Checking Inside the Black Box: Regression Testing Based on Value Spectra Differences

Tao Xie David Notkin

Dept. of Computer Science & Engineering, University of Washington, Seattle

12 Sept. 2004

ICSM 2004, Chicago

Synopsis

- Context:

  Traditional regression testing strongly focuses on black-box comparison of program outputs

- Problem:

  Behavior deviations (behavior differences) might be difficult to be propagated to observable outputs

- Goal:

  Detect and understand behavior deviations inside the black box

Approaches

Existing approaches:

- Fault propagation models [Thompson et al. 93, Voas 92]
- Structural program spectra, e.g. branch, path
  [Ball&Larus 96, Reps et al. 97, Harrold et al. 00]

Our new approach:

- Compute value spectra from a program execution
  - characterize program states of user functions
- Compare value spectra from two versions
  - detect and understand deviation propagation

Outline

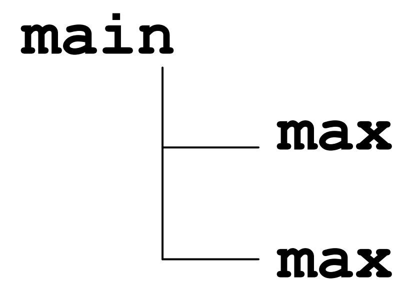
- Value Spectra
- Value Spectra Comparison
- Experiment
- Conclusion

Example Program

```
int main(int argc, char *argv[]) {
    int i, j;
    if (argc != 3) {
        printf("Wrong arguments!");
        return 1;
    }
    i = atoi(argv[1]);
    j = atoi(argv[2]);
    if (max(i, j) >= 0){
        if (max(i, j) == 0){
            printf("0");
        } else {
            printf("1");
        }
    } else {
        printf("-1");
    }
    return 0;
}
```

Example Program

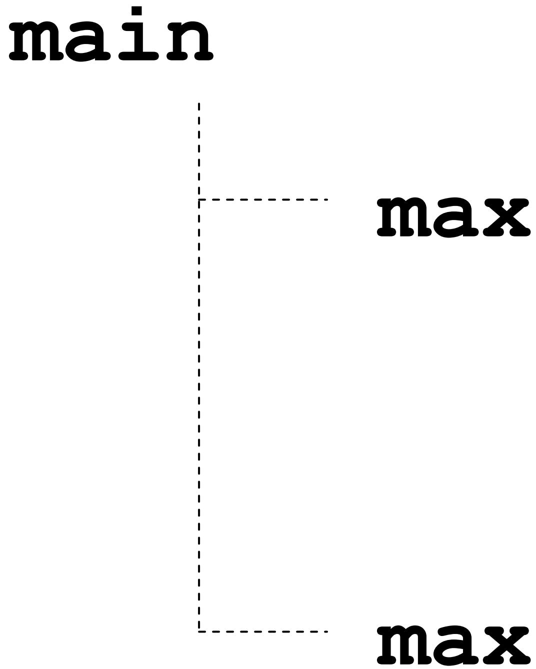
```
int main(int argc, char *argv[]) {
    int i, j;
    if (argc != 3) {
        printf("Wrong arguments!");
        return 1;
    }
    i = atoi(argv[1]);
    j = atoi(argv[2]);
    if (max(i, j) >= 0){
        if (max(i, j) == 0){
            printf("0");
        } else {
            printf("1");
        }
    } else {
        printf("-1");
    }
    return 0;
}
```



Program black-box input "0 1"

Program black-box output "1"

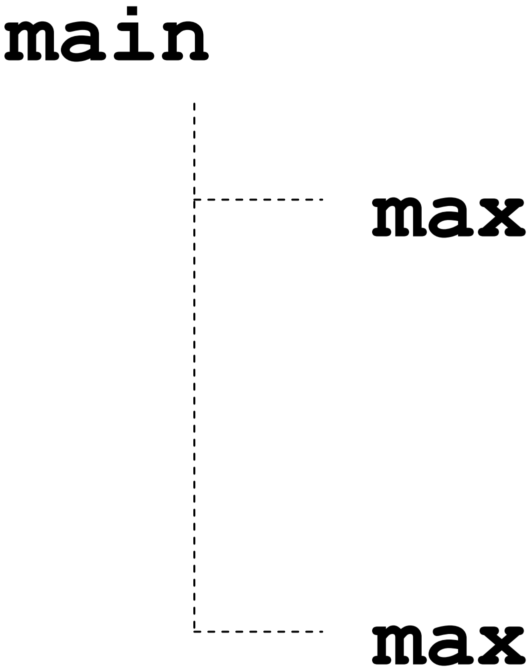
Dynamic Call Tree



Dynamic Call Tree

***Main*-entry state**

argc	argv[1]	argv[2]
3	“0”	“1”



