Fault Localization in Model Checking

Implementation and Evaluation of Fault-Localization Techniques with Distance Metrics

Angelos Kafounis

Bachelor Thesis

Mentor:
Thomas Lemberger
Professor in charge:
Prof. Dr. Dirk Beyer

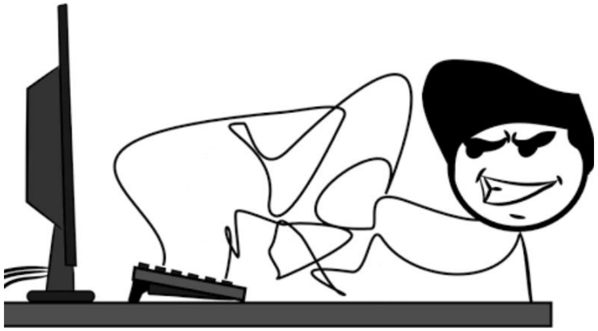
30.09.2020

Agenda

- Motivation
- Background
- Implementation
- Evaluation
- Future Work
- Conclusion

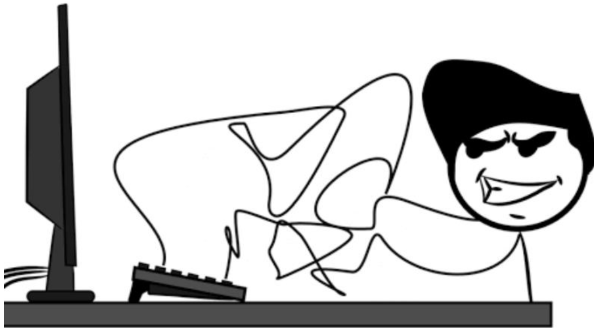
Motivation

Programming like a boss

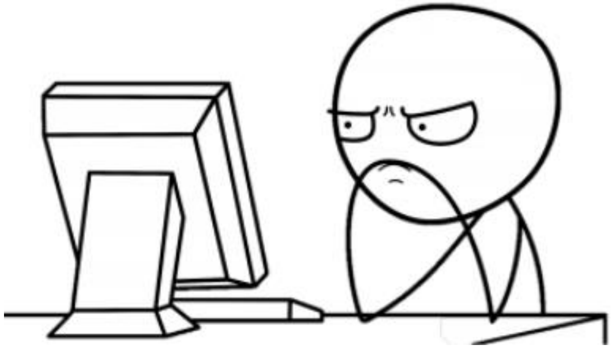


Motivation

Programming like a boss

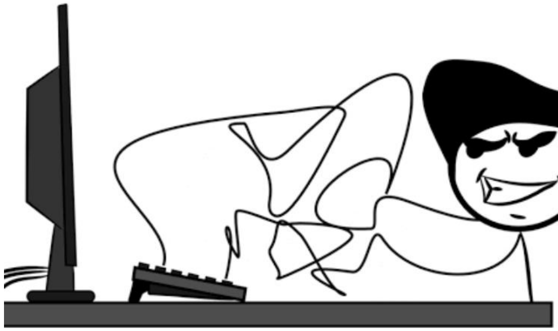


Suddenly something is not working

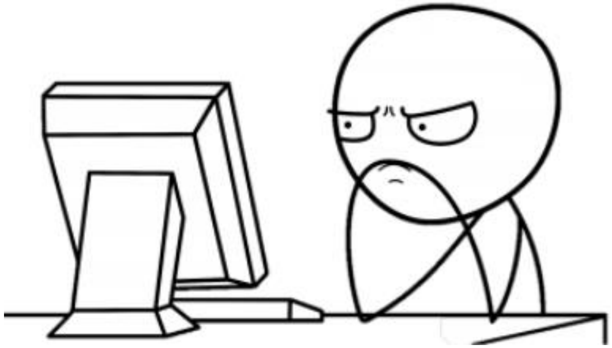


Motivation

Programming like a boss



Suddenly something is not working

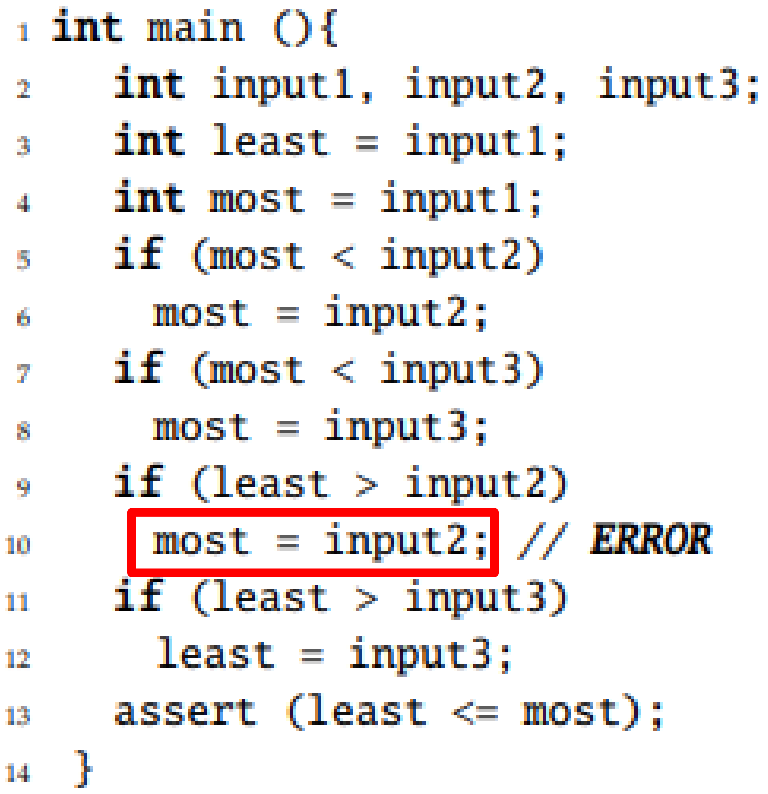


Looking for the fault

Motivation

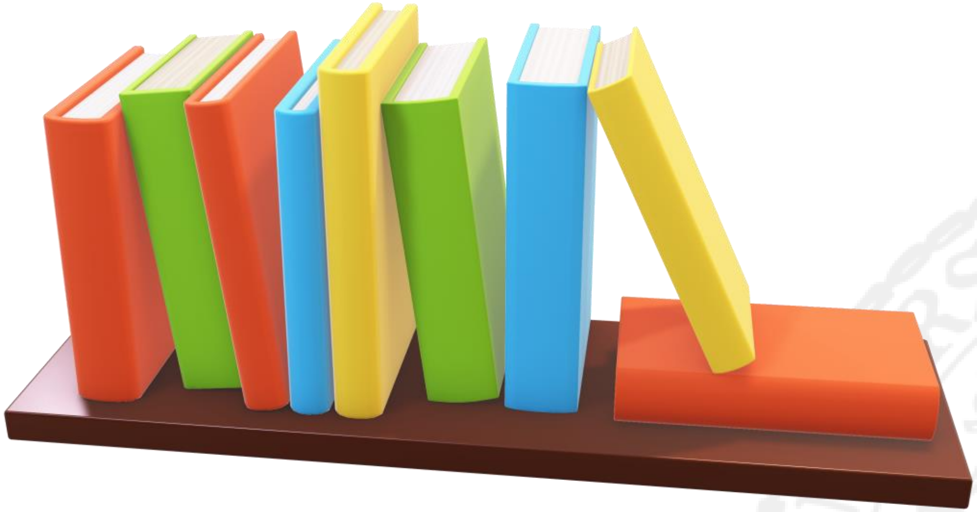
Goal of the Research

- Help developers locate the fault in a program that violates its specification, using distance metrics

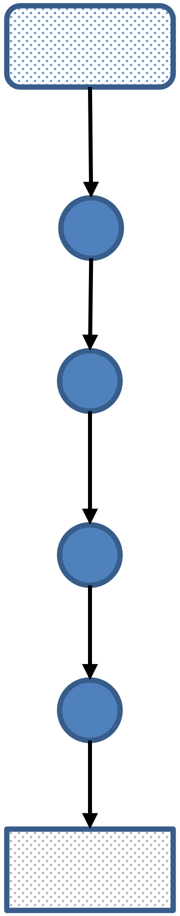


```
int main (){
   int input1, input2, input3;
   int least = input1;
   int most = input1;
   if (most < input2)
     most = input2;
   if (most < input3)
     most = input3;
   if (least > input2)
     most = input2; // ERROR
   if (least > input3)
     least = input3;
   assert (least <= most);
 }
```

