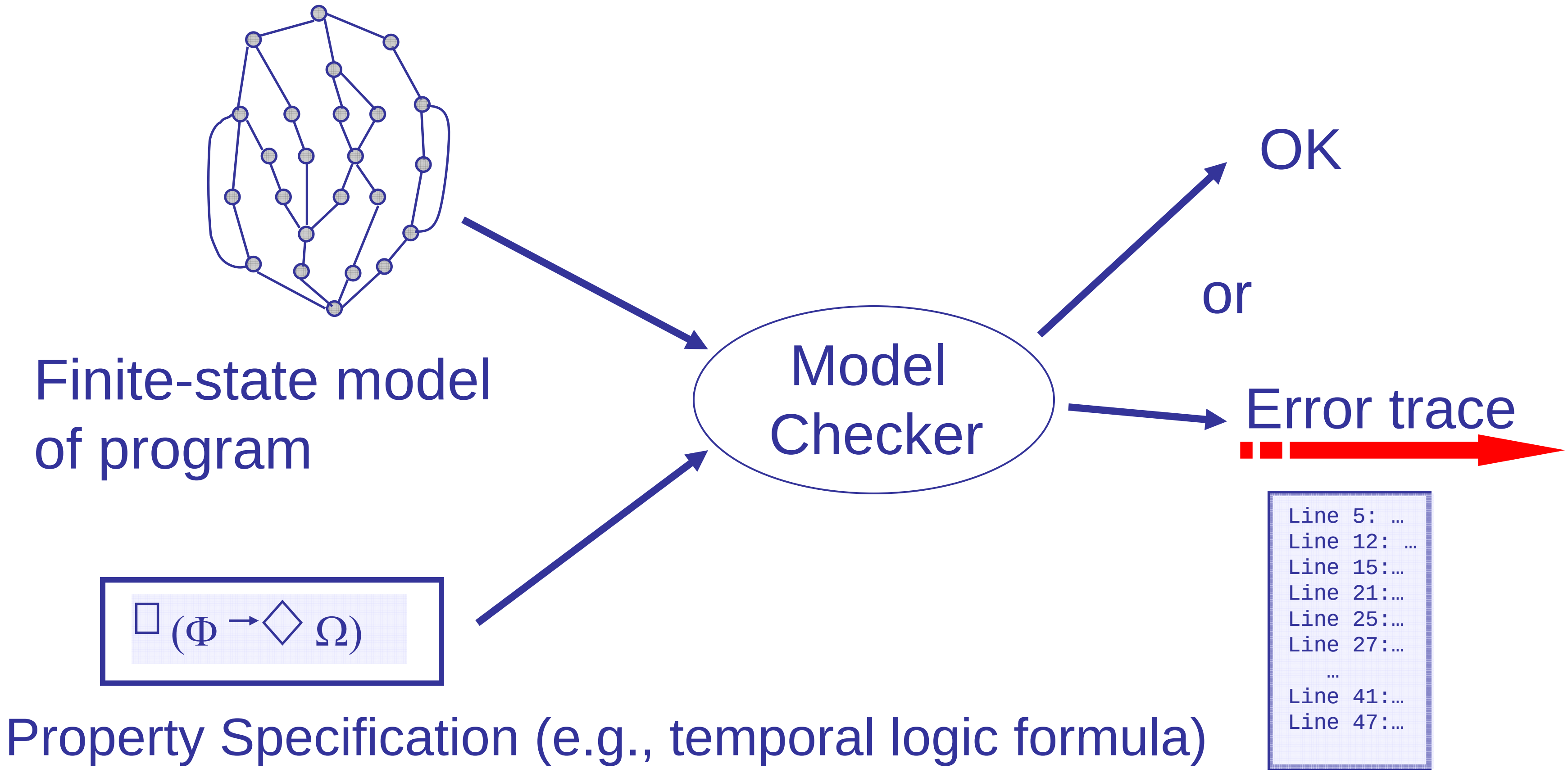
Study 2:
Test Selection Percentages



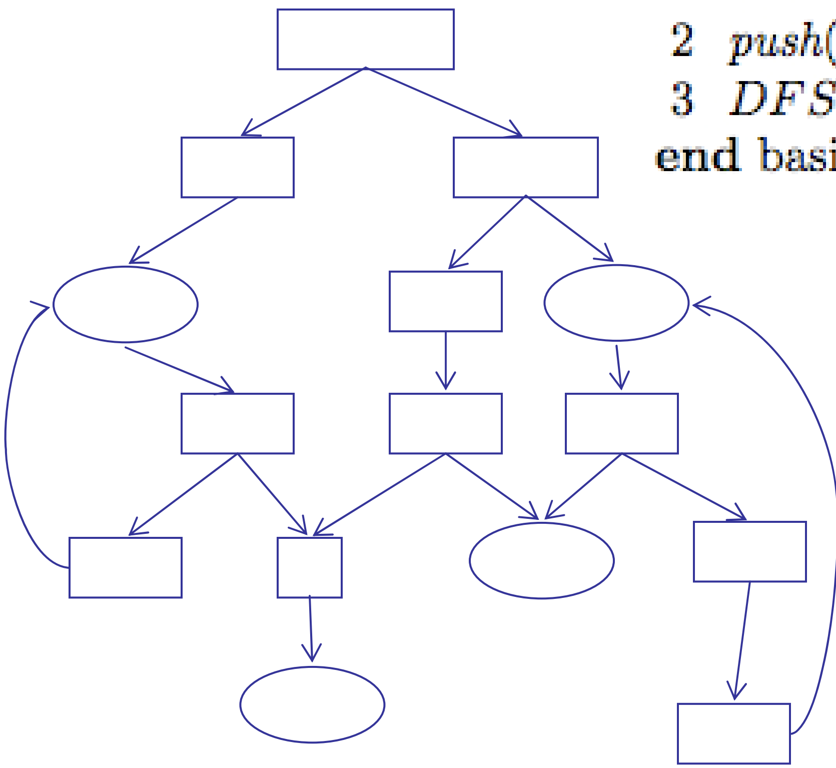
Overview of Presentation

- Evolving software
- Regression test selection
  - Dejavu algorithm
  - Analytical and empirical evaluation
- Regression model checking
  - Algorithm
  - Empirical evaluation
- Ongoing and Future Work

Model Checking



Basic DFS Approach



basicDFS()
  $seen := \{s_0\}$
  $push(stack, s_0)$
  $DFS(s_0)$
end basicDFS()

DFS(s)
  for $\alpha \in enabled(s)$ **do**
    $s' := \alpha(s)$
    if $error(s')$ **then**
      $ce := stack$
      exit
    if $s' \notin seen$ **then**
      $seen := seen \cup \{s'\}$
      $push(stack, s')$
      $DFS(s')$