Dynamic Call Tree

***Main*-entry state**

argc	argv[1]	argv[2]
3	"0"	"1"

```
main
  |
  |---- max
  |
  |
  |---- max
```

***Main*-exit state**

argc	argv[1]	argv[2]	ret
3	"0"	"1"	0

Dynamic Call Tree

Main-entry state

argc	argv[1]	argv[2]
3	"0"	"1"

`main`

`max`

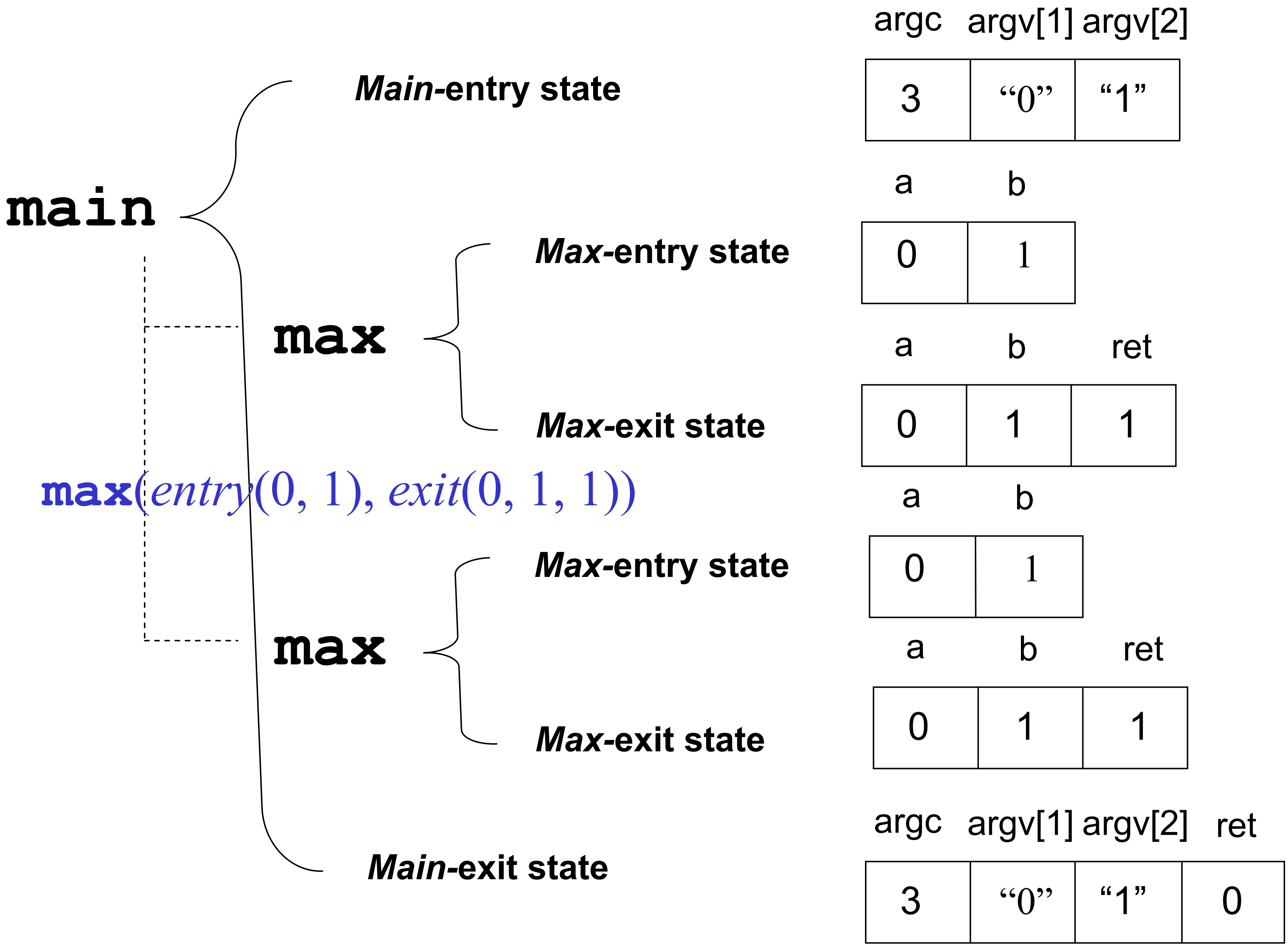
`main`(*entry*(3, "0", "1"), *exit*(3, "0", "1", 0))