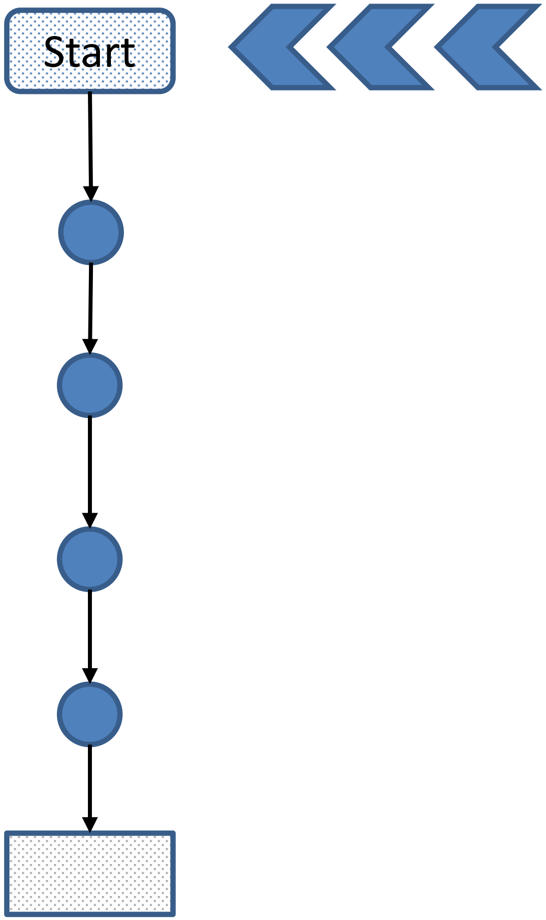
Background Knowledge



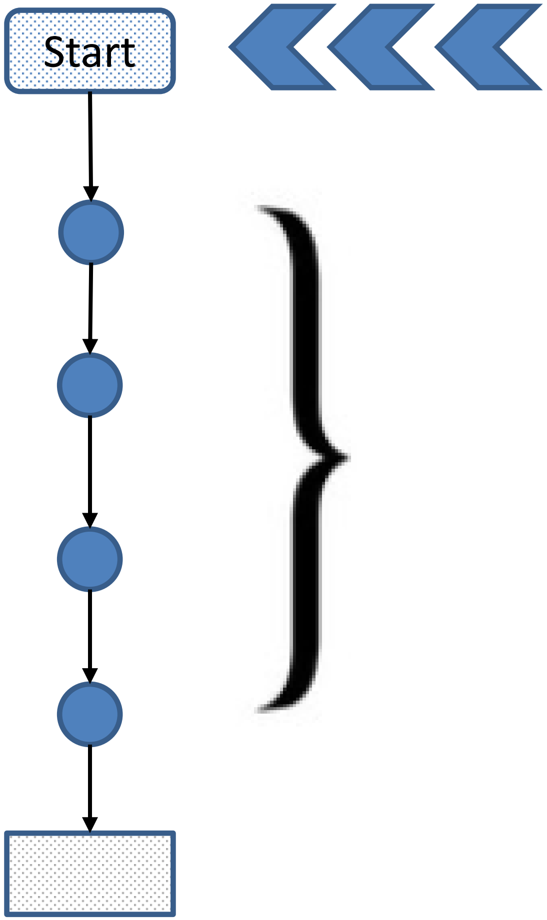
Parts of Program Execution



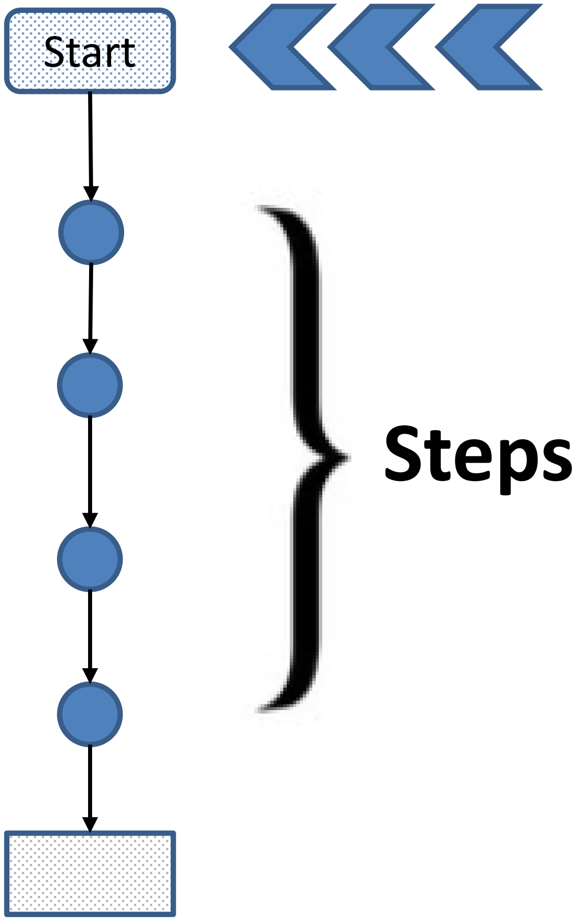
Parts of Program Execution



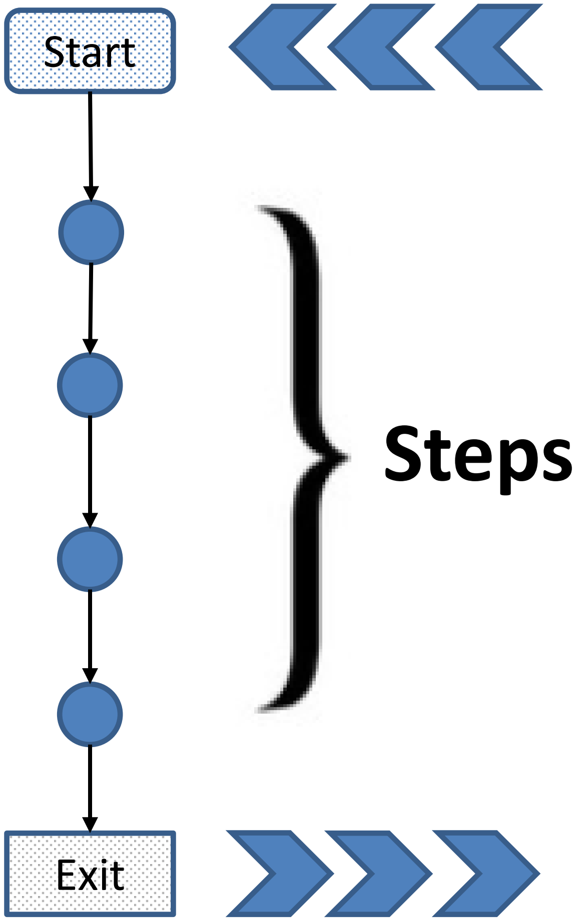
Parts of Program Execution



Parts of Program Execution



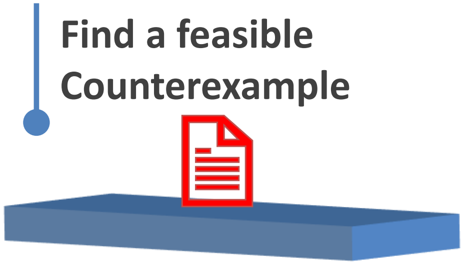
Parts of Program Execution



Distance Metrics for Program Executions

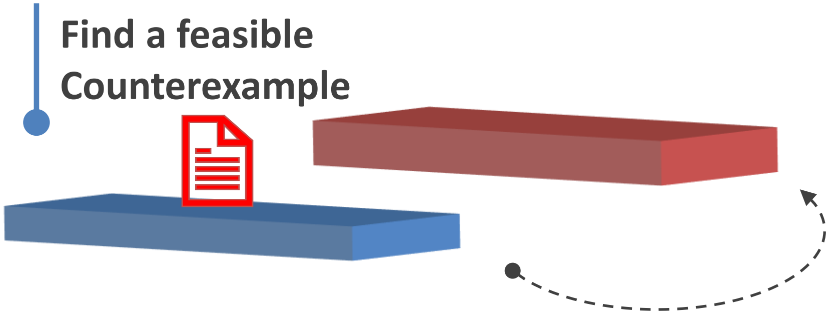
Distance Metrics for Program Executions

Distance Metrics for Program Executions

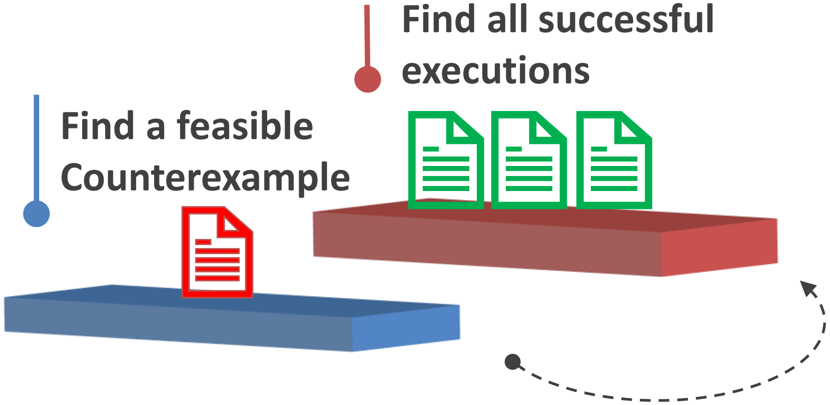


Find a feasible Counterexample

Distance Metrics for Program Executions


Find a feasible Counterexample

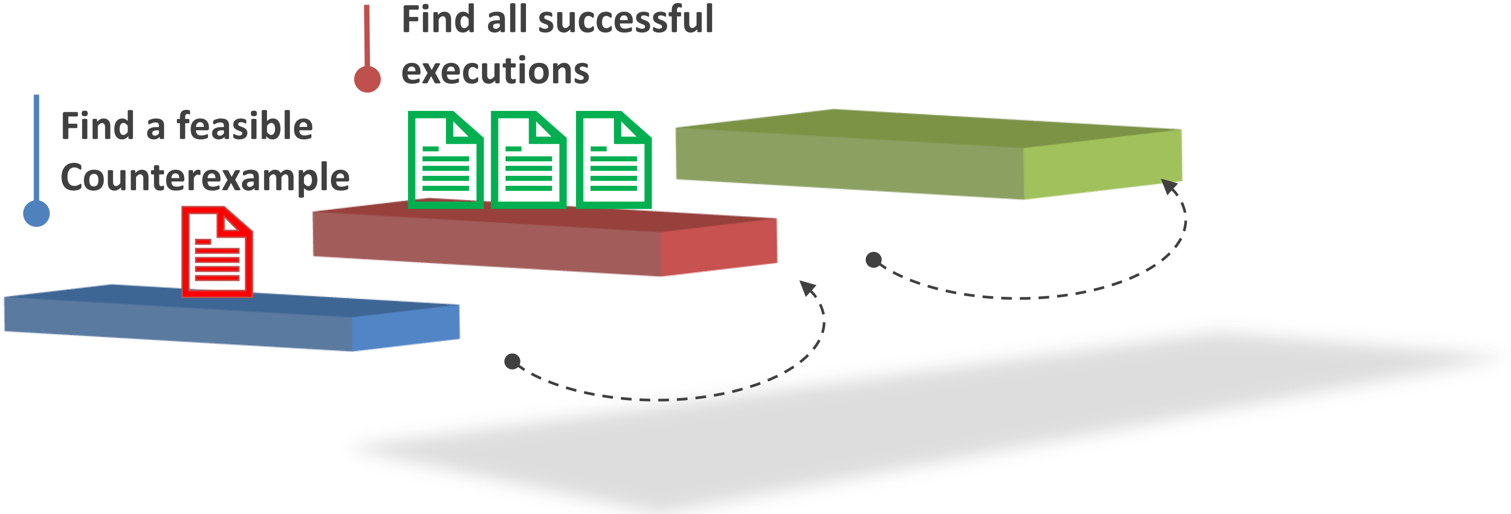
Distance Metrics for Program Executions



Find a feasible Counterexample

Find all successful executions

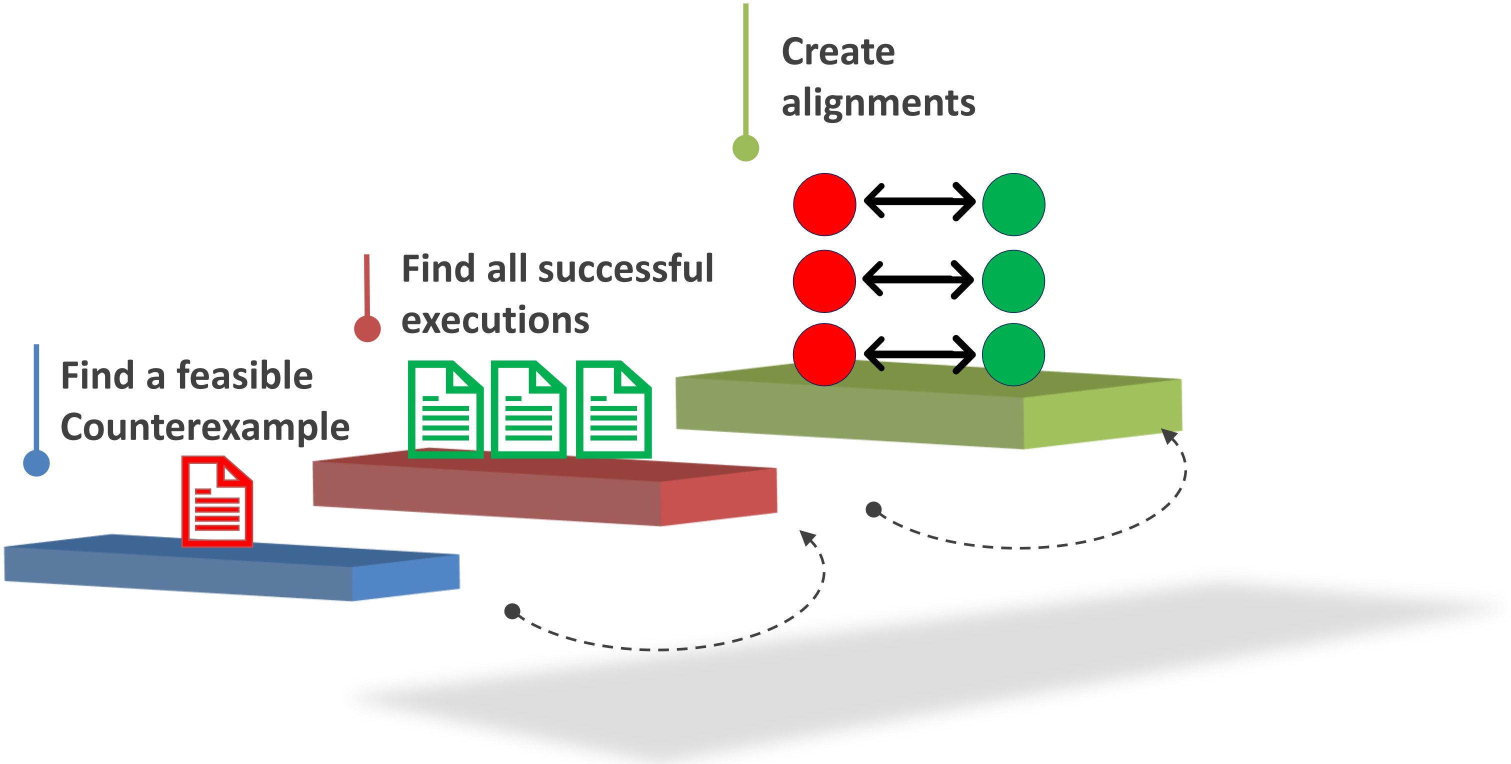
Distance Metrics for Program Executions



Find all successful executions

Find a feasible Counterexample

Distance Metrics for Program Executions

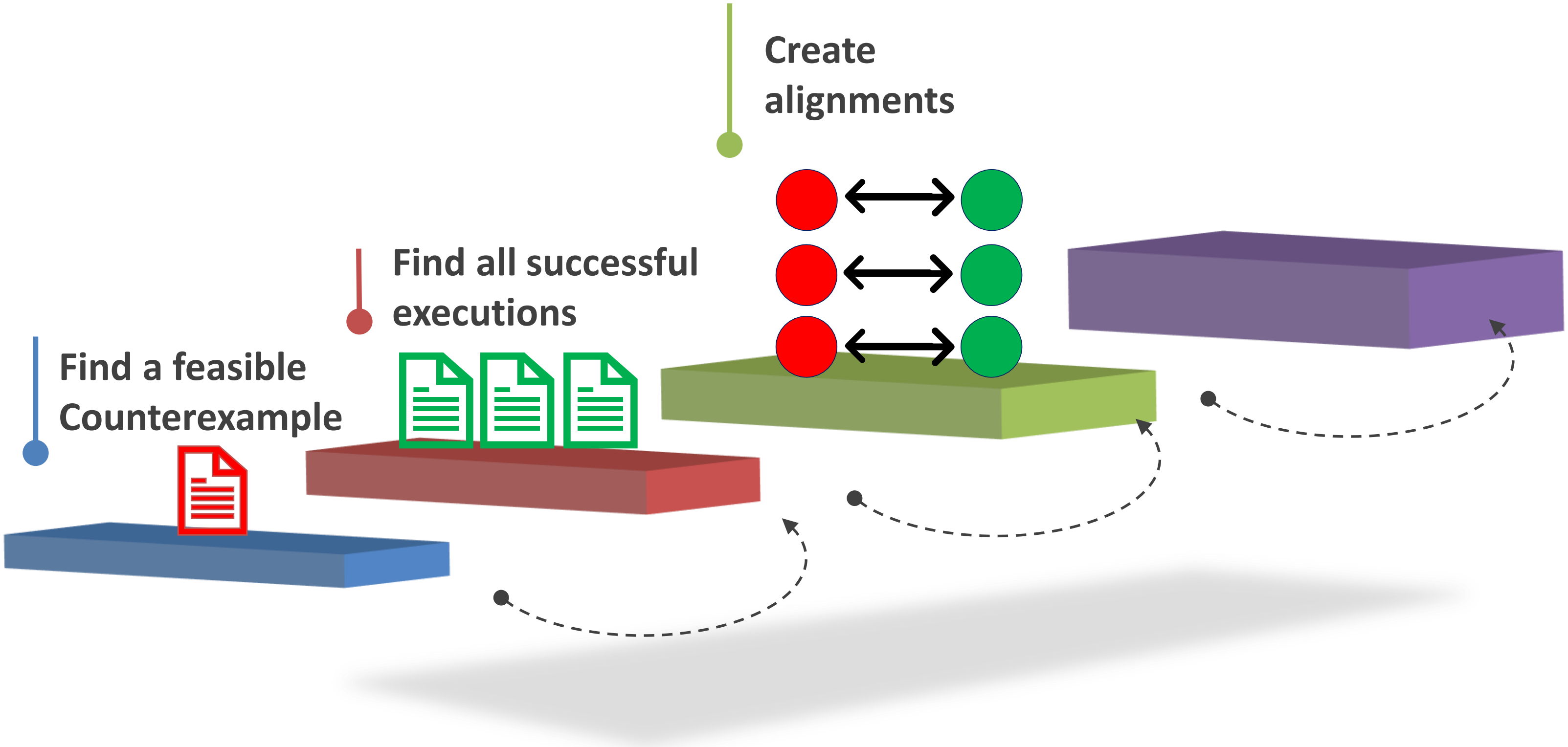


Create alignments

Find all successful executions

Find a feasible Counterexample

Distance Metrics for Program Executions