      $pop(stack)$
  endfor
end DFS()

Why use Model Checking?

- Automatically check, e.g.,
  - invariants, safety & liveness properties
  - absence of dead-lock and live-lock,
  - complex event sequencing properties,

> “Between the key being inserted and the key being removed, the ignition can be activated at most twice.”

- In contrast to testing, model checking gives complete coverage by exhaustively exploring all paths in a system

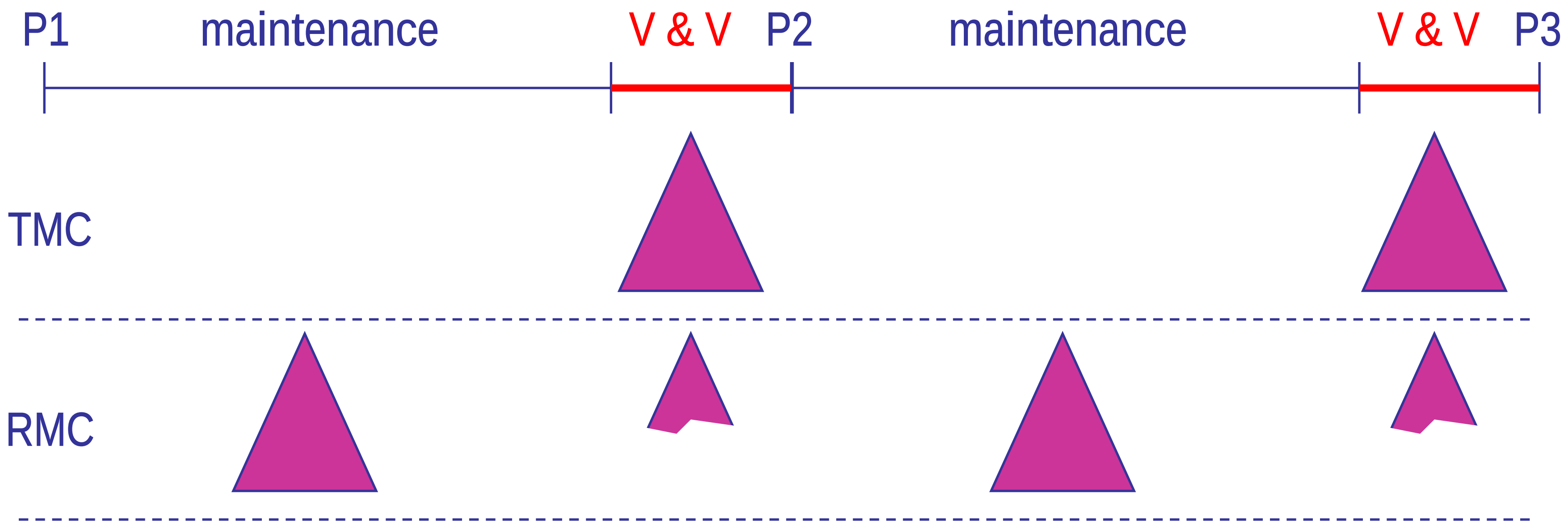
What Makes Model Checking Difficult to Apply in Practice?

- Model construction
- Property specification
- State explosion
- Output interpretation

What Makes Model Checking Difficult to Apply in Practice?