`max`

Main-exit state

argc	argv[1]	argv[2]	ret
3	"0"	"1"	0

Dynamic Call Tree



Value Spectra

- Value hit spectra
  - **`main`**(*entry*(3, “0”, “1”), *exit*(3, “0”, “1”, 0))
  - **`max`**(*entry*(0, 1), *exit*(0, 1, 1))

- Value count spectra
  - include additional count information

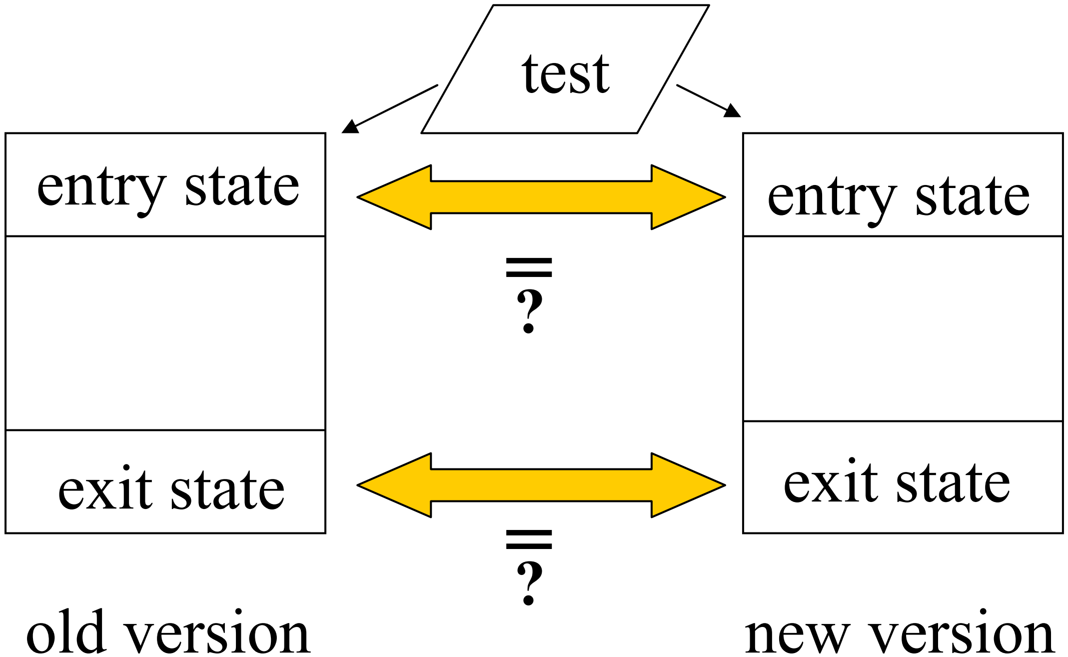
- Value trace spectra
  - include additional sequence order information