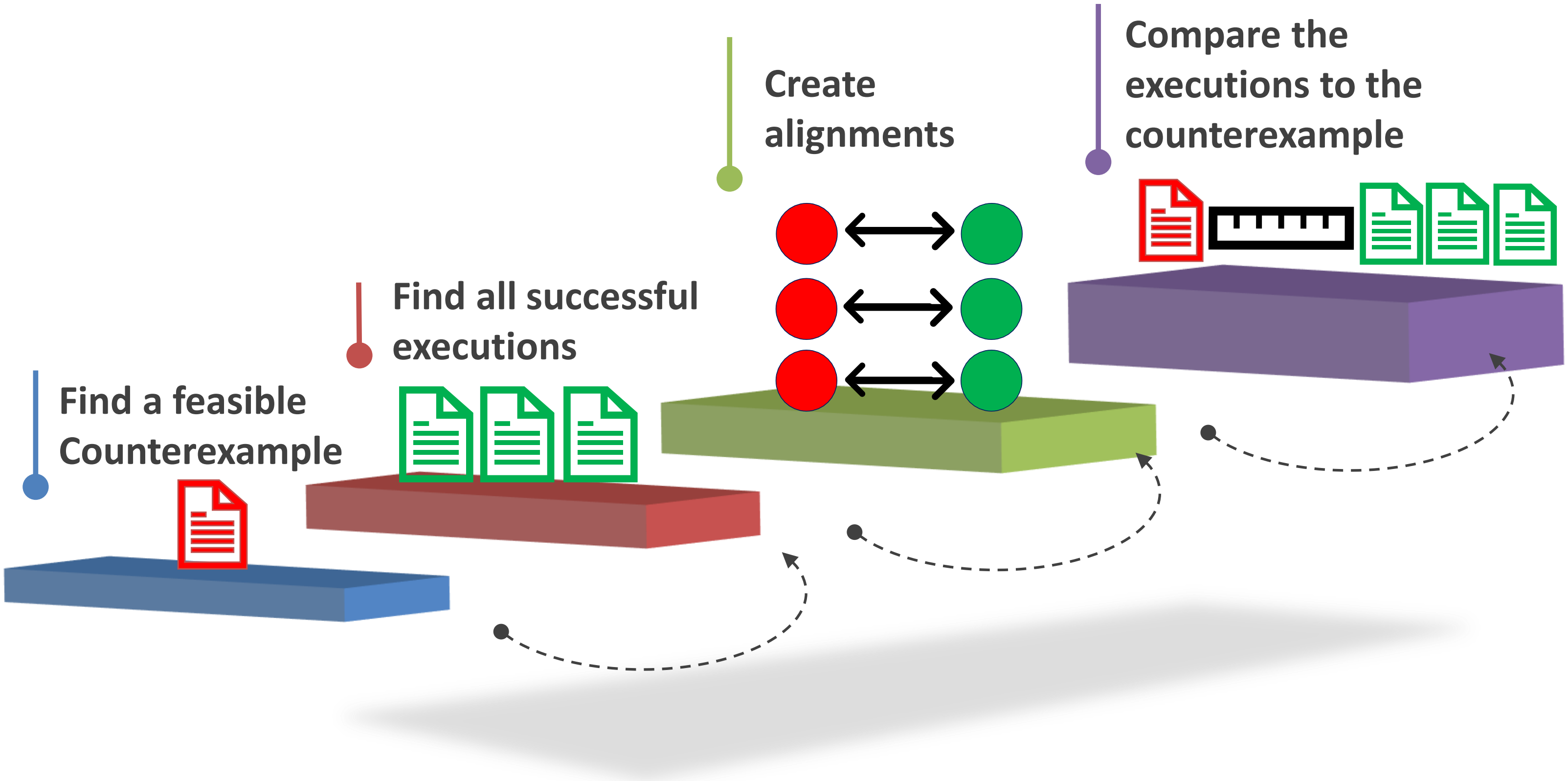


Find a feasible Counterexample

Find all successful executions

Create alignments

Distance Metrics for Program Executions



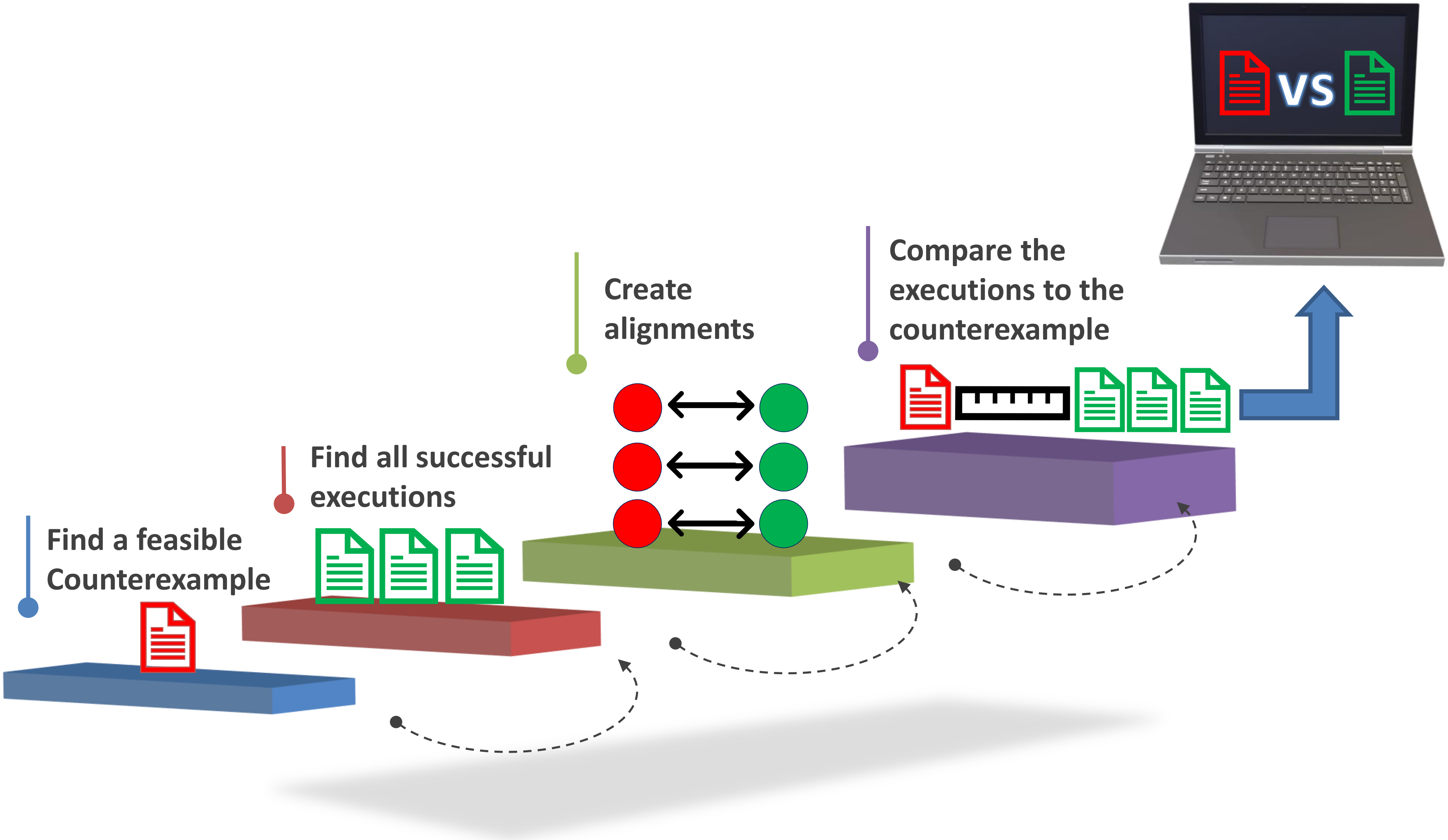
Find a feasible Counterexample

Find all successful executions

Create alignments

Compare the executions to the counterexample

Distance Metrics for Program Executions



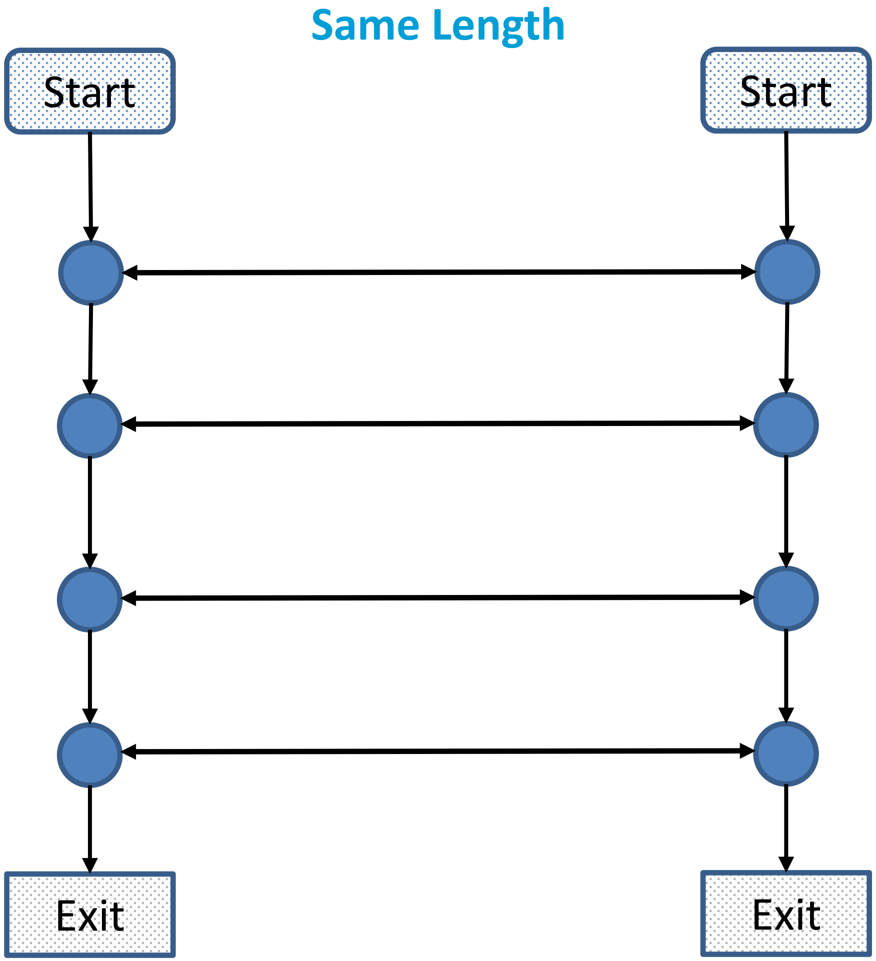
Find a feasible
Counterexample

Find all successful
executions

Create
alignments

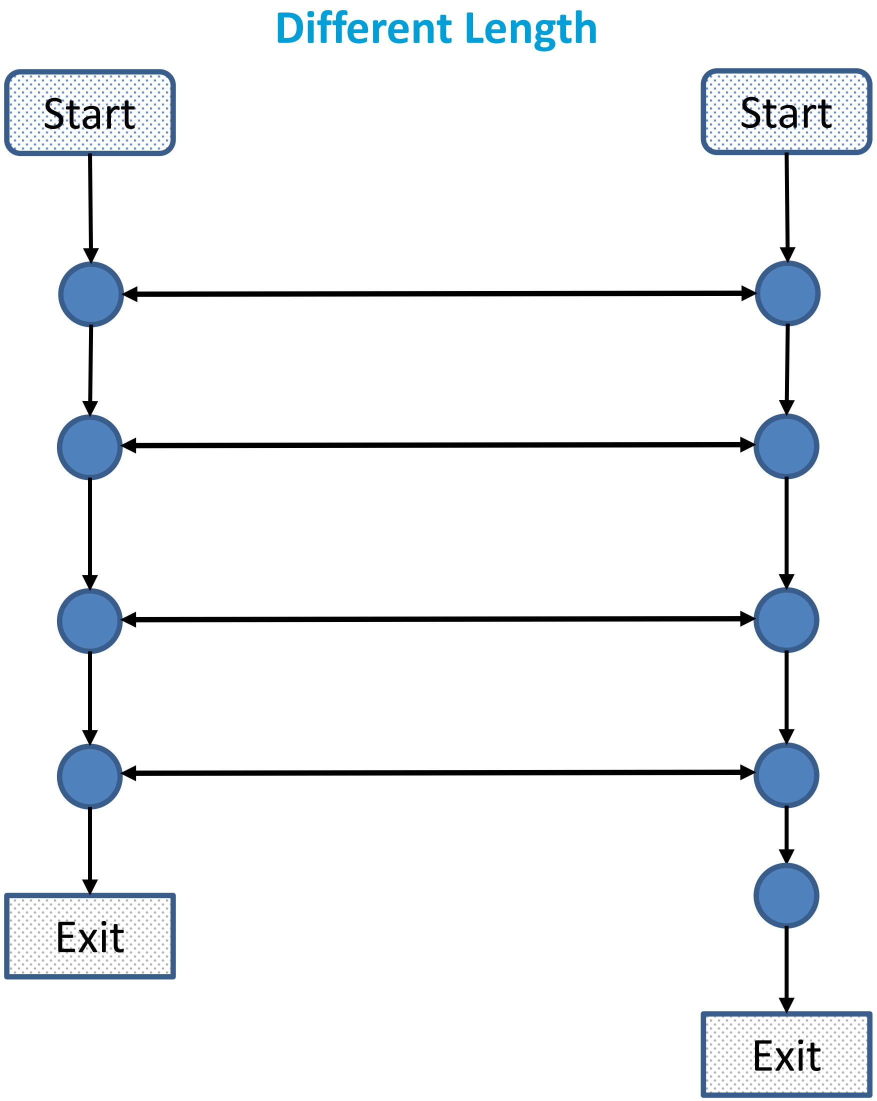
Compare the
executions to the
counterexample

Comparison of Program Executions

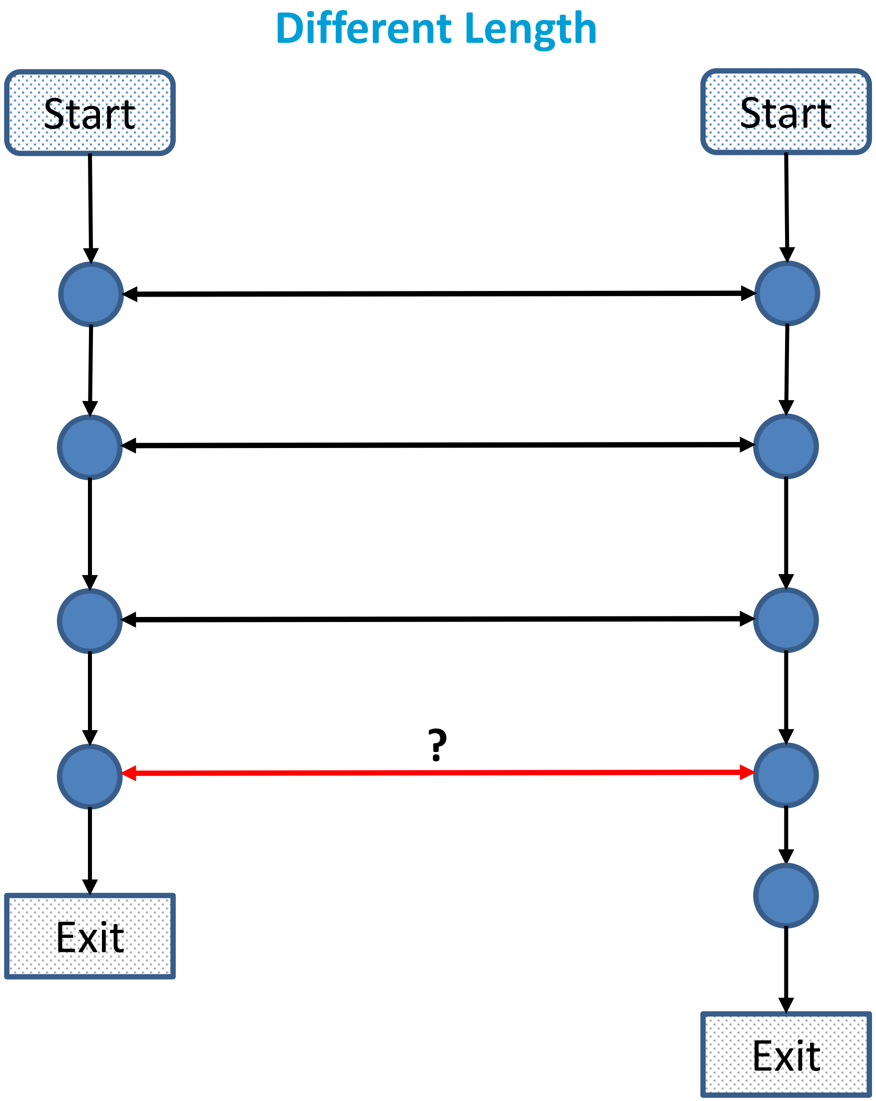


Same Length

Comparison of Program Executions

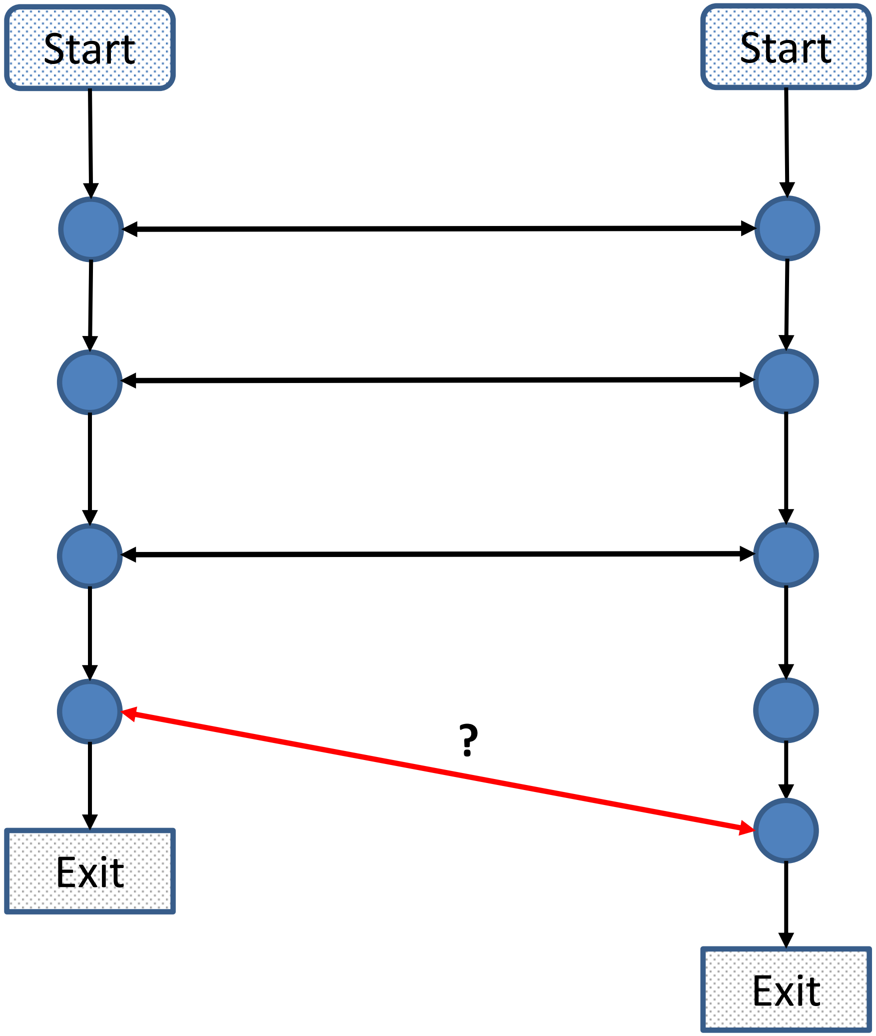

Different Length

Comparison of Program Executions

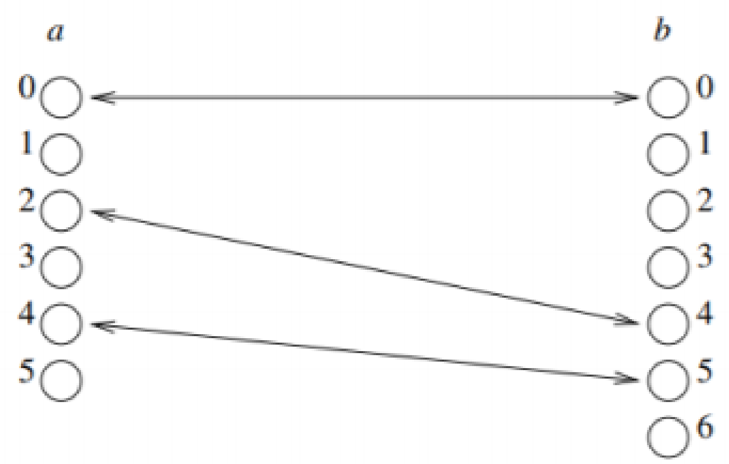


Comparison of Program Executions

Different Length

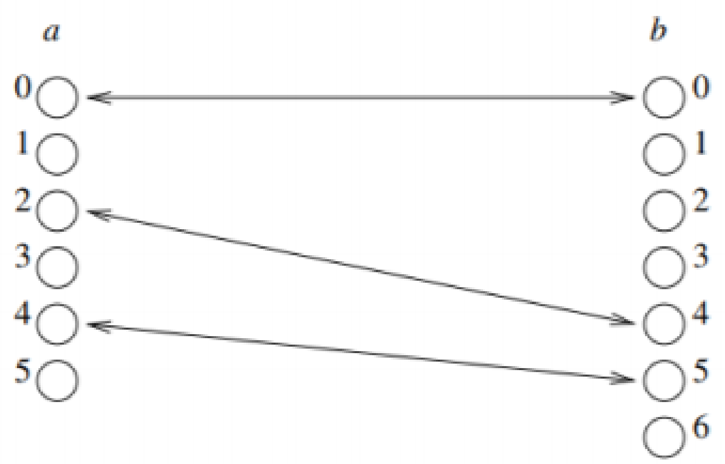


Alignments



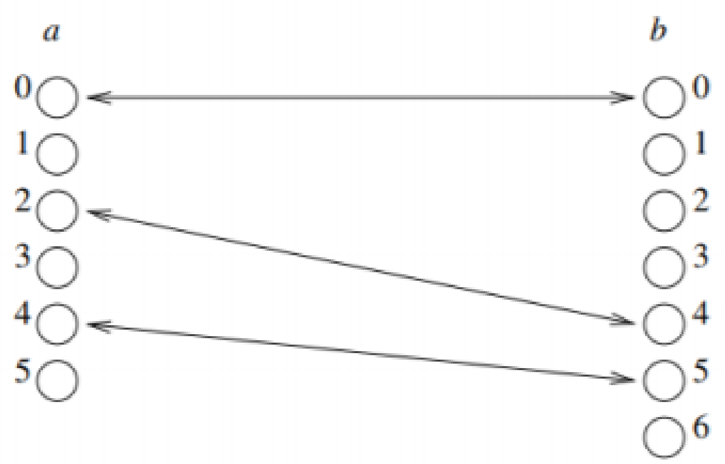
Program executions of different length

Alignments



Program executions of different length

Alignments



Program executions of different length

What step of the execution ‘a’