- Model construction
- Property specification
- State explosion
- Output interpretation
- System evolution

Regression Model Checking



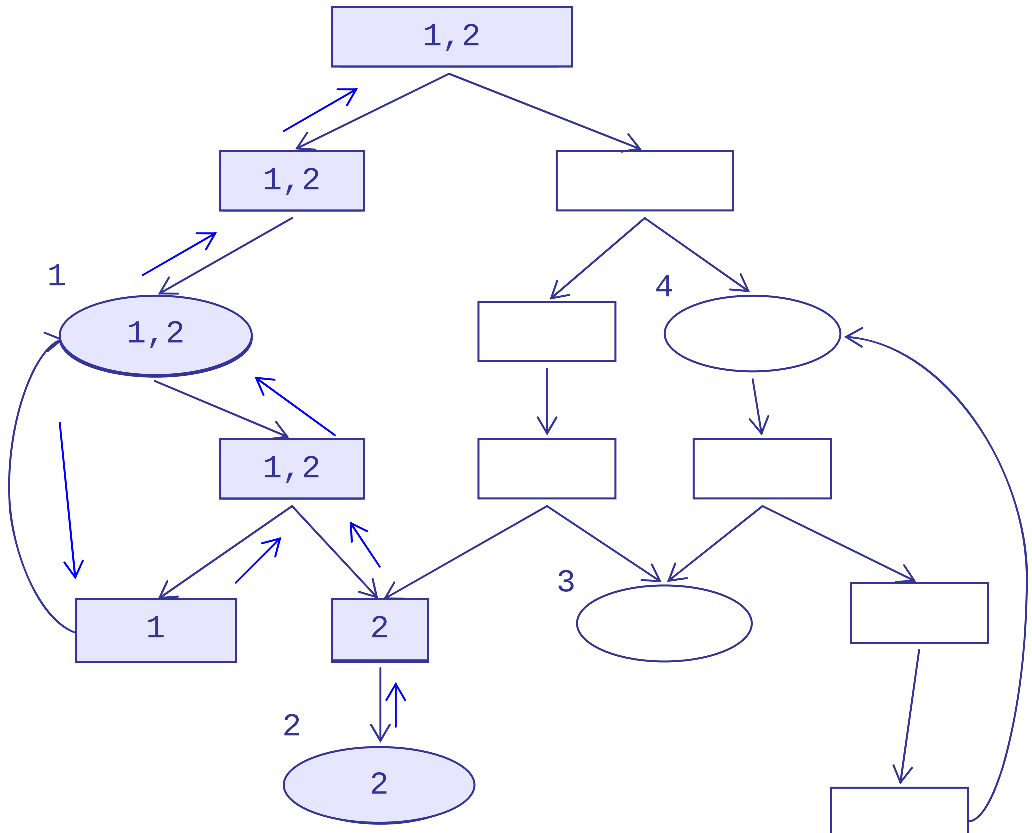
Overview of Presentation

- Testing evolving software
- Regression test selection
  - Dejavu algorithm
  - Analytical and empirical evaluation
- Regression model checking
  - Approach
  - Empirical results
- Ongoing and Future Work