Outline

- Value Spectra
- Value Spectra Comparison
- Experiment
- Conclusion

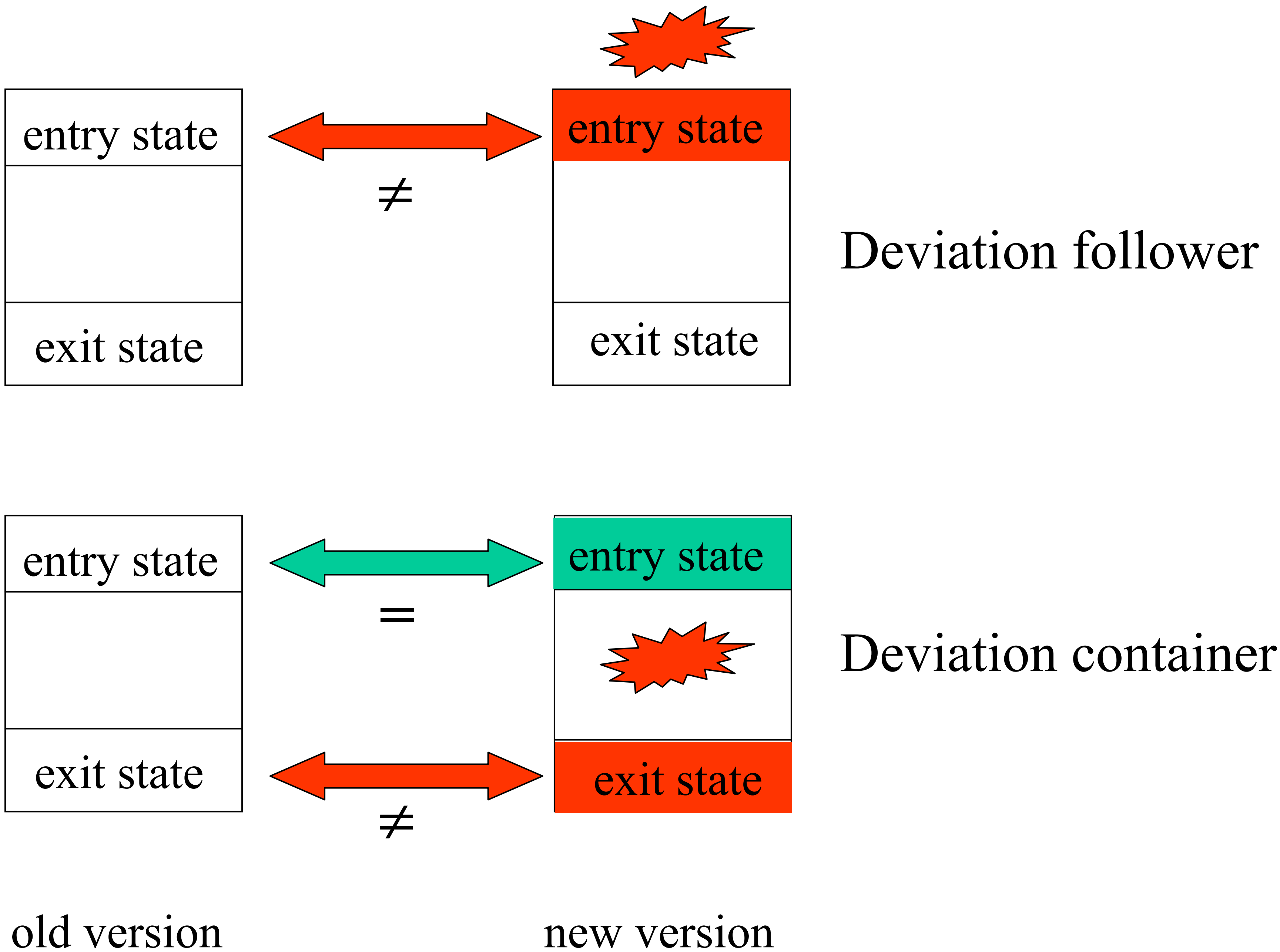
Spectra Comparison

- Function execution comparison



- State linearization to compare the values of pointer-type variables [Xie, Marinov, Notkin ASE 04]