<u>vs</u>

What step of the execution ‘b’

Alignments

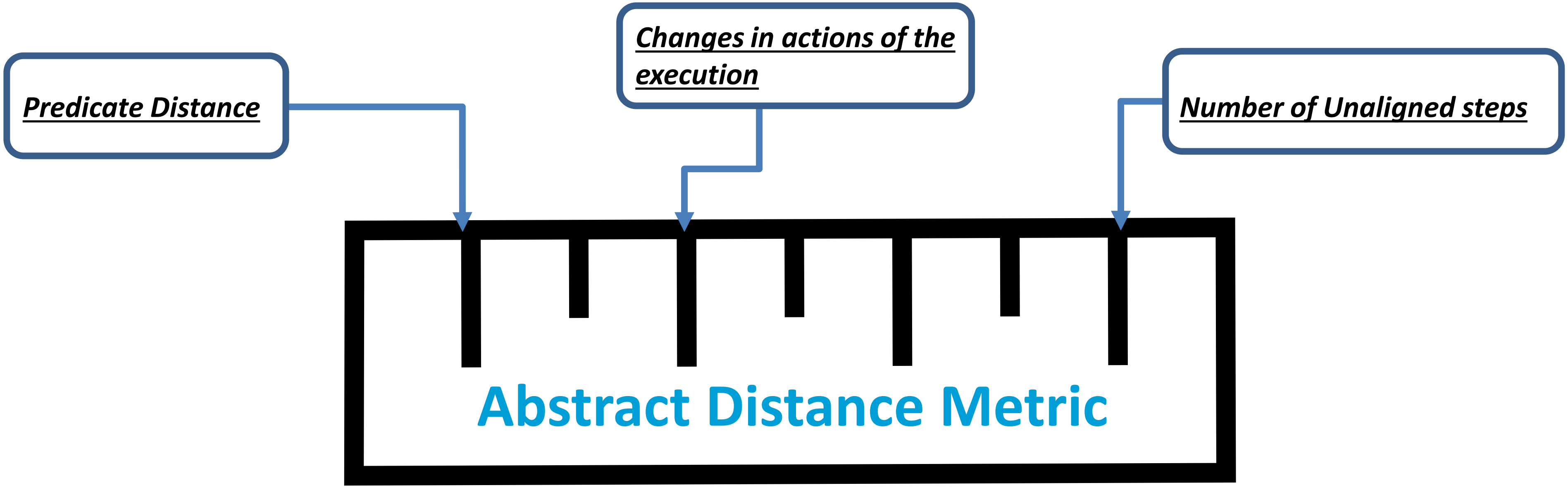
- Aligned steps
- Unaligned steps = NOT aligned steps

Distance Metrics

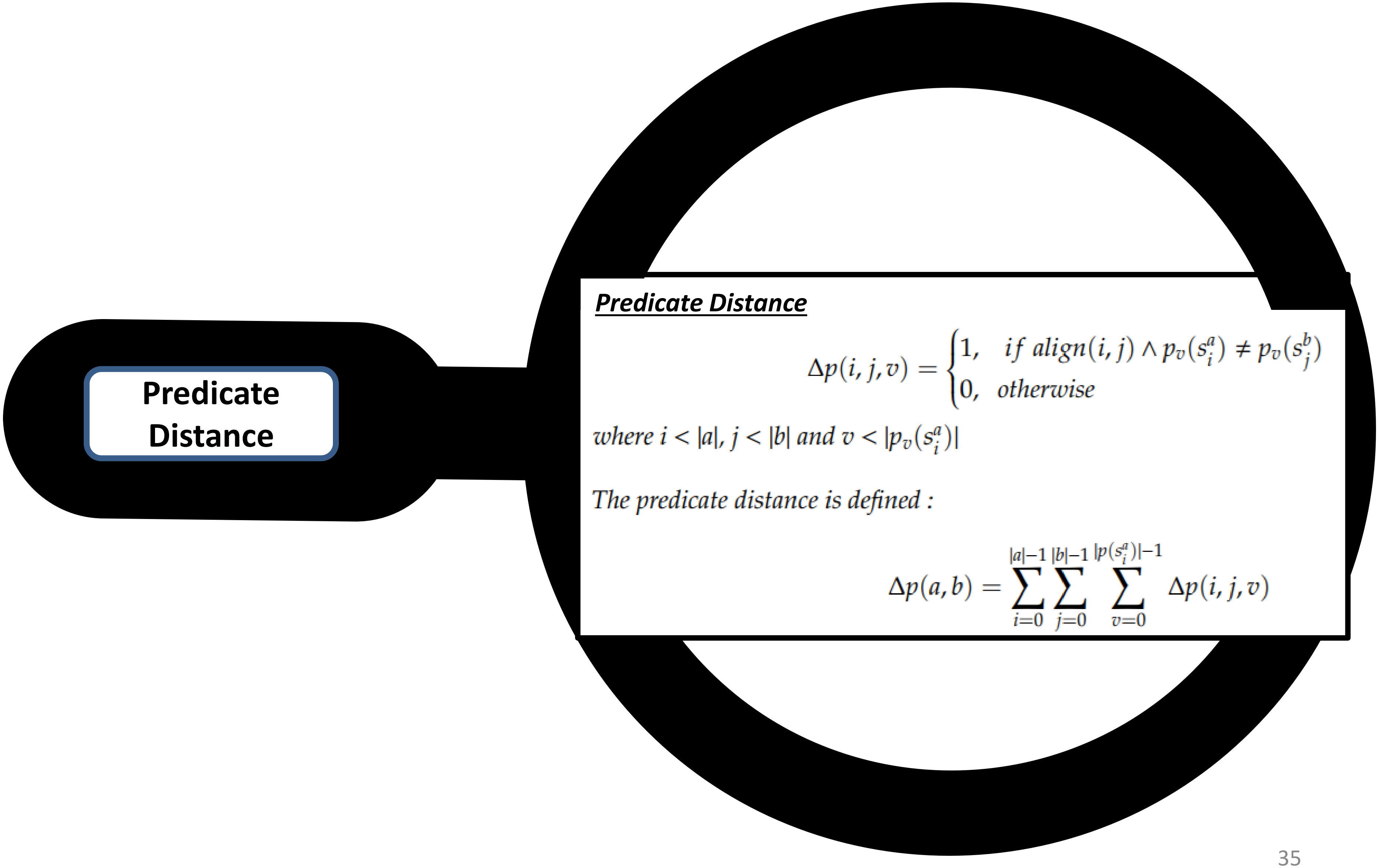
Distance Metrics

1. **Abstract Distance Metric**

Abstract Distance Metric



Abstract Distance Metric



Predicate Distance

Predicate Distance

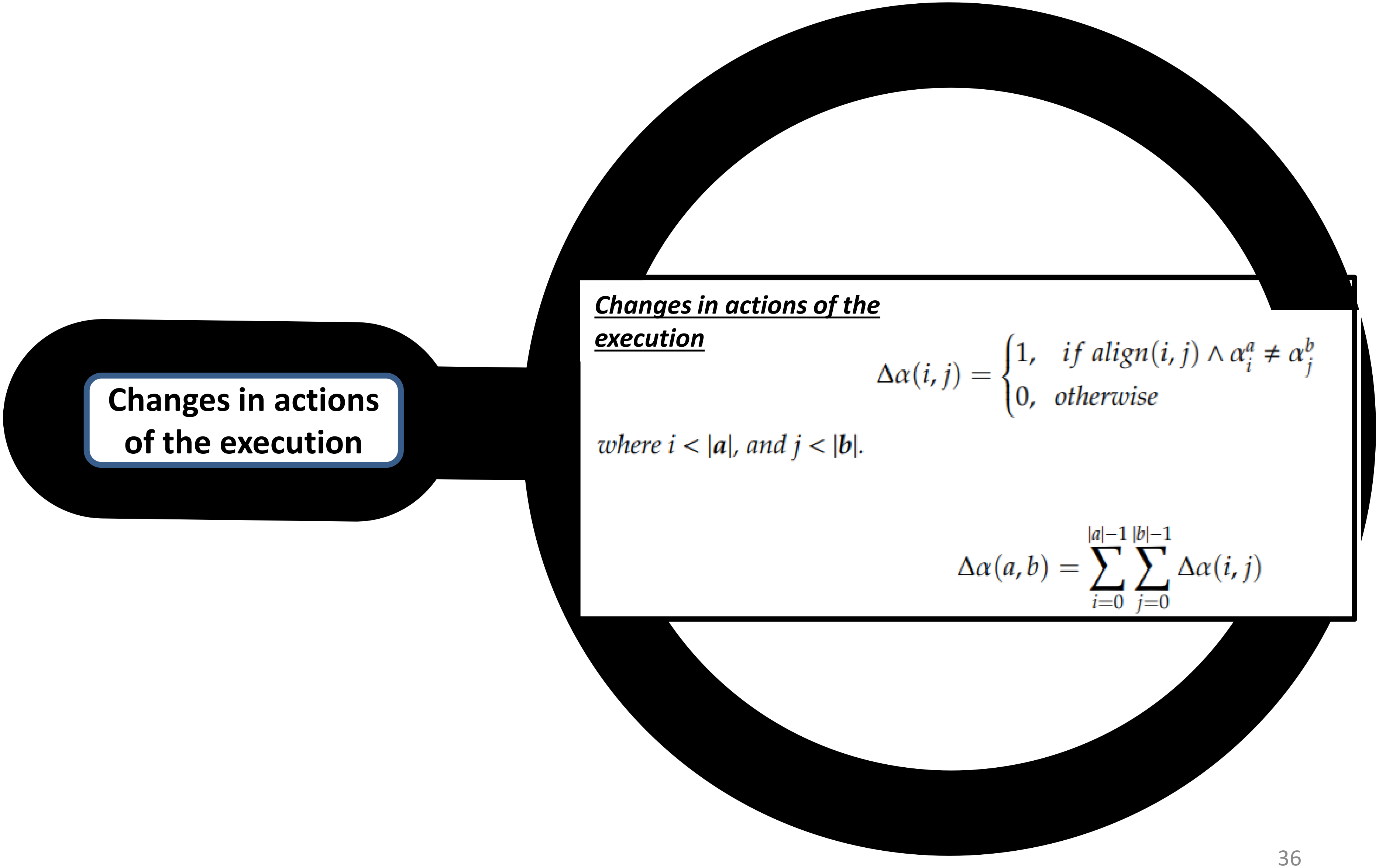
$$\Delta p(i,j,v) = \begin{cases} 1, & if\ align(i,j) \wedge p_v(s_i^a) \neq p_v(s_j^b) \\ 0, & otherwise \end{cases}$$

where $i < |a|$, $j < |b|$ *and* $v < |p_v(s_i^a)|$

The predicate distance is defined :

$$\Delta p(a,b) = \sum_{i=0}^{|a|-1} \sum_{j=0}^{|b|-1} \sum_{v=0}^{|p(s_i^a)|-1} \Delta p(i,j,v)$$

Abstract Distance Metric



Changes in actions of the execution

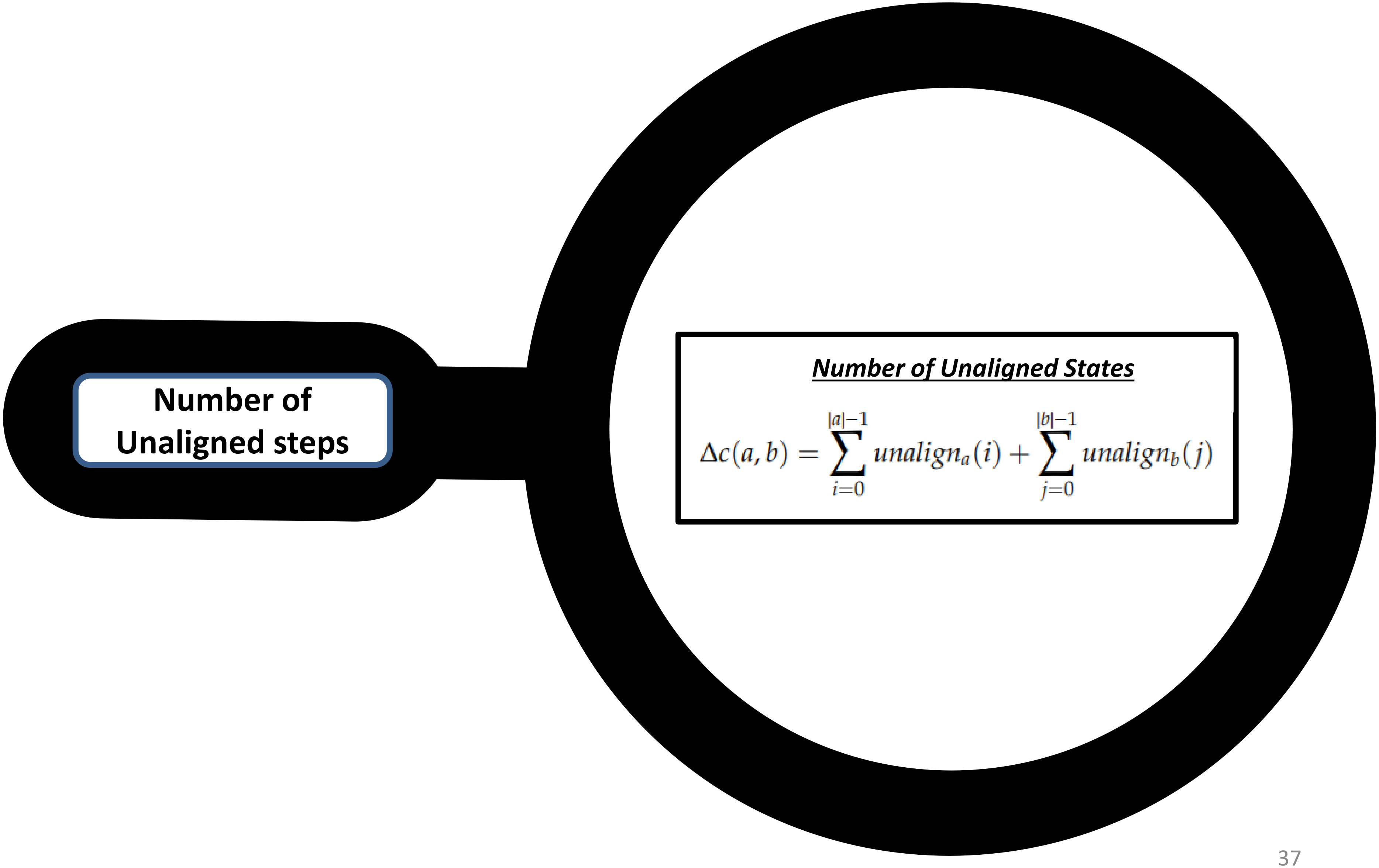
__Changes in actions of the execution__

$$\Delta\alpha(i,j) = \begin{cases} 1, & if\ align(i,j) \wedge \alpha_i^a \neq \alpha_j^b \\ 0, & otherwise \end{cases}$$