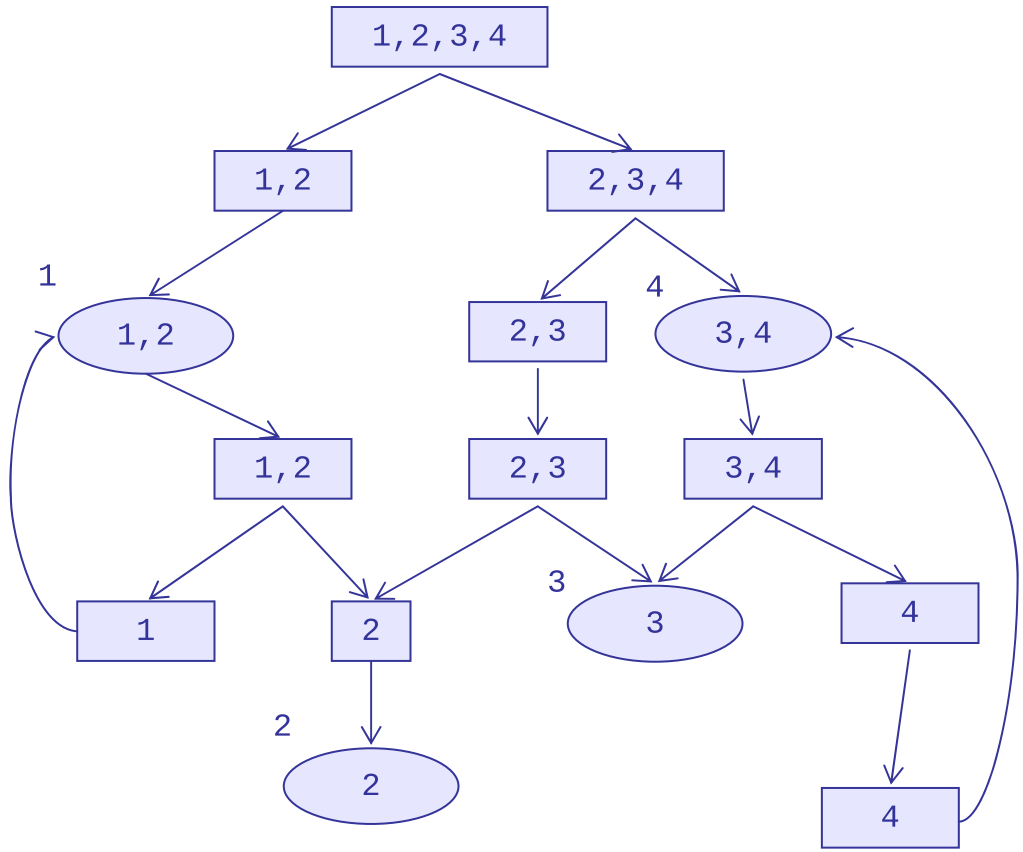
Regression Model Checking (RMC) Process

1) A *recording phase* of RMC (recordingRMC) gathers data while model checking P in the preliminary period
2) Dejavu computes the dangerous elements with respect to P and P'
3) A *pruning phase* of RMC (pruningRMC) determines which sub-state spaces in P' are safe and prunes them in the critical period, performing model checking only where needed
4) RMC preserves property checking relative to a traditional model check