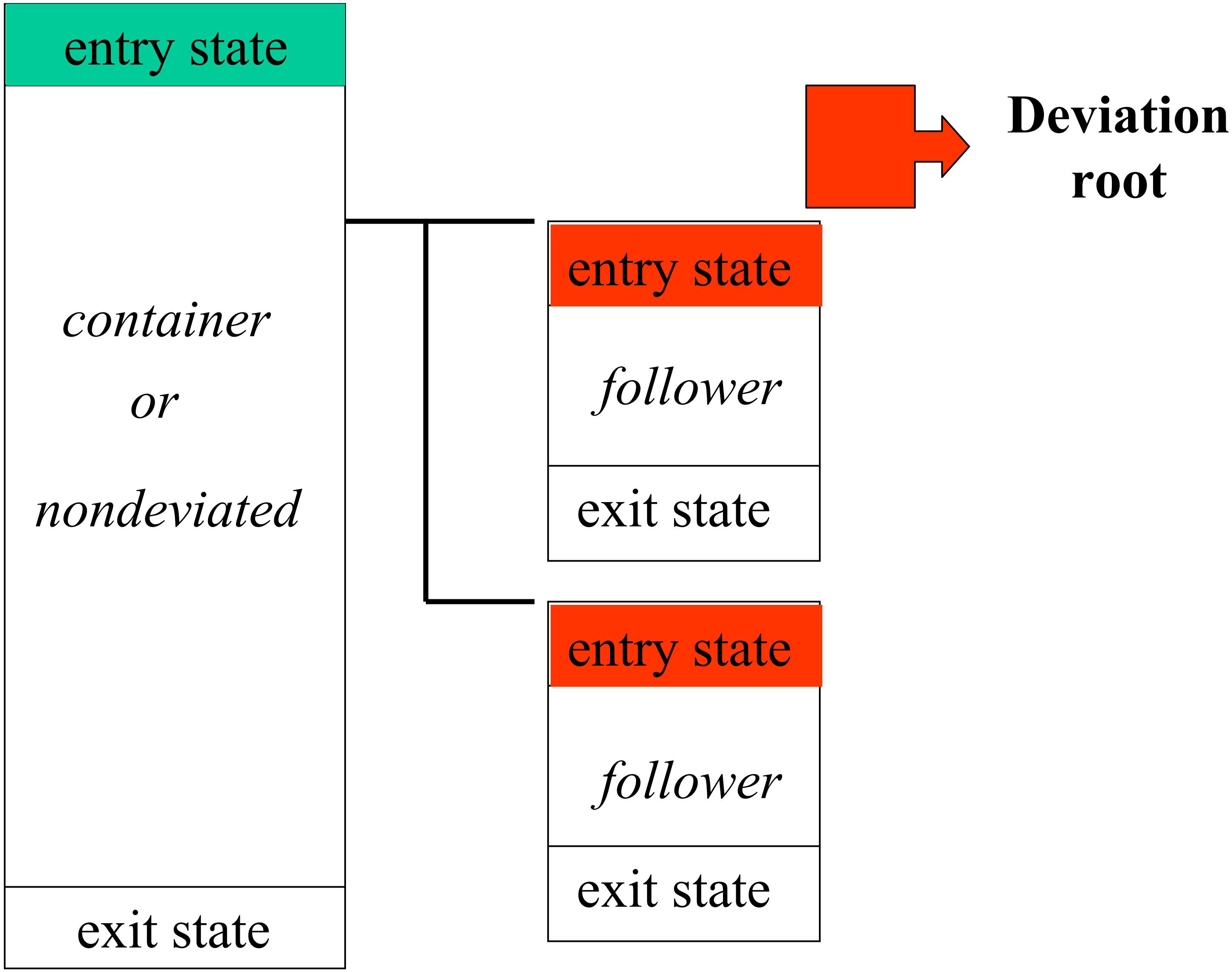
Understanding Deviations



Understanding Deviation Propagations

- Understand where the deviations start and how they are propagated.
- Deviation root
  - a program change that triggers specific behavior deviations
- Deviation-root localization
  - two heuristics