$where\ i < |\mathbf{a}|,\ and\ j < |\mathbf{b}|.$

$$\Delta\alpha(a,b) = \sum_{i=0}^{|a|-1} \sum_{j=0}^{|b|-1} \Delta\alpha(i,j)$$

Abstract Distance Metric

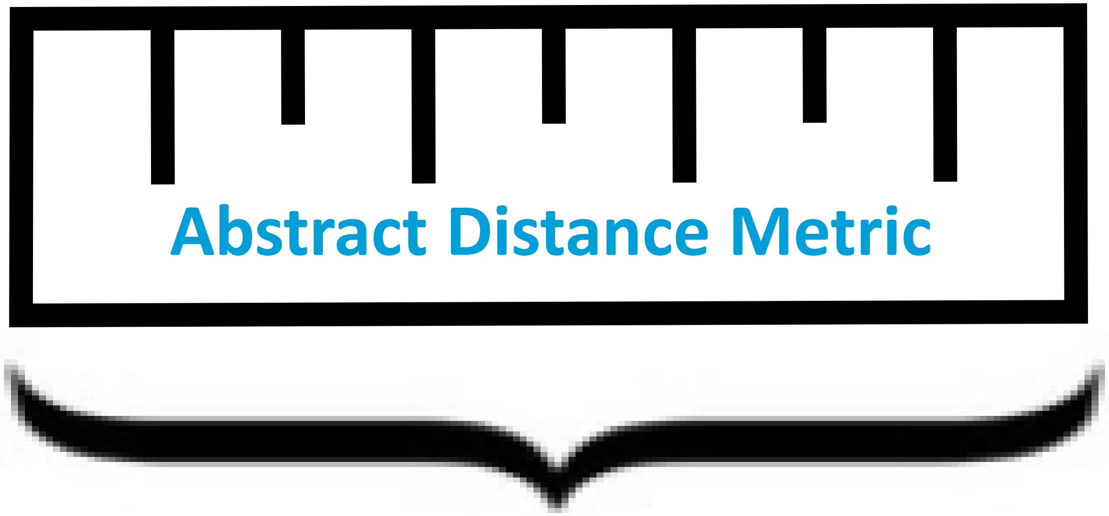


Number of Unaligned steps

Number of Unaligned States

$$\Delta c(a,b) = \sum_{i=0}^{|a|-1} unalign_a(i) + \sum_{j=0}^{|b|-1} unalign_b(j)$$

ADM


Abstract Distance Metric

$$\boxed{d(a,b) = W_p \cdot \Delta p(a,b) + W_a \cdot \Delta \alpha(a,b) + W_c \cdot \Delta c(a,b)}$$

Distance Metrics

1. **Abstract Distance Metric**
2. **Control Flow Distance Metric**

Control Flow Distance Metric

Control Flow Distance Metric

- Steps:
  - Find all branches
  - Align the branches of the two executions with each other
  - Compare the outgoing edges of the aligned branches

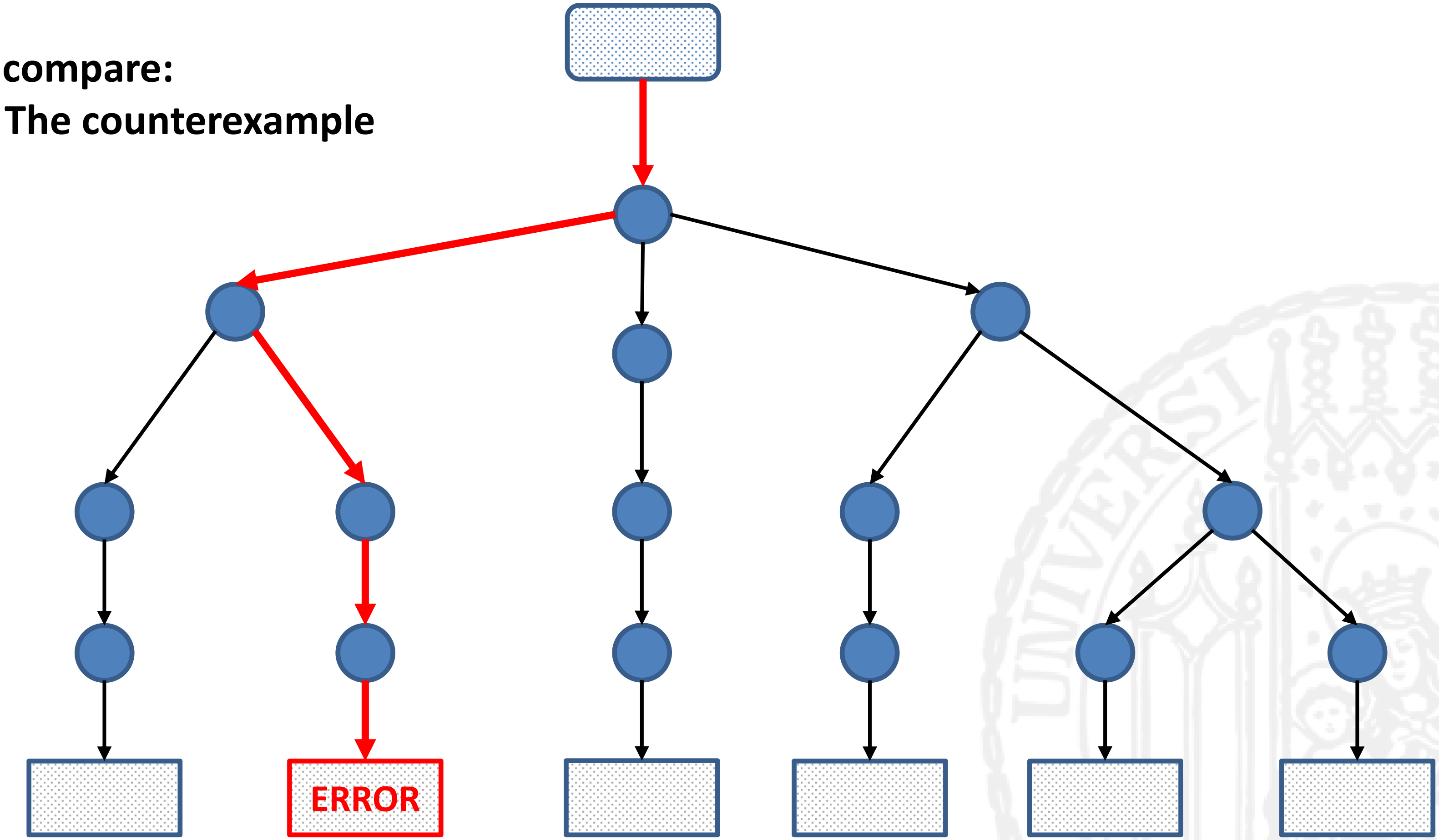
Control Flow Distance Metric

Example

Control Flow Distance Metric

To compare:

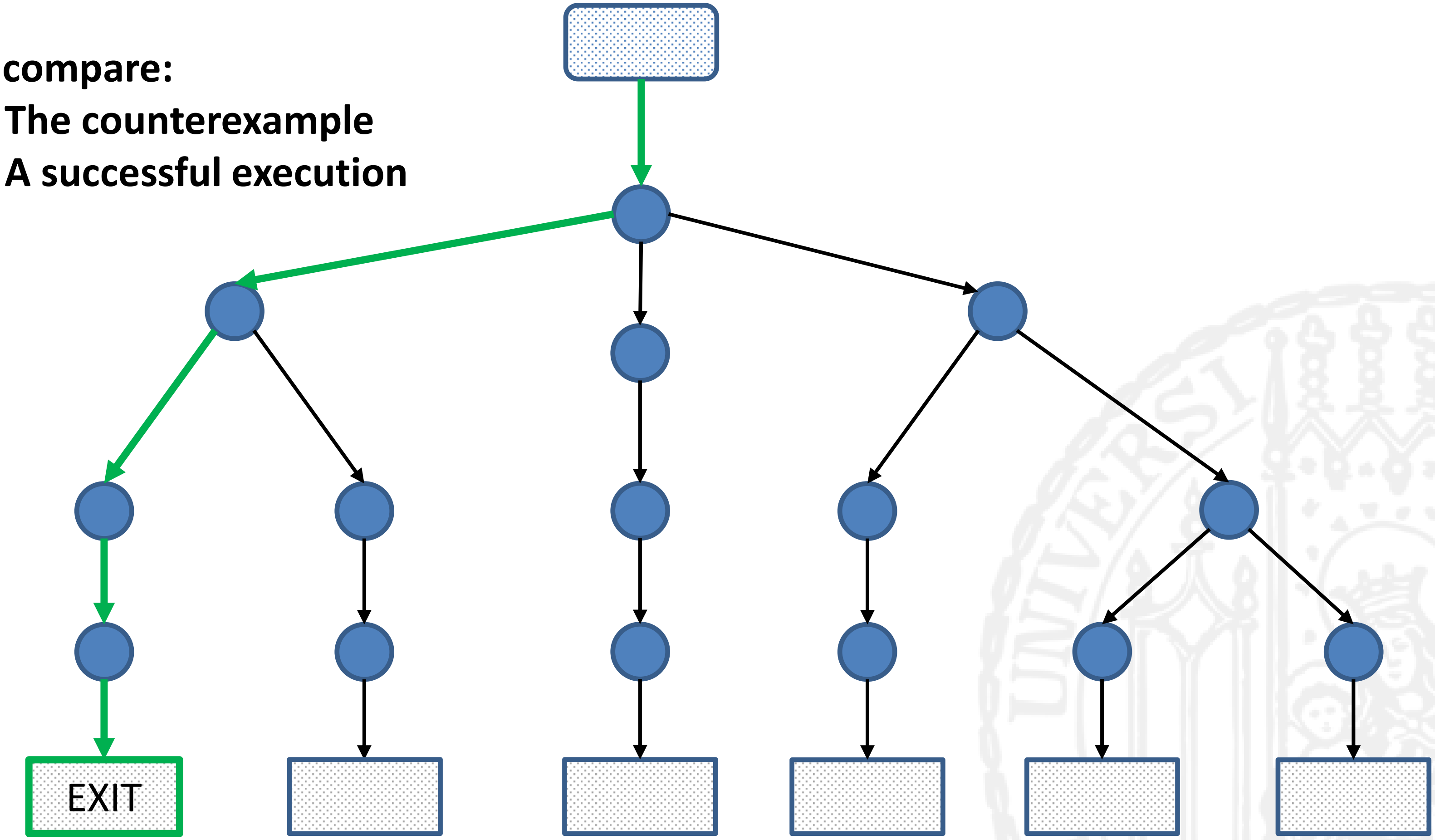
1. **The counterexample**



Control Flow Distance Metric

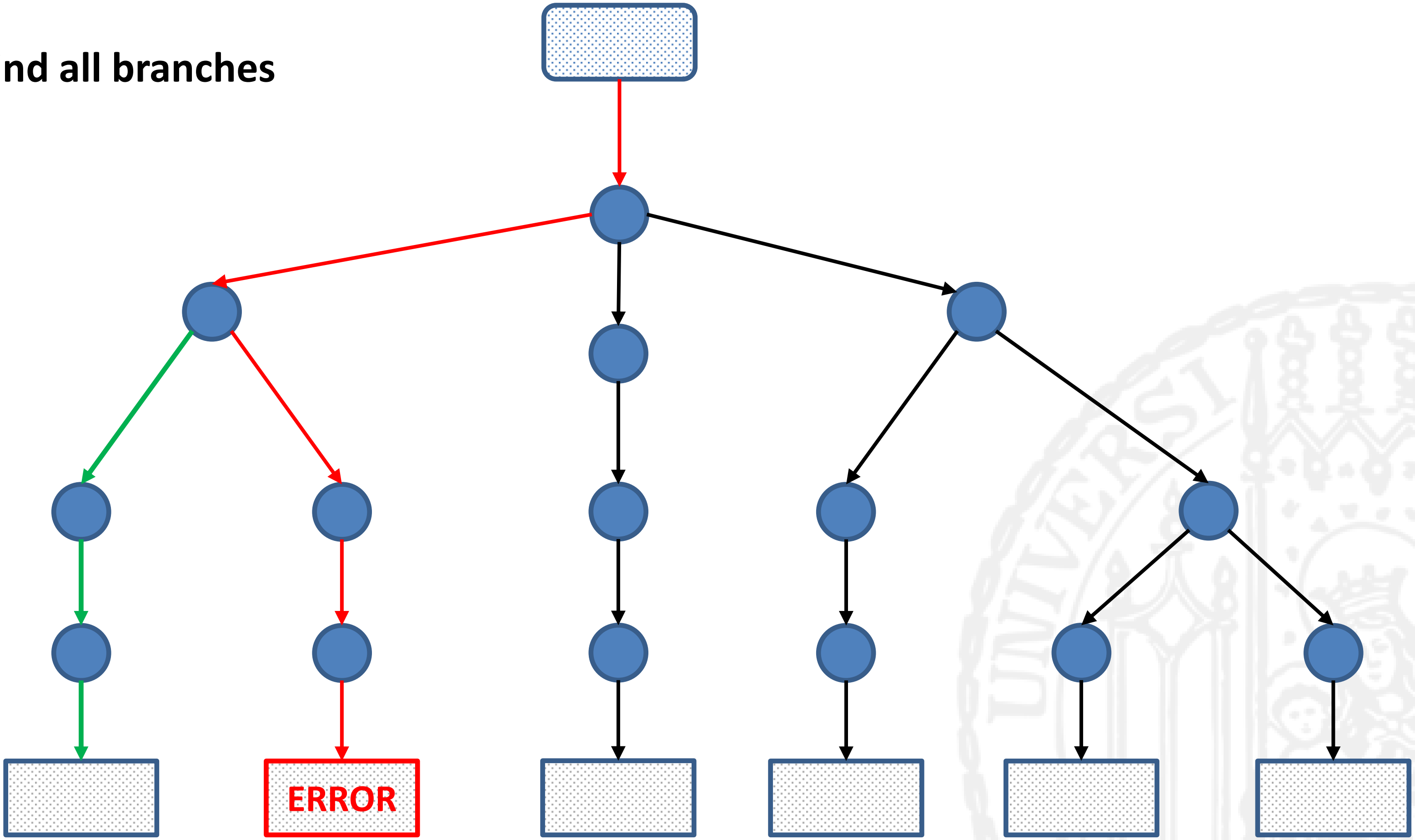
To compare:

1. **The counterexample**
2. **A successful execution**



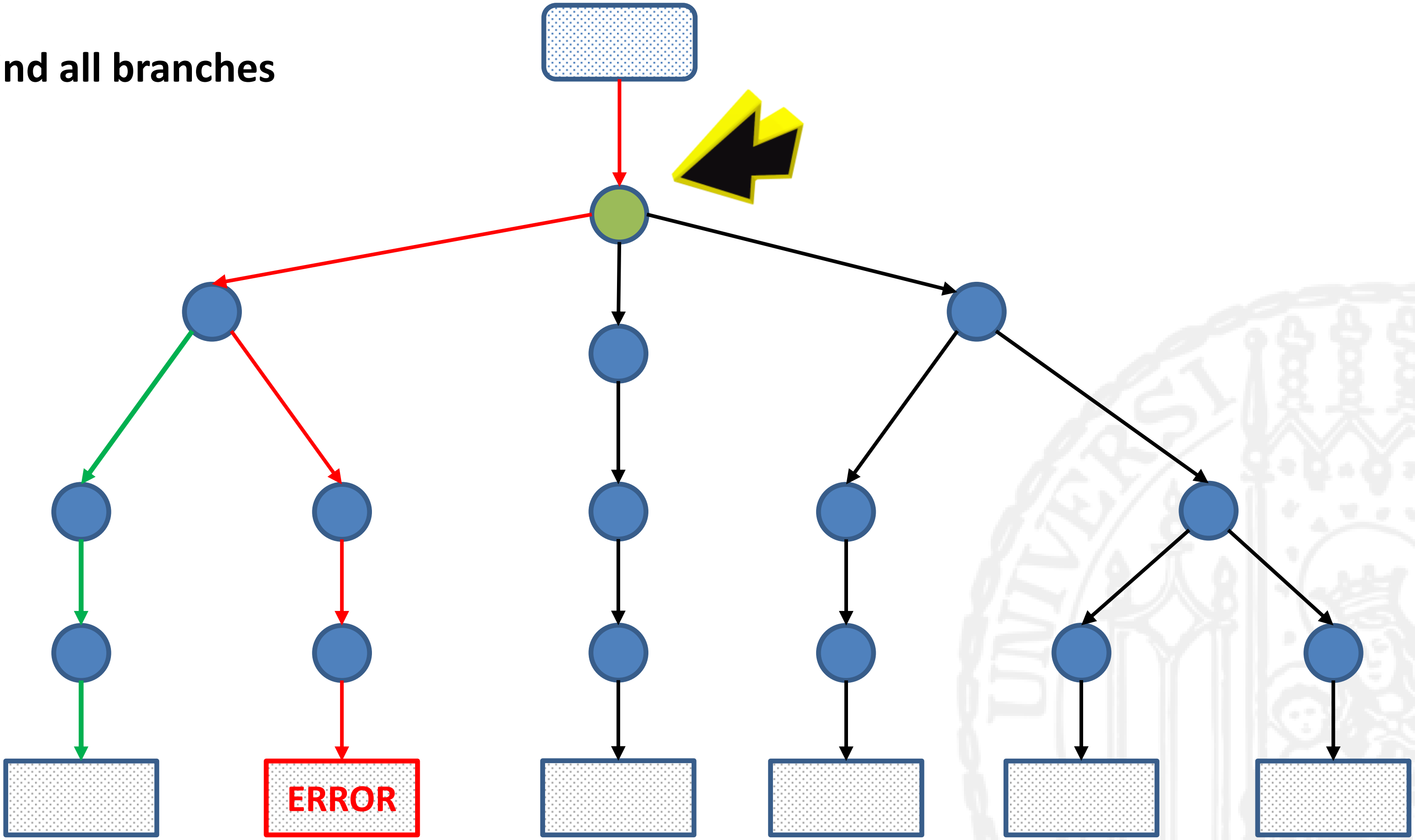
Control Flow Distance Metric

1. Find all branches



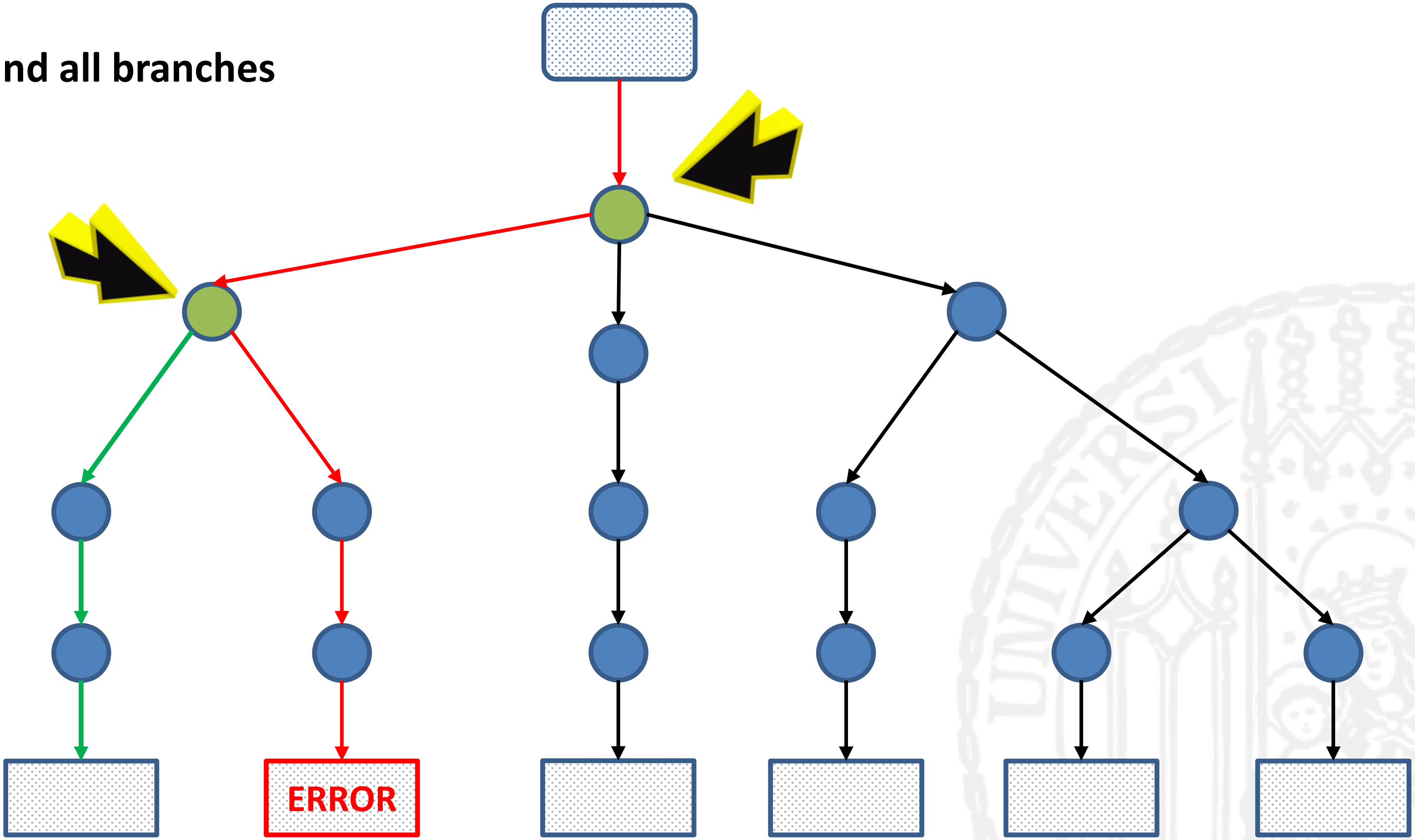
Control Flow Distance Metric

1. **Find all branches**



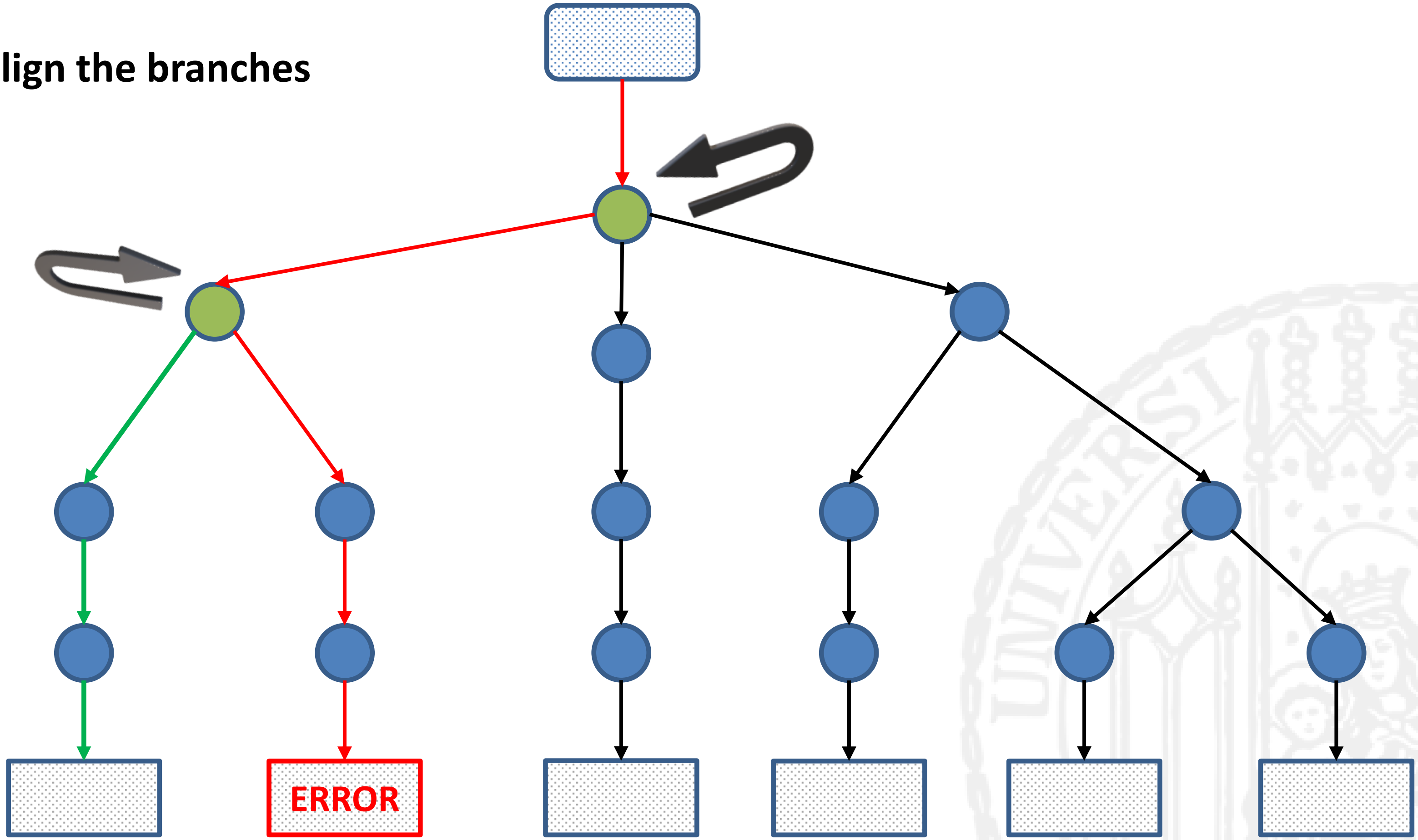
Control Flow Distance Metric

1. **Find all branches**

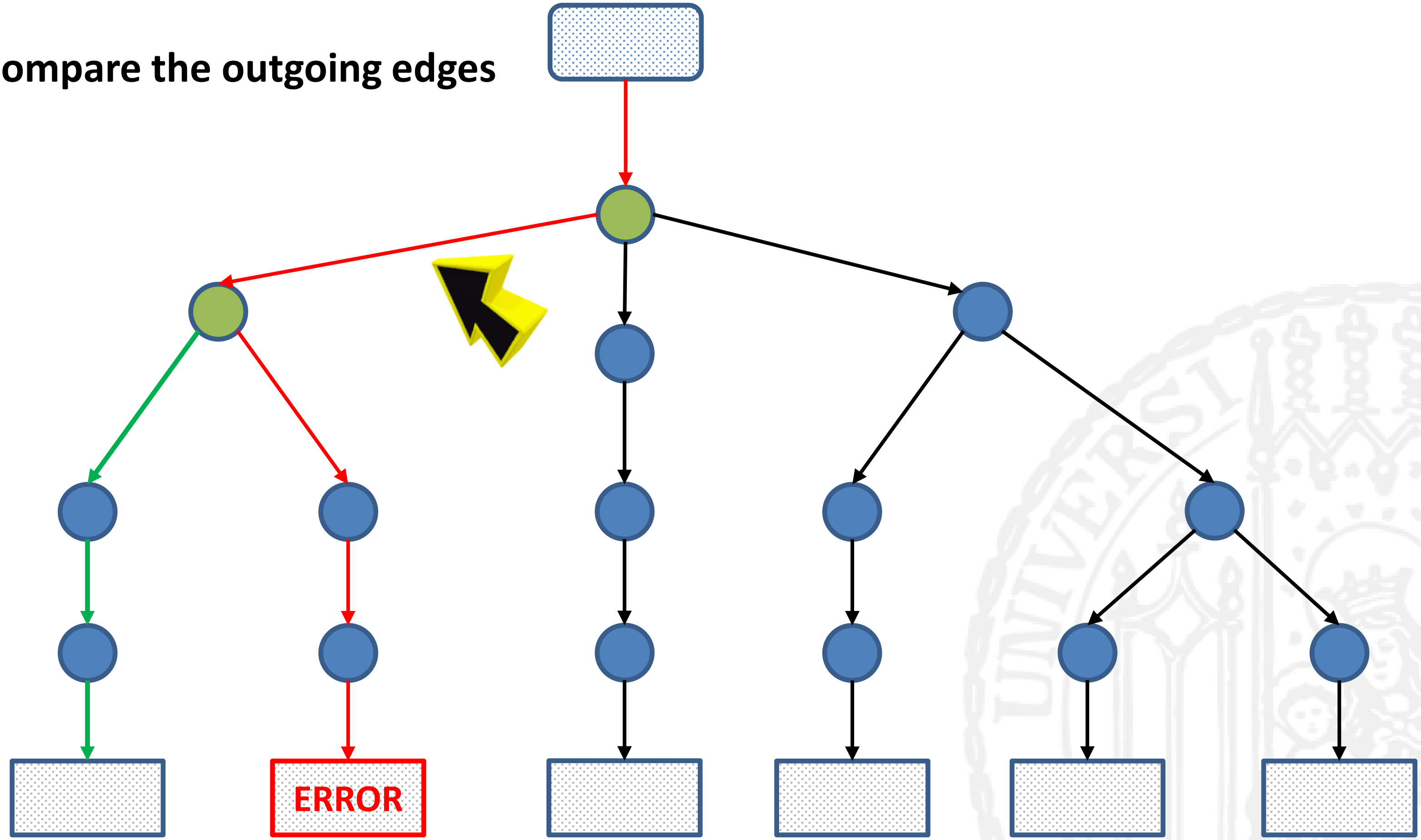


Control Flow Distance Metric

2. Align the branches



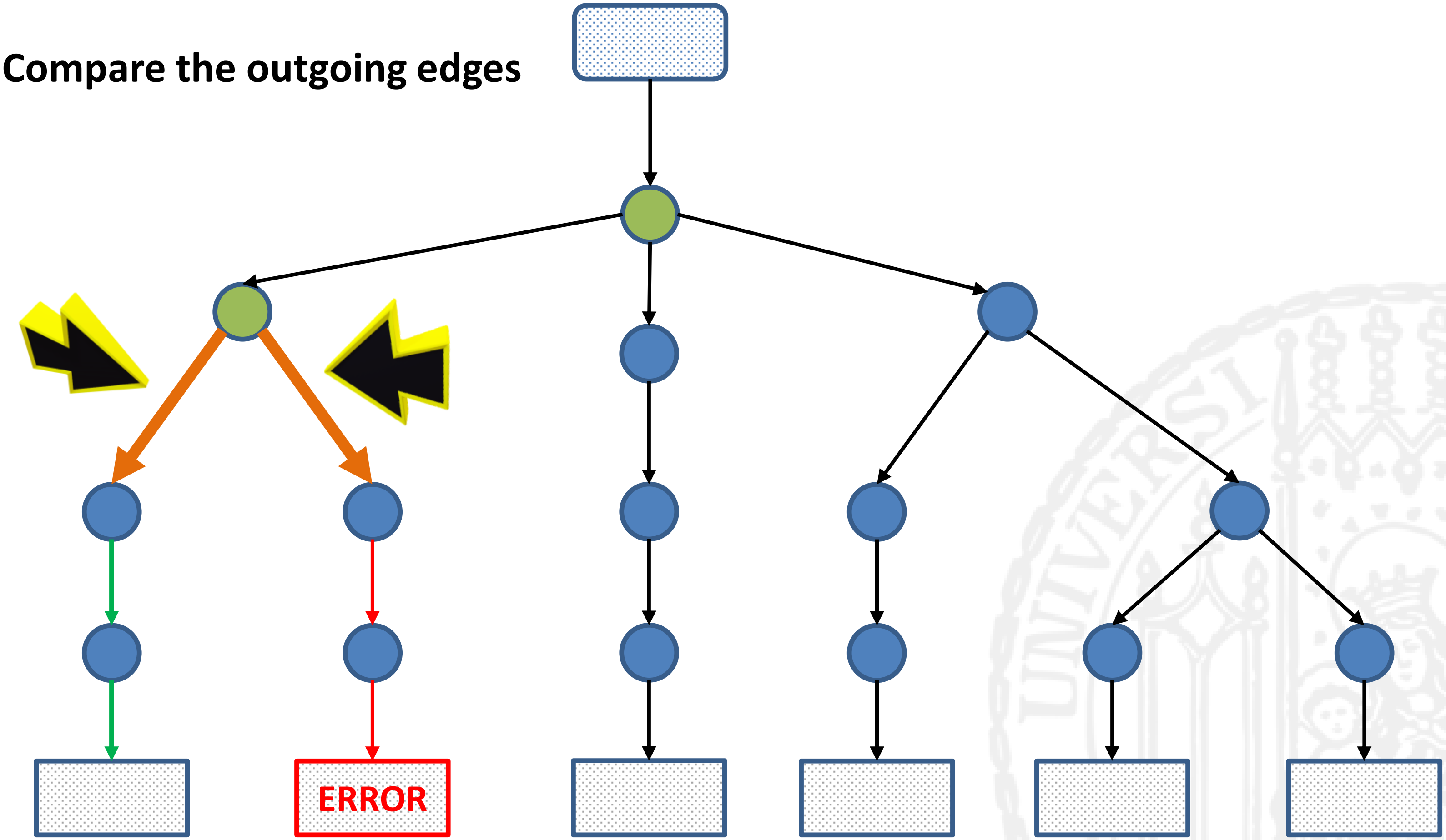
Control Flow Distance Metric



3. Compare the outgoing edges

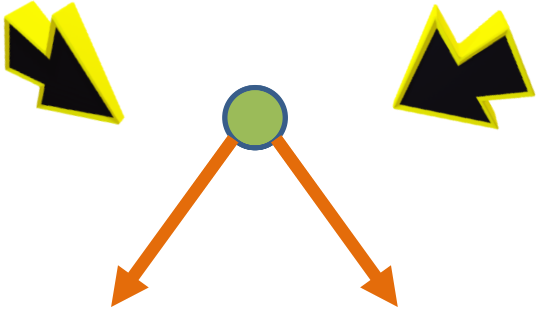
Control Flow Distance Metric

3. Compare the outgoing edges



Control Flow Distance Metric

The difference is:



Problem with Distance Metrics for Fault-Localization

If the number of successful executions is excessive:

i. Huge amount of calculations to find all the successful executions

ii. Enormous number of comparison in order to find the successful execution which is closer to the counterexample

Solution!