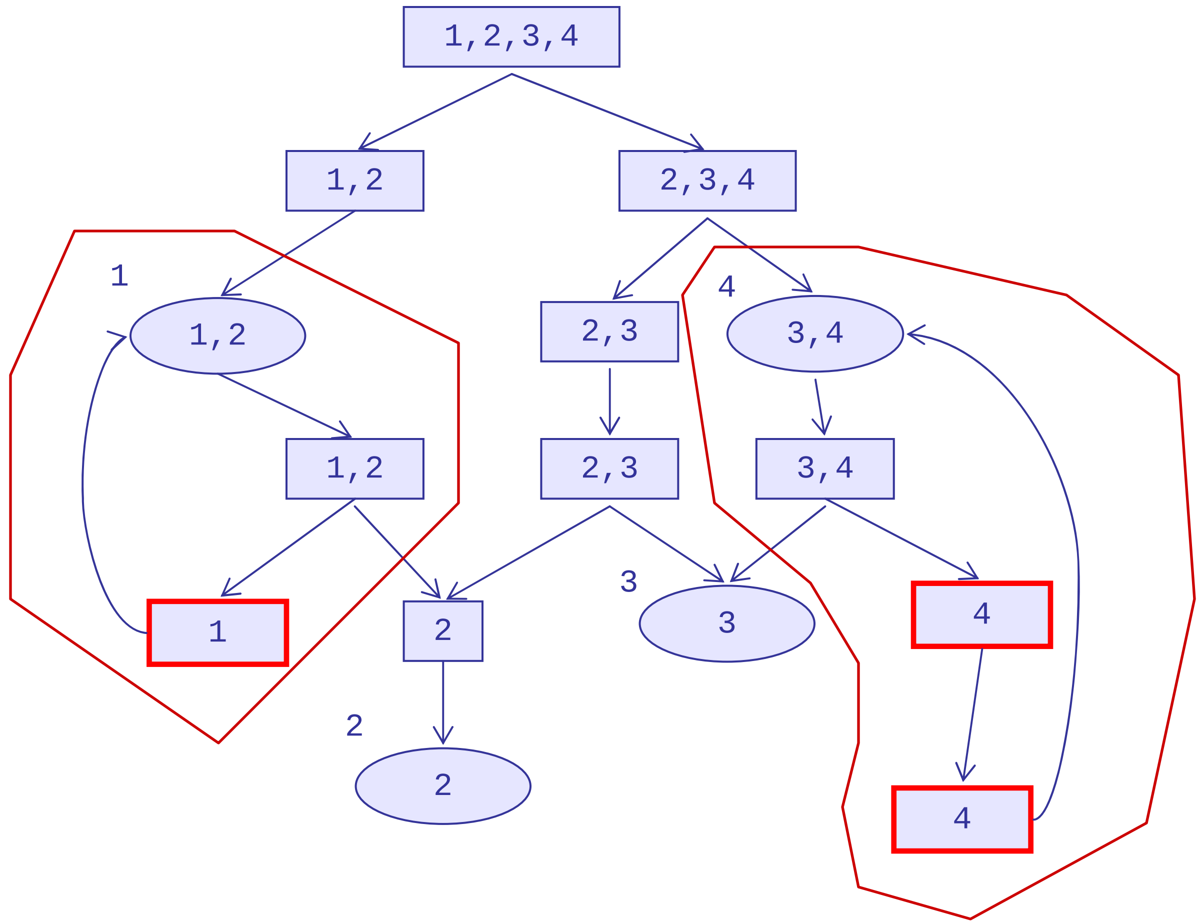
RecordingRMC



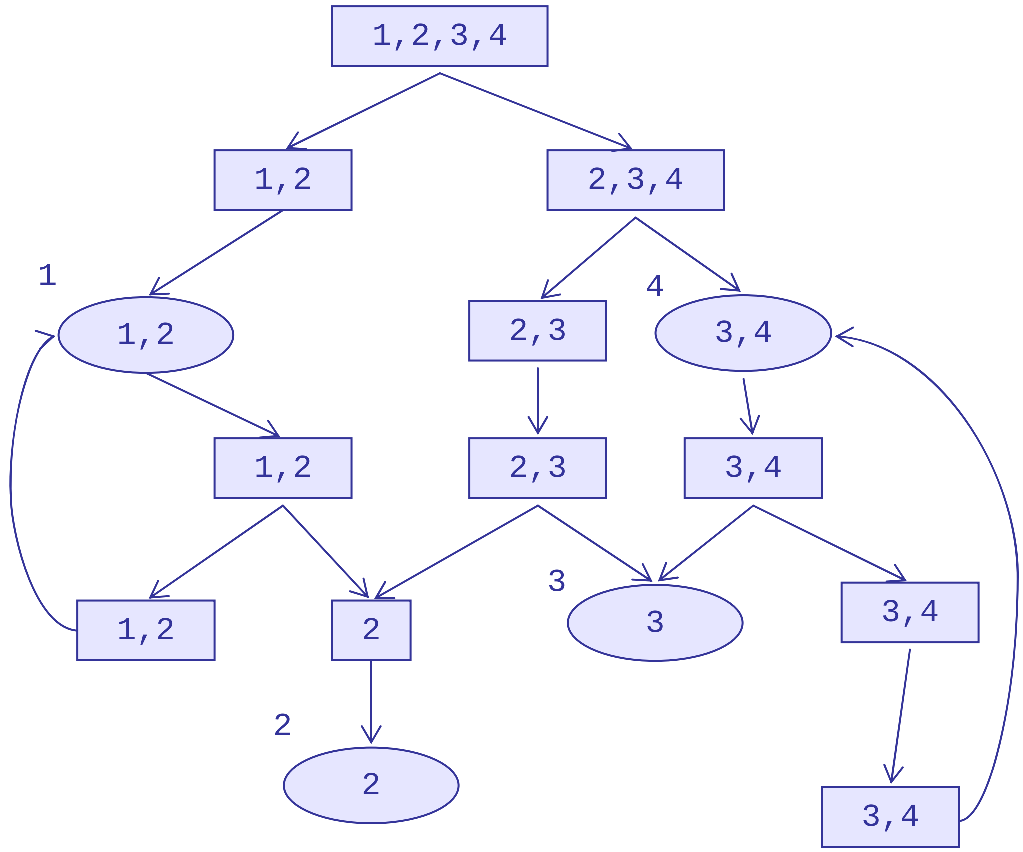
RecordingRMC



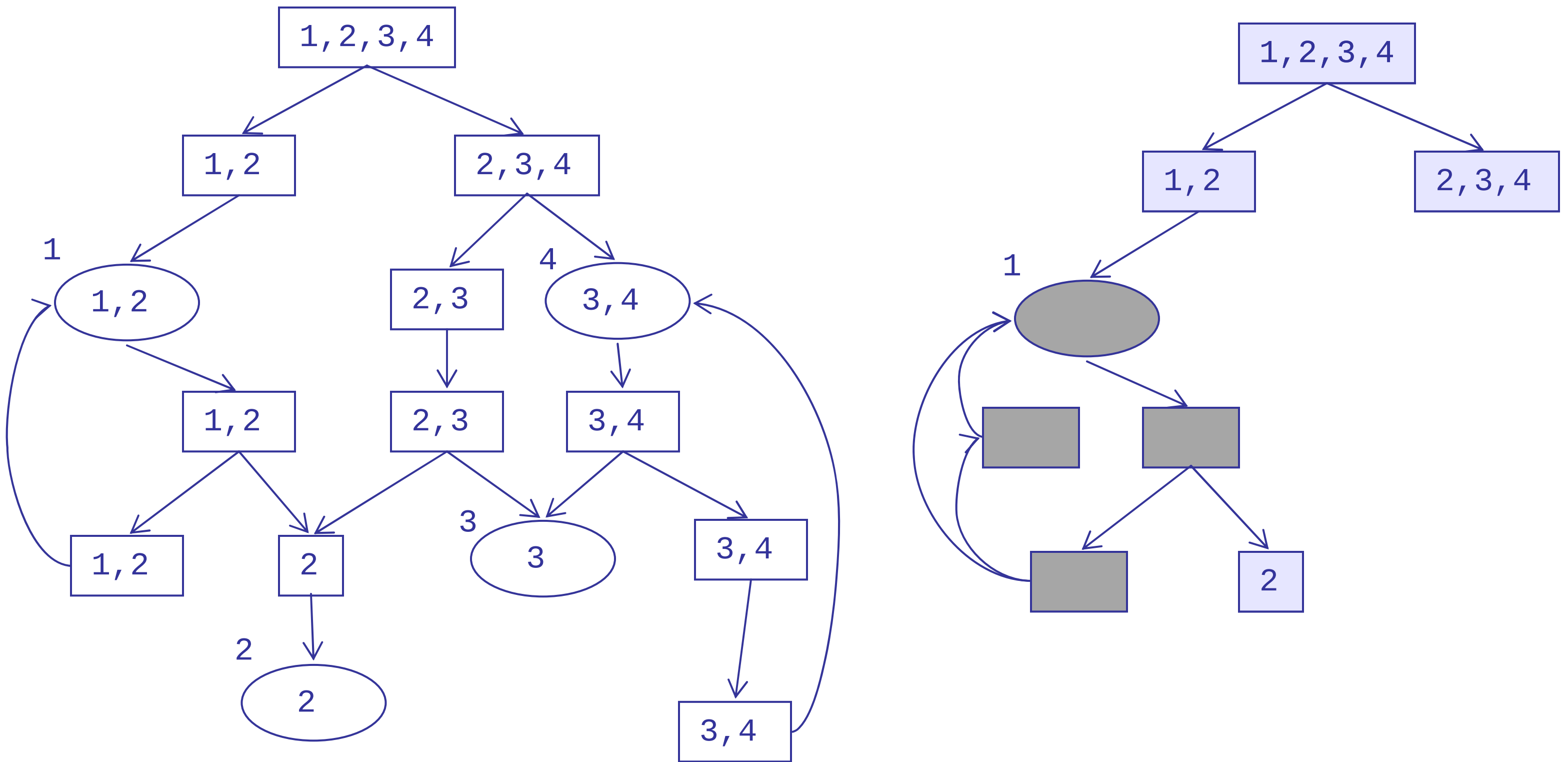
RecordingRMC



RecordingRMC

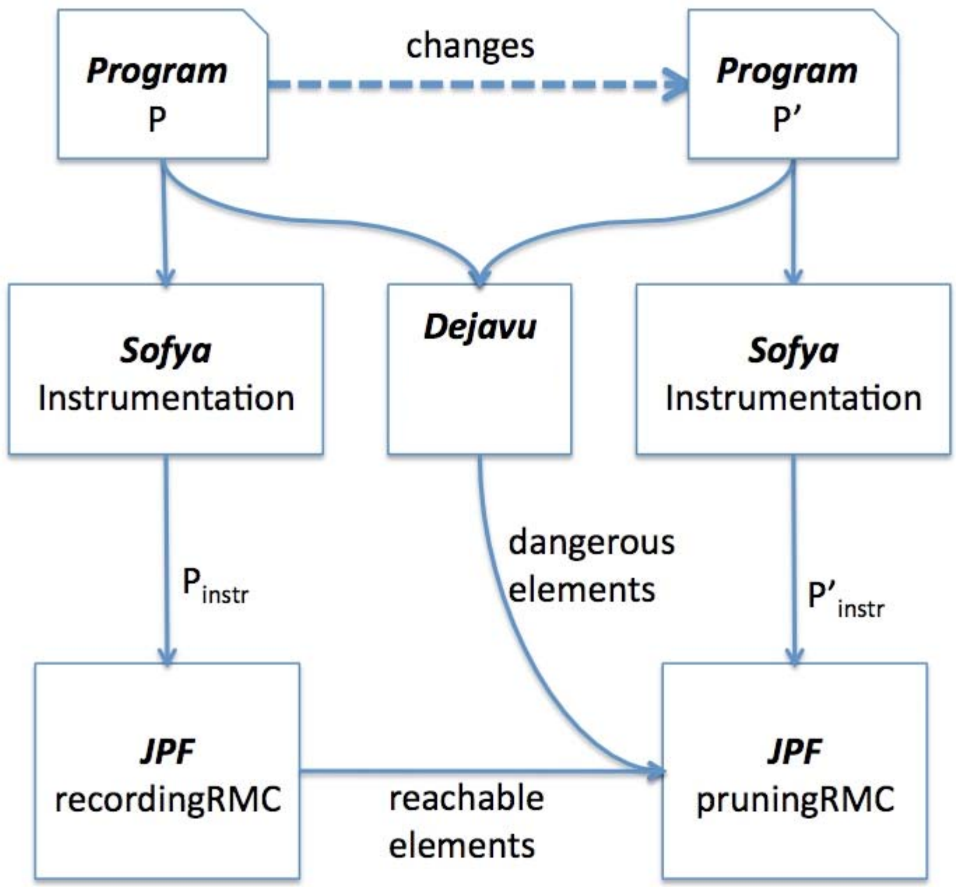


PruningRMC



Dangerous elements = {1}

RMC Implementation



Overview of Presentation

- Evolving software
- Regression test selection
  - Dejavu algorithm
  - Analytical and empirical evaluation
- Regression model checking
  - Algorithm
  - Empirical evaluation
- Ongoing and Future Work

Research Question

RQ1: How does RMC compare in the critical period to TMC

RQ2: How does RMC compare overall to TMC

Objects

Object	*Error*	*Thd*	*Cls*	*Meth*	*LOC*	*Vers*
Daisy	AssertionViolation	3	21	106	744	13
Elevator	ArrayException	4	12	96	934	29
Alarmclock	NullPtrException	3	6	20	125	13
RaxExtended	AssertionViolation	6	11	23	127	9
ReplicatedWorkers	Deadlock	6	14	50	304	25

Variables

- Independent variables:
  - RMC and TMC (default mode of JPF)

Variables

- Independent variables:
  - RMC and TMC (default mode of JPF)
- Dependent variables:
  - execution time (ET_{TMC}, ET_{RMC})
  - memory usage (MU_{TMC}, MU_{RMC})

Obtaining Versions - Choices

1. Actual versions
2. Versions created by people with enough related experience to modify the program
3. Versions created by simulating changes at beginnings of methods

Threats to Validity

- External Validity
  - representativeness of object programs
  - representativeness of versions
- Internal Validity
  - faults in implementation of algorithms
  - faults in tools
- Construct Validity
  - metrics we have chosen