Heuristic 1: Earliest dev follower's preceded area



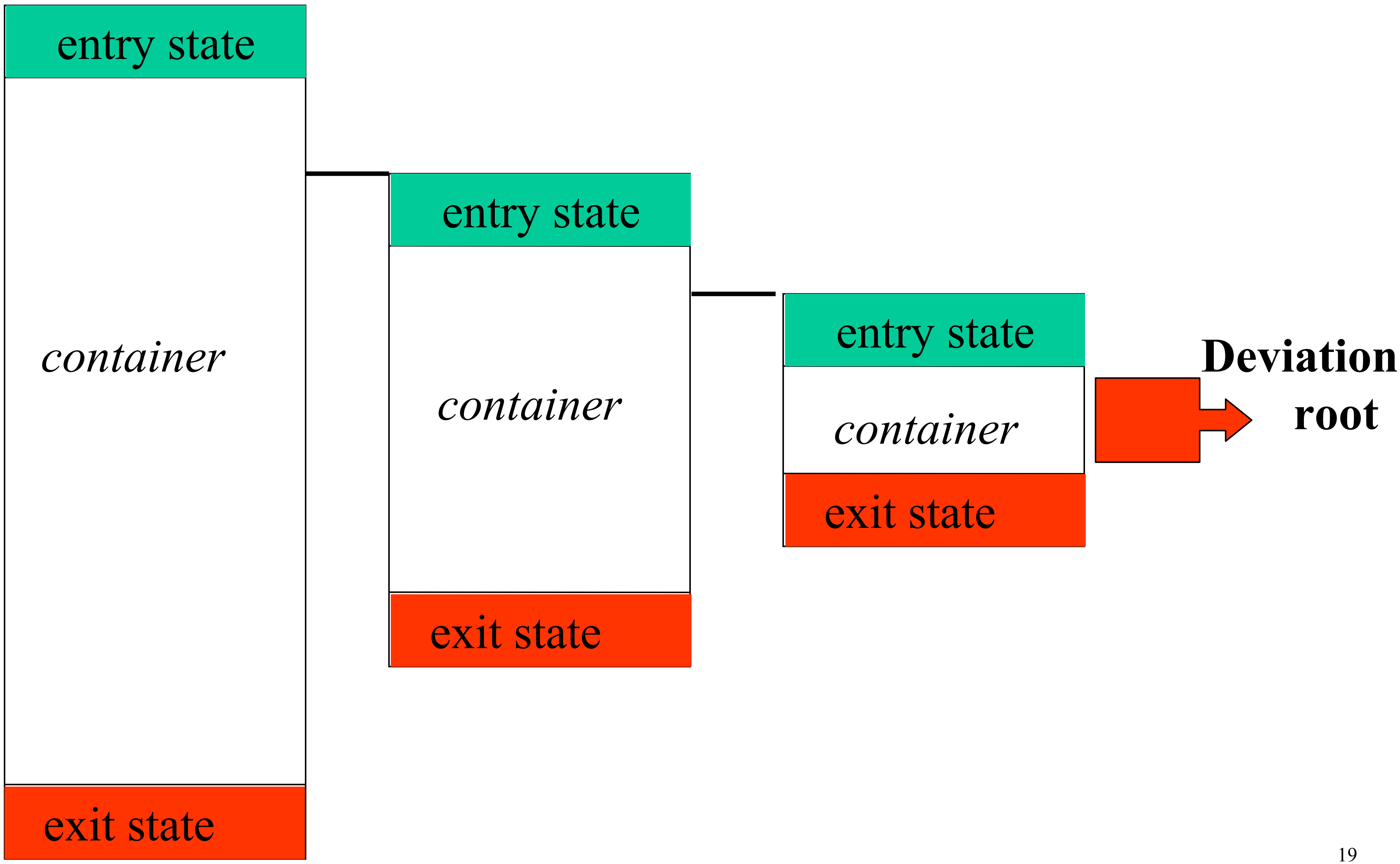
Deviation Follower Example

The 58th test on the 9th faulty version of tcas program

[Hutchins et al. 94]

```
main
 |___initialize
 |___alt_sep_test
      |___Non_Crossing_Biased_Climb
      |    |___Inhibit_Biased_Climb
      |    |___Own_Above_Threat
      |___Non_Crossing_Biased_Descend
      |    |___Inhibit_Biased_Climb
      |    |___Own_Below_Threat--------[dev follower]
      |    |___ALIM--------------------[dev follower]
      |___Own_Above_Threat
```

Heuristic 2: Innermost dev container's enclosed area



Deviation Container Example

The 91st test on the 9th faulty version of tcas program

[Hutchins et al. 94]

```
main
 |
 |___initialize
 |___alt_sep_test-------------------[dev container]
      |___Non_Crossing_Biased_Climb
      |    |___Inhibit_Biased_Climb
      |    |___Own_Above_Threat
      |    |___ALIM
      |___Own_Below_Threat
      |___Non_Crossing_Biased_Descend-[dev container]
           |___Inhibit_Biased_Climb
           |___Own_Below_Threat
```

Outline

- Value Spectra
- Value Spectra Comparison
- Experiment
- Conclusion

Experimental Subjects

[Hutchins et al. 94, Rothermel&Harrold 98]

program	*funcs*	*loc*	*tests*	*vers_used*
printtok	18	402	4130	7
printtok2	19	483	4115	10
replace	21	516	5542	12
schedule	18	299	2650	9
schedule2	16	297	2710	10
tcas	9	138	1608	9
totinfo	7	346	1052	6

Questions to Be Answered

- How effectively can we use the value spectra differences to expose deviations (comparing to using output differences)?

- How accurately can we use the two heuristics to locate deviation roots?

Experiment Design

- Run each test on both the original version and a faulty version (instrumented using Daikon frontend [Ernst et al. 01])
- Compute value spectra of two versions
- Compare value spectra of two versions
  [compare outputs of two versions]
- Locate deviation roots from spectra differences

Measurements

- Deviation exposure ratio:

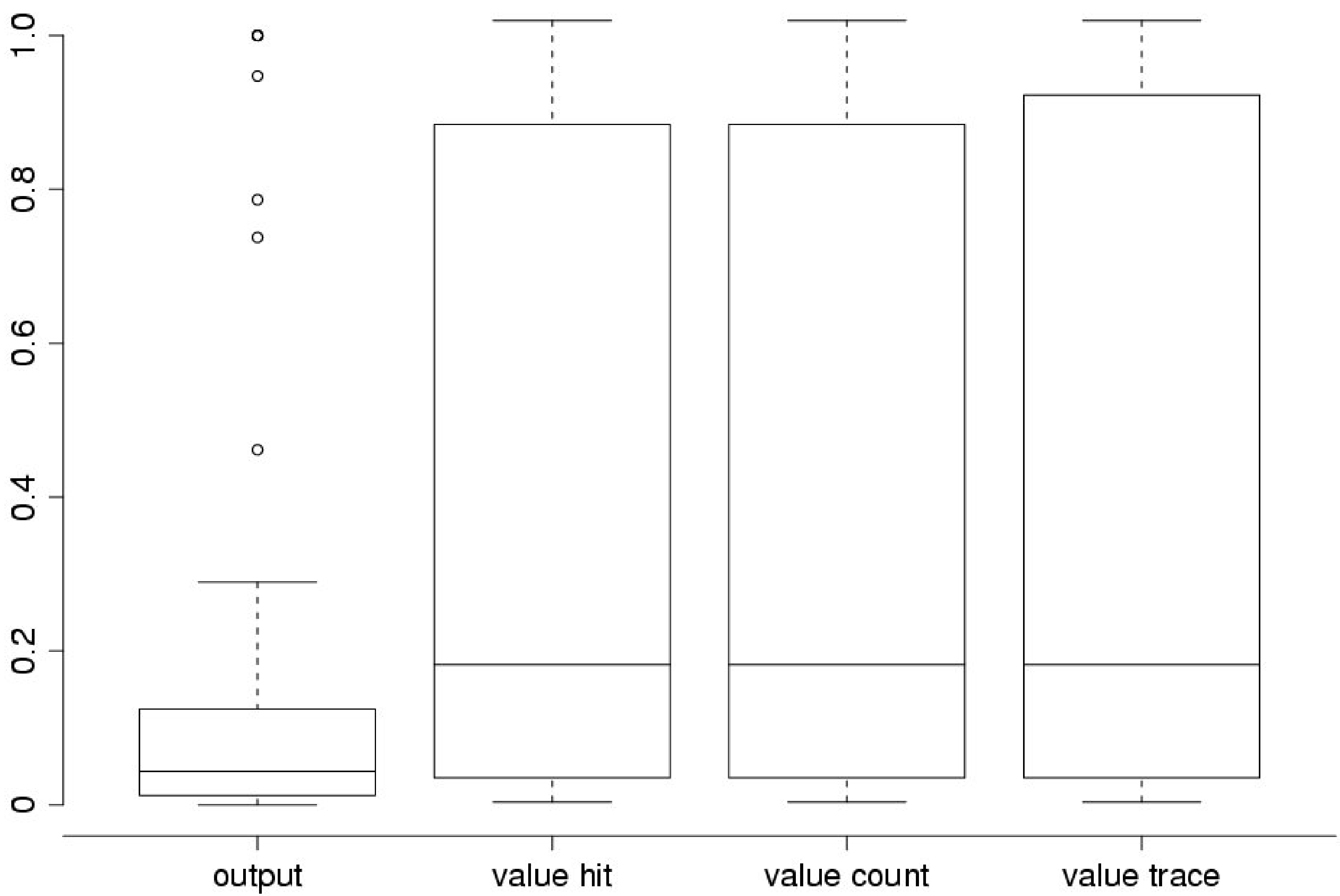
$$\frac{|\text{tests exhibiting spectra diffs}|}{|\text{tests covering the changed lines}|}$$

- Deviation-root localization ratio:

$$\frac{|\text{tests succeeding in locating roots}|}{|\text{tests exhibiting spectra diffs}|}$$