Automated Path Generation

Path Generation

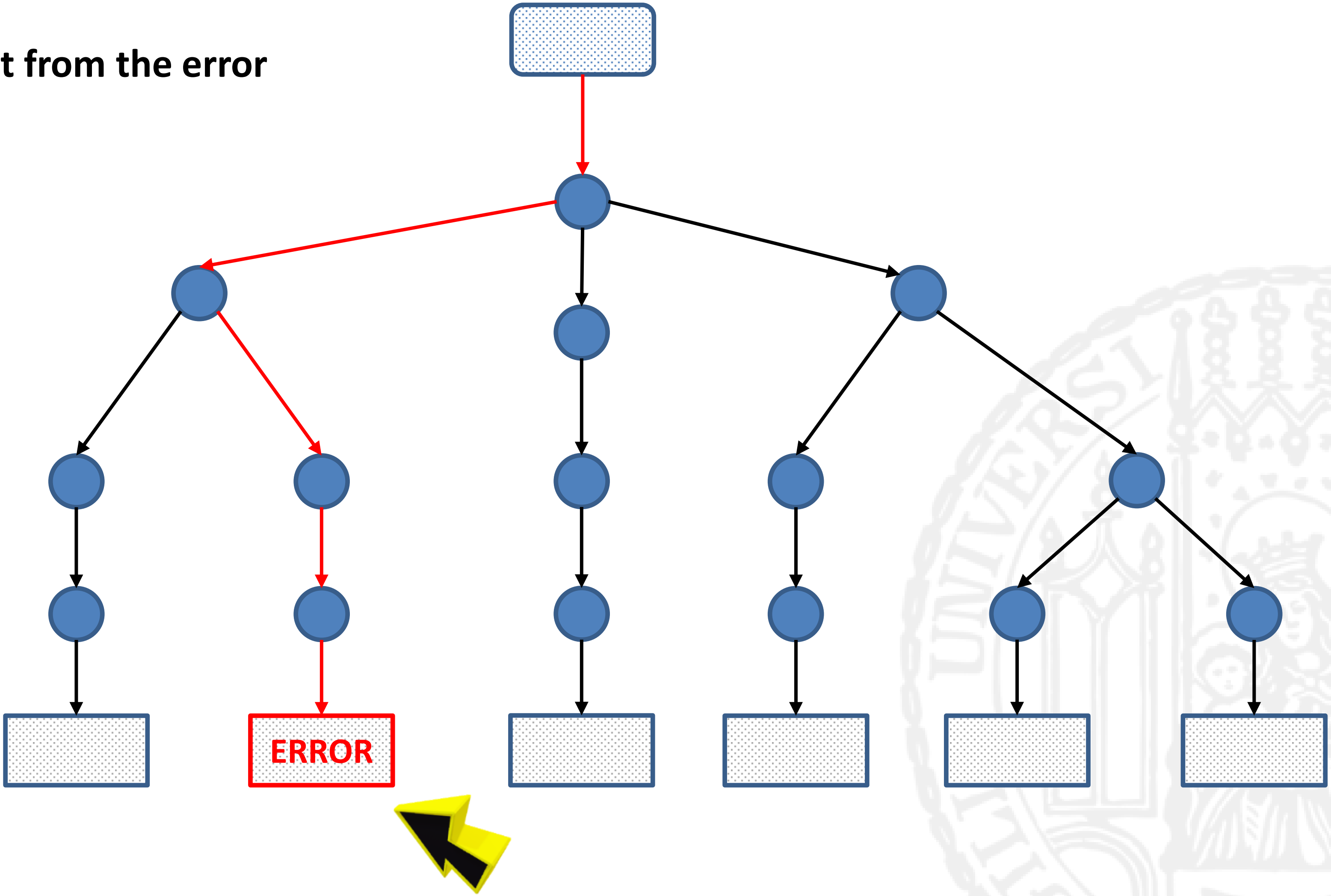
- Instead of comparing all successful executions with the counterexample
- We generate automatically the successful execution which is closer to the counterexample
- Much faster