Time / Memory Cost Ratios

Program	TMC / RMC-Cri	TMC / RMC
Daisy	5.0 3.0	0.97 1.09
Elevator	43.4 1.6	1.57 0.36
AlarmClock	1.2 1.3	0.99 1.14
RaxExtnded	79.6 2.1	1.56 0.75
ReplWorkers	43.5 1.0	1.47 0.24

Time / Memory Cost Ratios

Program	TMC / RMC-Cri	TMC / RMC
Daisy	5.0 3.0	0.97 1.09
Elevator	43.4 1.6	1.57 0.36
AlarmClock	1.2 1.3	0.99 1.14
RaxExtnded	79.6 2.1	1.56 0.75
ReplWorkers	43.5 1.1	1.47 0.24

RMC-Cri consistently faster than TMC; in three cases substantially

TMC uses between 1.1 and 3.0 times more memory than RMC-Cri

Time / Memory Cost Ratios

Program	TMC / RMC-Cri	TMC / RMC
Daisy	5.0 3.0	0.97 1.09
Elevator	43.4 1.6	1.57 0.36
AlarmClock	1.2 1.3	0.99 1.14
RaxExtnded	79.6 2.1	1.56 0.75
ReplWorkers	43.5 1.0	1.47 0.24

RMC is faster than TMC in 3 of 5 cases, and not much slower in 2 others

RMC uses more memory than TMC in 3 of 5 cases, and a bit less in 2 others

Summary

- In the critical period, RMC requires less time than TMC, and less memory
- Savings increase with costs of analysis
- Overall, RMC can also provide savings in time compared to TMC, but it does this through increased memory usage

Overview of Presentation

- Evolving software
- Regression test selection
  - Dejavu algorithm
  - Analytical and empirical evaluation
- Regression model checking
  - Algorithm
  - Empirical evaluation
- Ongoing and Future Work

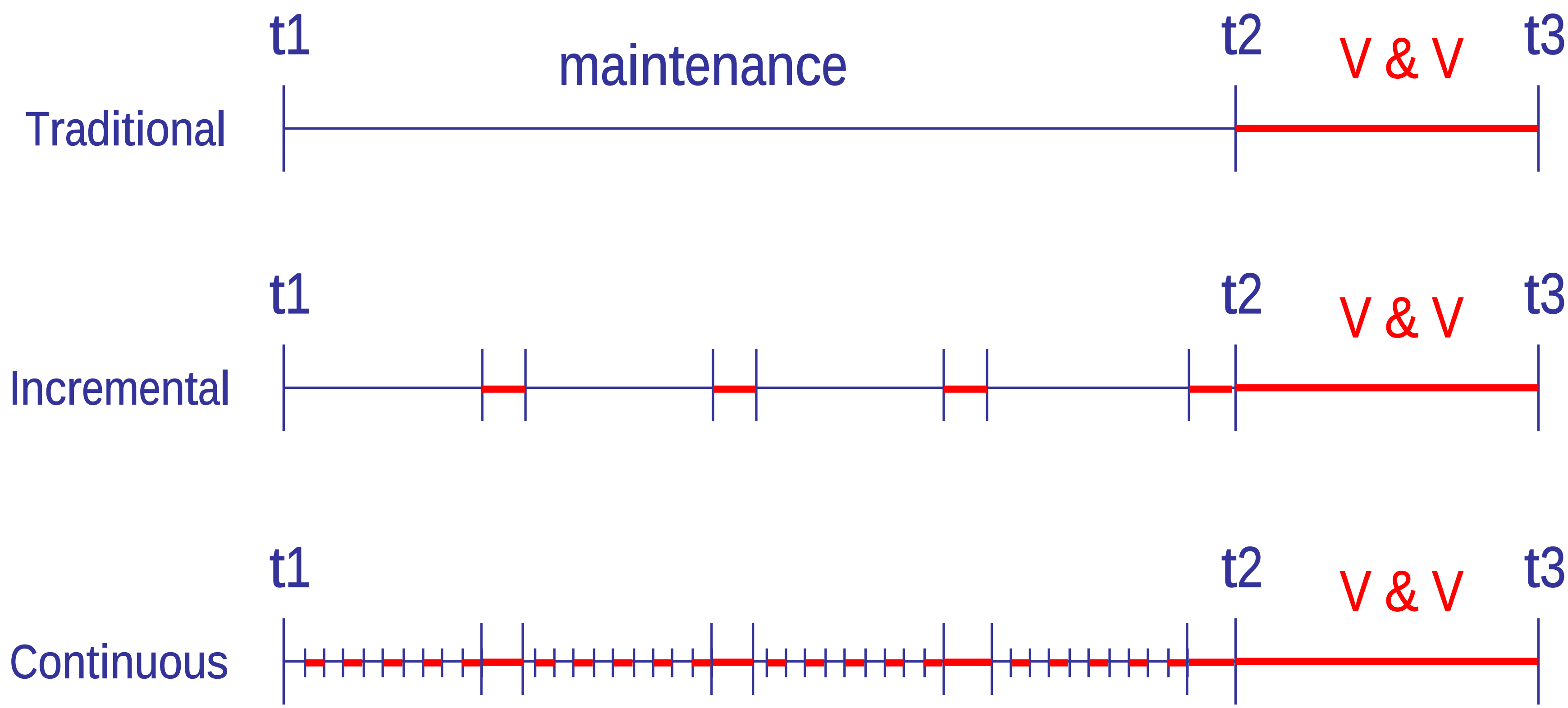
Ongoing and Future Work: Regression Testing