- The higher, the better

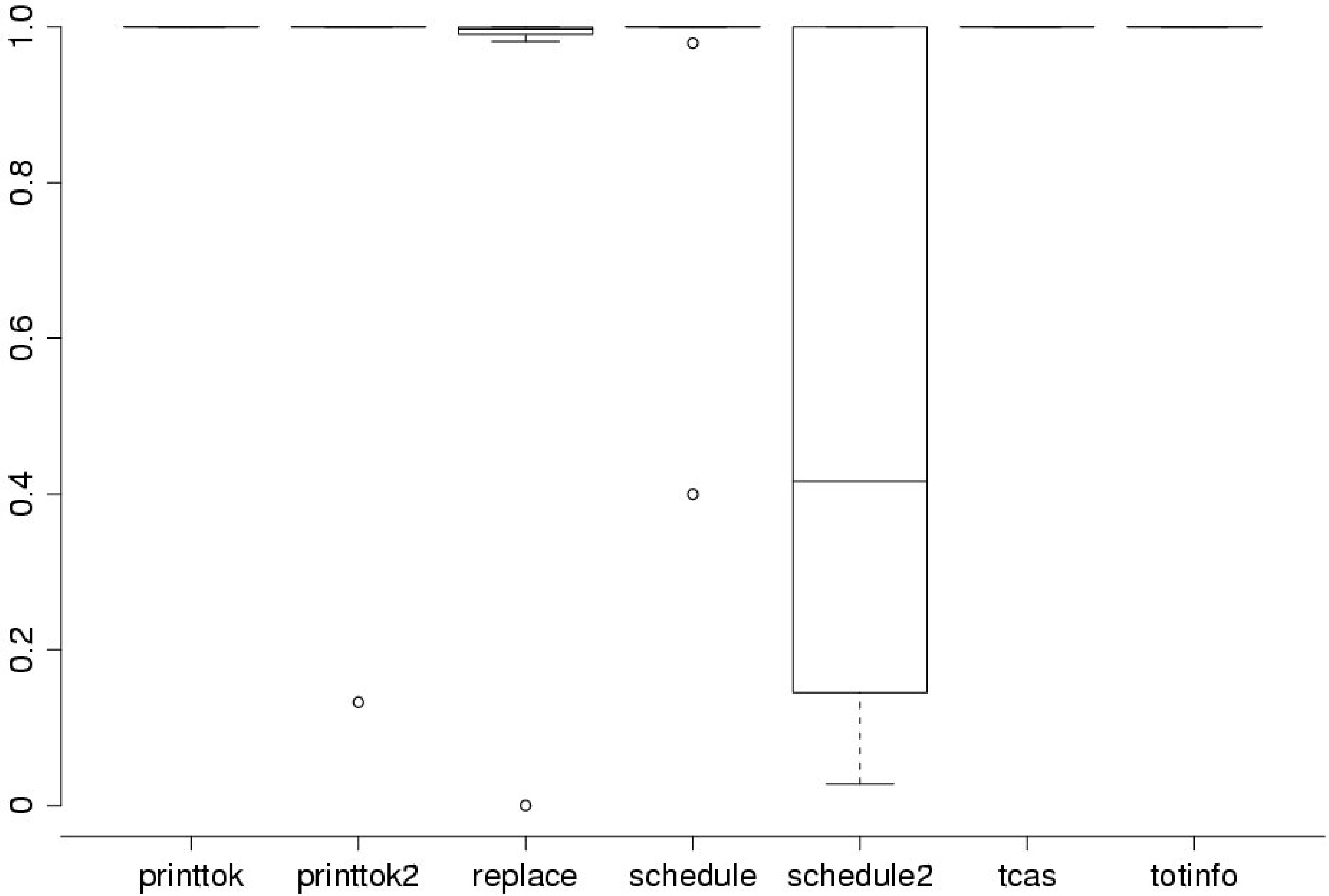
Deviation Exposure Ratios



What We Have Learned

- When program outputs are the same for two versions, deviations can be still detected based on value spectra differences.
- Value hit spectra seem to be good enough; adding count information or sequence information does not improve much.

Deviation-Root Localization Ratios
(Value Hit Spectra)



What We Have Learned

- Identified deviation roots are accurate for most programs.

- The exceptional case is *schedule2*, whose state changes lie in deep parts of a linked list.
  - By default, Daikon frontend looks into state information of complex data structures with depth of three

Threats to Validity

- Representative of true practice?
  - Subject programs, faults, and tests
- Instrumentation effects that bias the results
  - Faults on tools (analysis scripts, Daikon frontend)
- Use of approximate state information

Conclusion

- Checking only black-box outputs is limited in regression testing
- Value spectra enrich the existing program spectra family
- Comparing value spectra helps detect and understand deviation propagation
- The experimental results have shown
  - Comparing value spectra is effective in detecting deviations
  - Two heuristics are effective in locating deviation roots

Questions?

Scalability

- Cost = $O(|vars| \times |userfuncs| \times |testsuite|)$

program	*funcs*	*loc*	*tests*	*trace size/test (kb)*
printtok	18	402	4130	36
printtok2	19	483	4115	50
replace	21	516	5542	71
schedule	18	299	2650	982
schedule2	16	297	2710	272
tcas	9	138	1608	8
totinfo	7	346	1052	27

- **Execution slowdown: (2 ~7)** *schedule2*: **31;** *schedule*: **48**
- **Analysis time: (7 ~ 30 ms/test)** *schedule2*: **137;** *schedule*: **201 ms/test**

Related Work

- Structural program spectra [Ball&Larus 96, Reps et al. 97, Harrold et al. 00]
- GUI test oracles based on GUI states [Memon et al. 03]
- Relative debugging [Abramson et al. 96]
- Comparison checking [Jaramillo et al. 02]
- RELAY model [Thompson et al. 93]
- PIE (Propagation, Infection, and Execution) model [Voas 92]

Representation of Function Execution

- Function-entry state
  - Argument values
  - Global variable values
- Function-exit state
  - Updated argument values
  - Updated global variable values
  - Return value
- funcname (*entry*(argvals), *exit*(argvals, ret))

  `main`(*entry*(3, “0”, “1), *exit*(3, “0”, “1”, 0))

  `max`(*entry*(0, 1), *exit*(0, 1, 1))