- Test case prioritization and test augmentation
- Economic models for evaluating costs/benefits
- Configurable systems and product lines
- Regression testing real-time embedded systems
- Testing under different process models

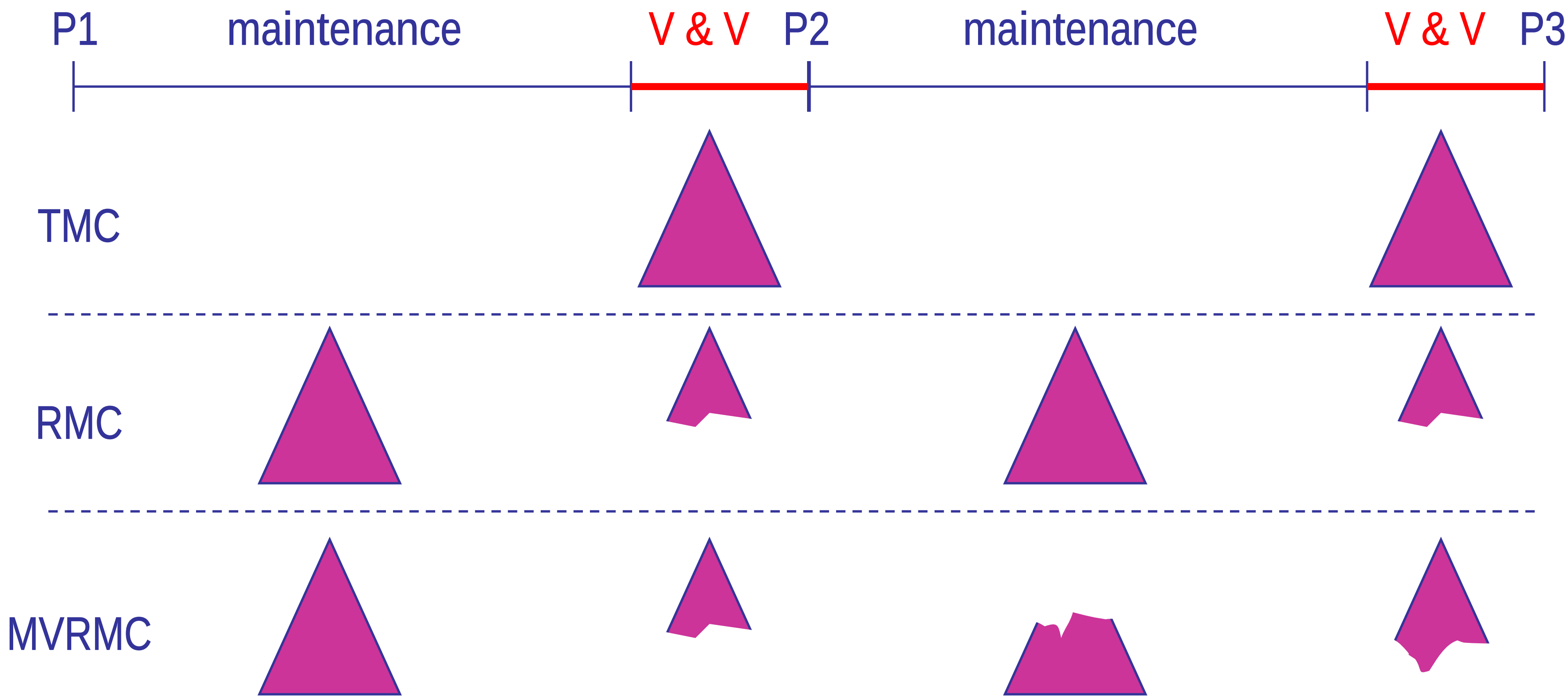
Ongoing and Future Work: Regression Model Checking

- Reduce memory costs by using coarser coverage elements or dominator analysis
- RMC is sensitive to the structure of program states; sub-setting program states may help
- Systems move through successions of releases; RMC ignores this, a multi-version RMC approach could be more cost-effective

More Process Models



Adapting to Other Processes



An Evolution-Centric Perspective on Software Testing

- Focus on evolution first
- Harness evolution
- Design for incremental validation

Acknowledgments

- Sponsors:
  - National Science Foundation
  - Boeing Commercial Airplane Group
  - Microsoft
  - Lockheed Martin
- Graduate Students:
  - Hyunsook Do
  - Marc Fisher
  - Jung-Min Kim (U. Md.)
  - Jim Law
  - Alexey Malishevsky
  - Guowei Yang
- Colleagues:
  - Matthew Dwyer
  - Sebastian Elbaum
  - Mary Jean Harrold
  - Sarfraz Kurshid
  - Alex Orso
  - Adam Porter
  - David Rosenblum

Software Verification: An Evolution-Centric Perspective

Gregg Rothermel

Dept. of Computer Science and Engineering
University of Nebraska - Lincoln

Supported by the National Science Foundation,
Microsoft, Lockheed Martin,
and Boeing Commercial Aircraft Group