Path Generation

Example

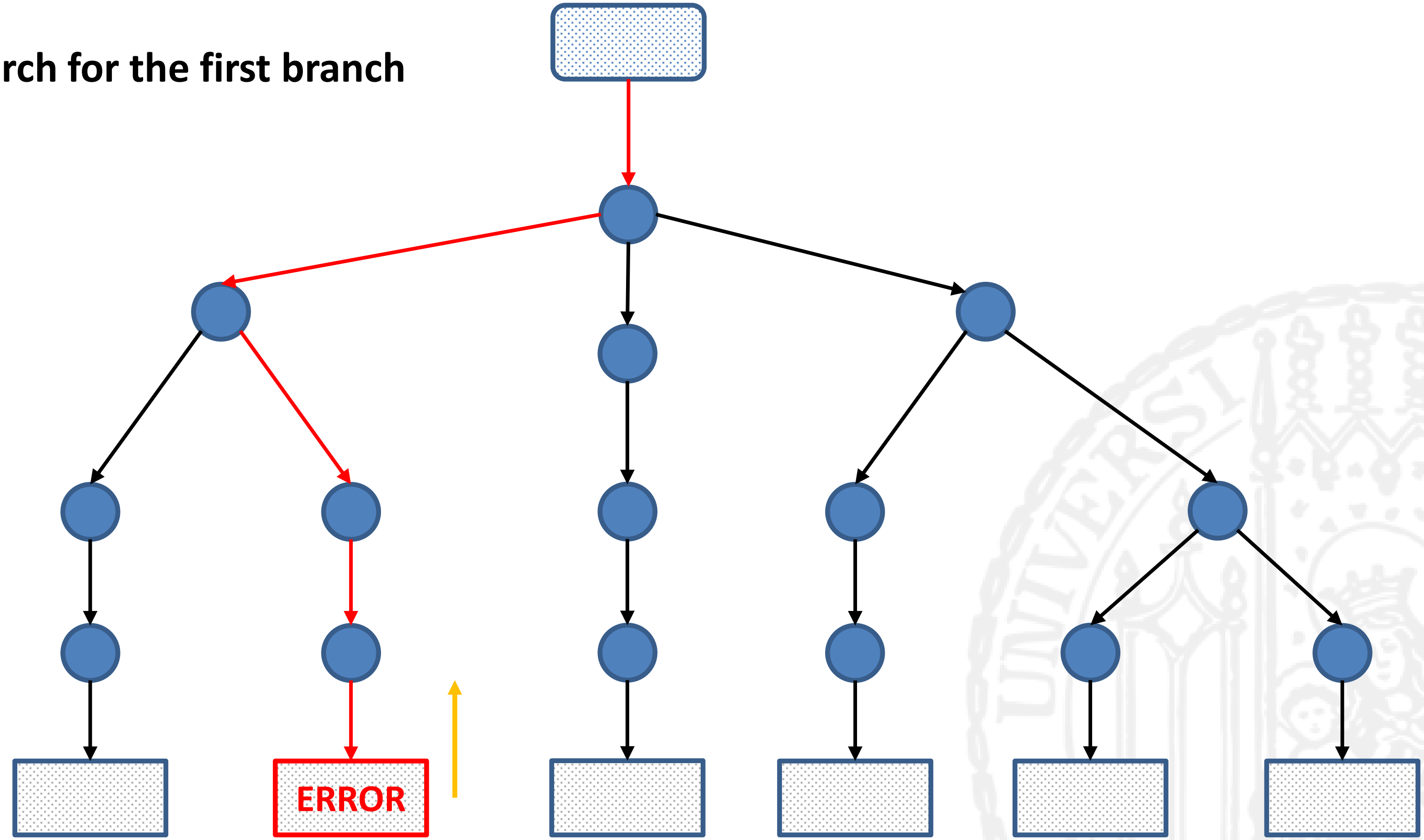
Path Generation

Start from the error



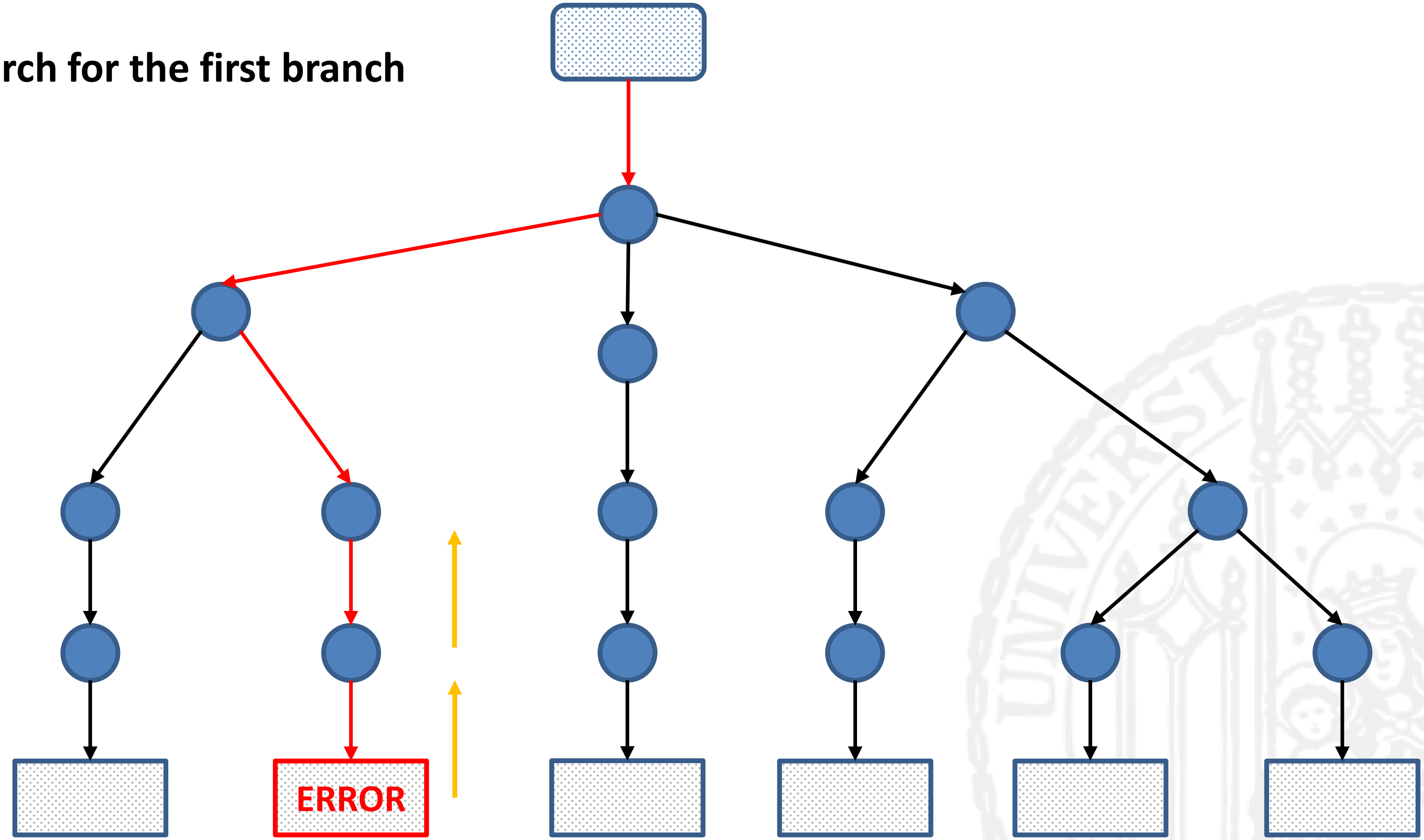
Path Generation

Search for the first branch



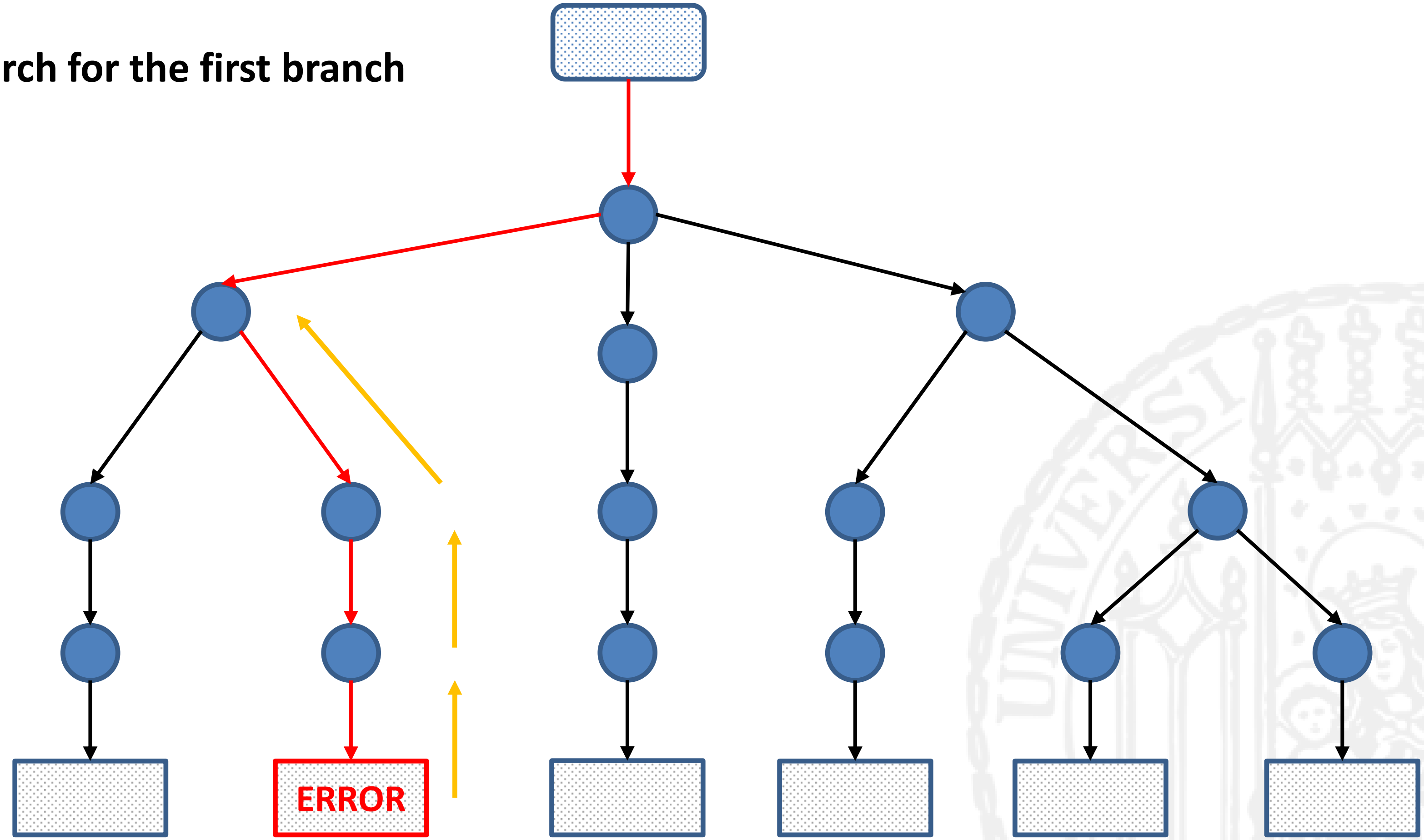
Path Generation

Search for the first branch



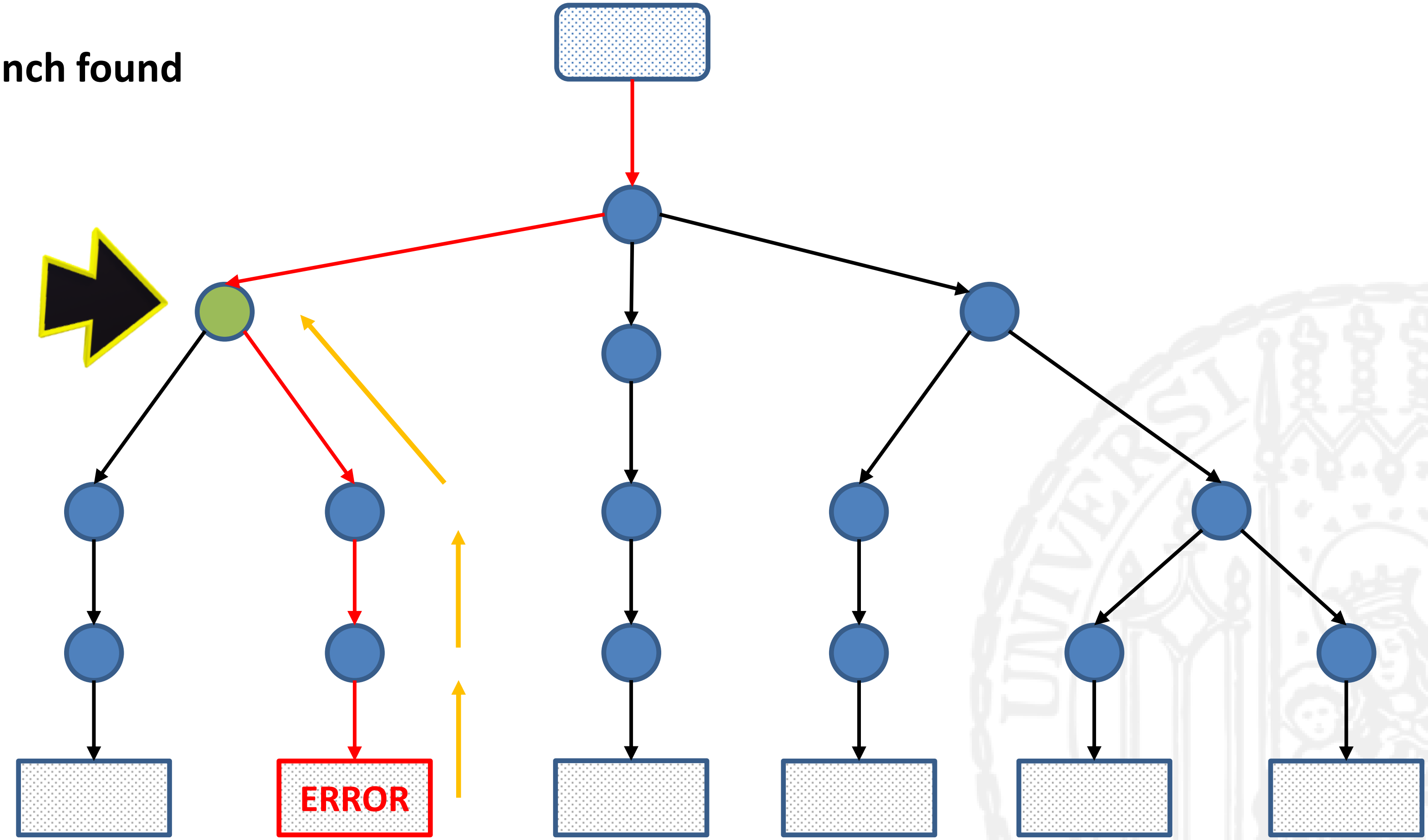
Path Generation

Search for the first branch



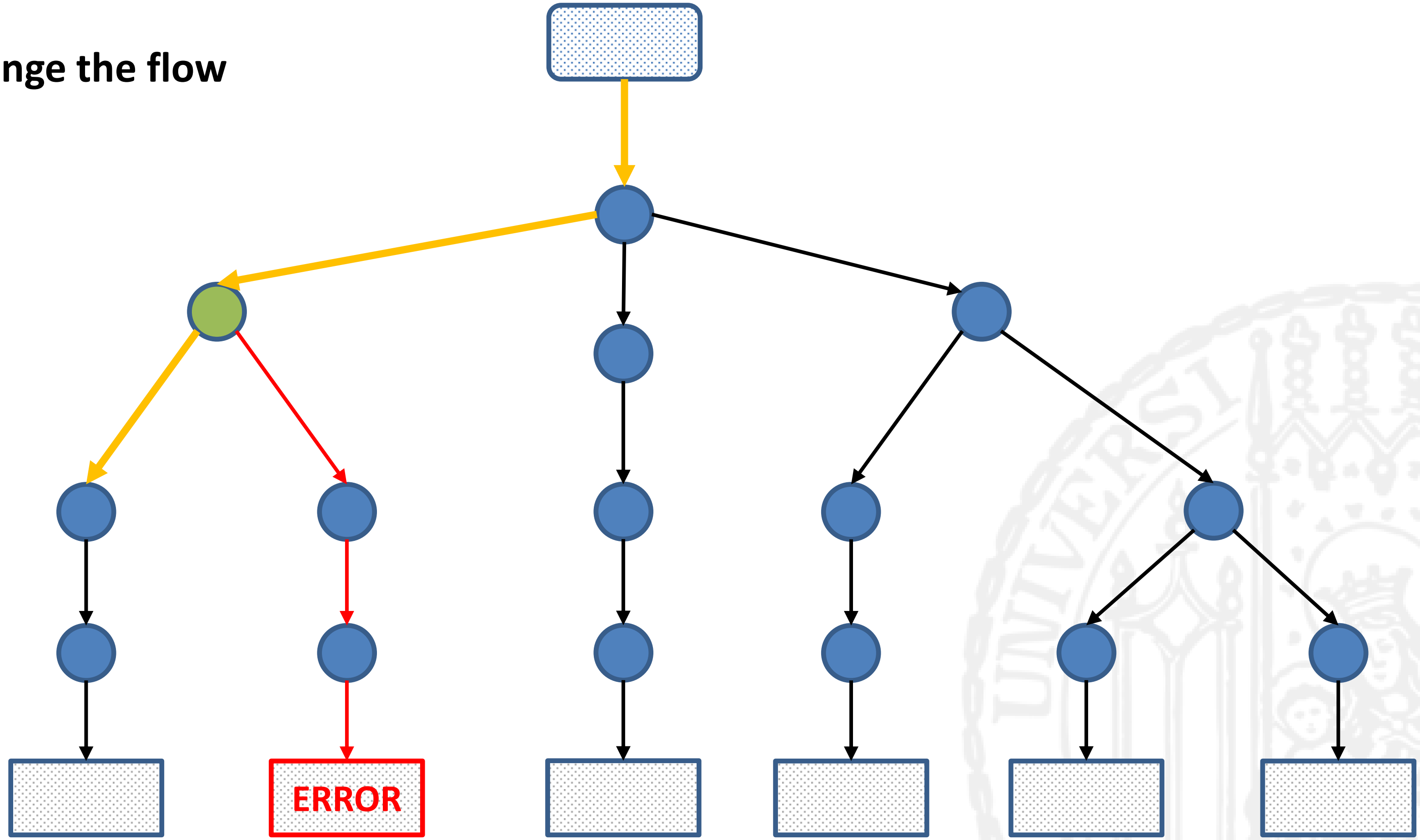
Path Generation

Branch found



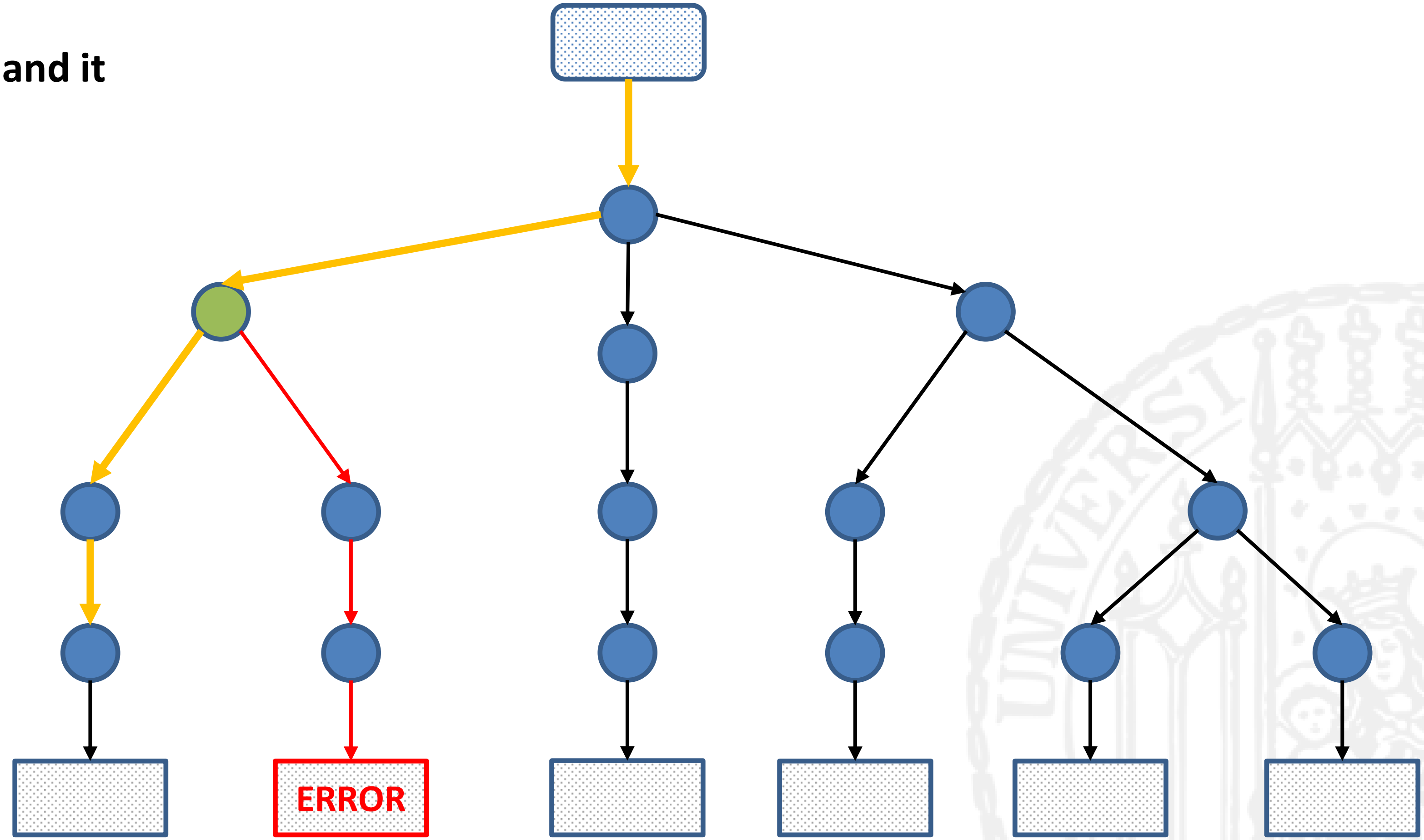
Path Generation

Change the flow



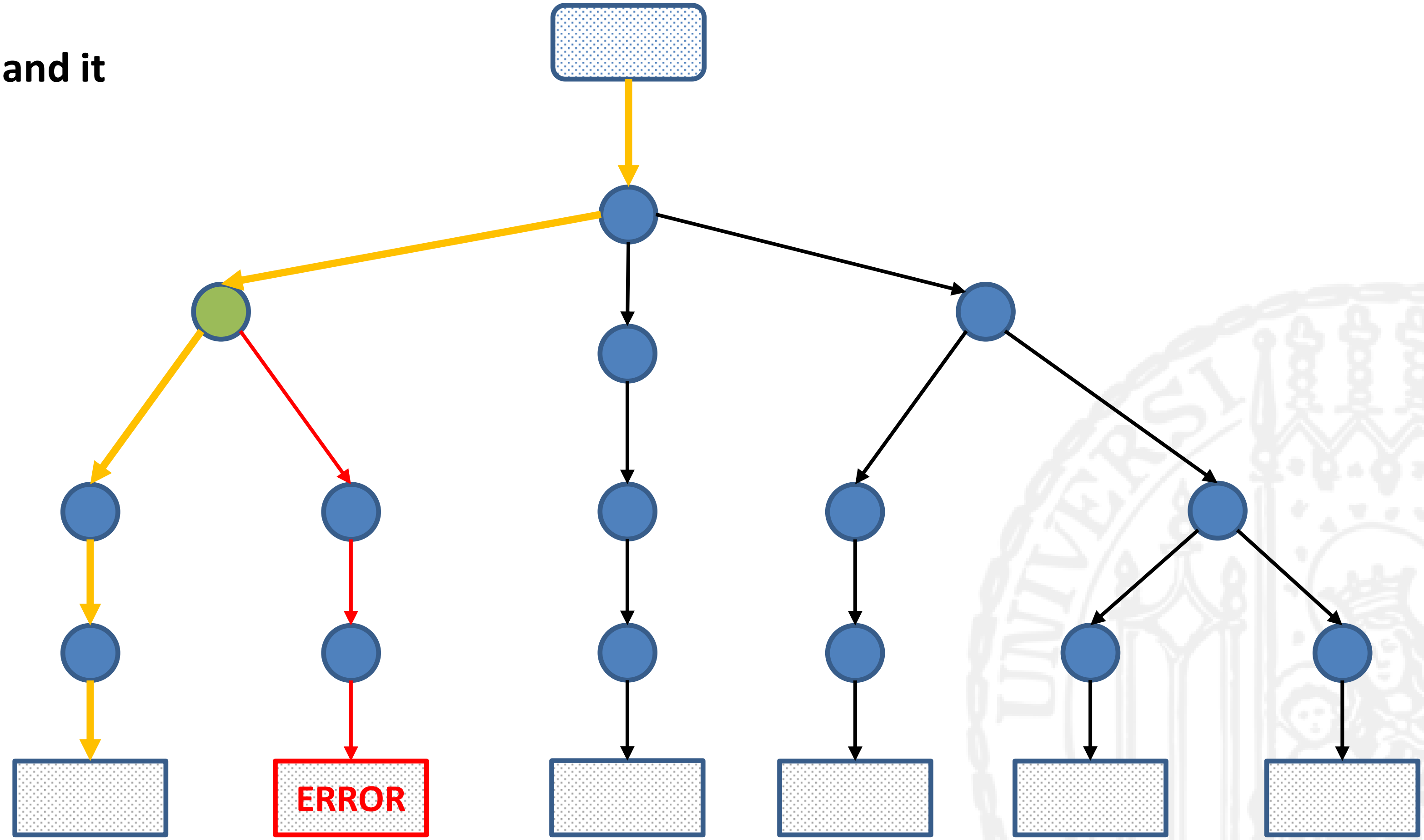
Path Generation

Expand it



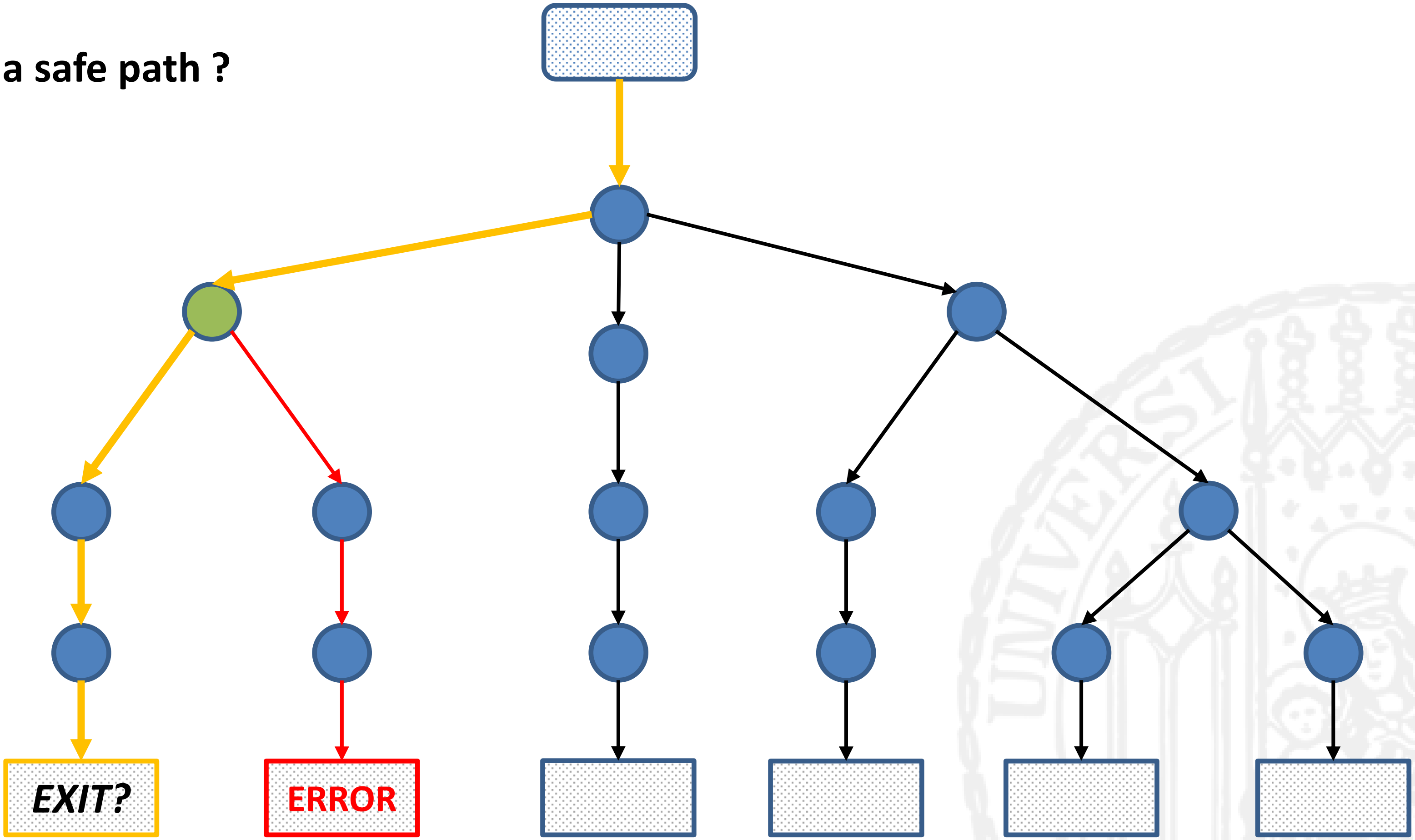
Path Generation

Expand it



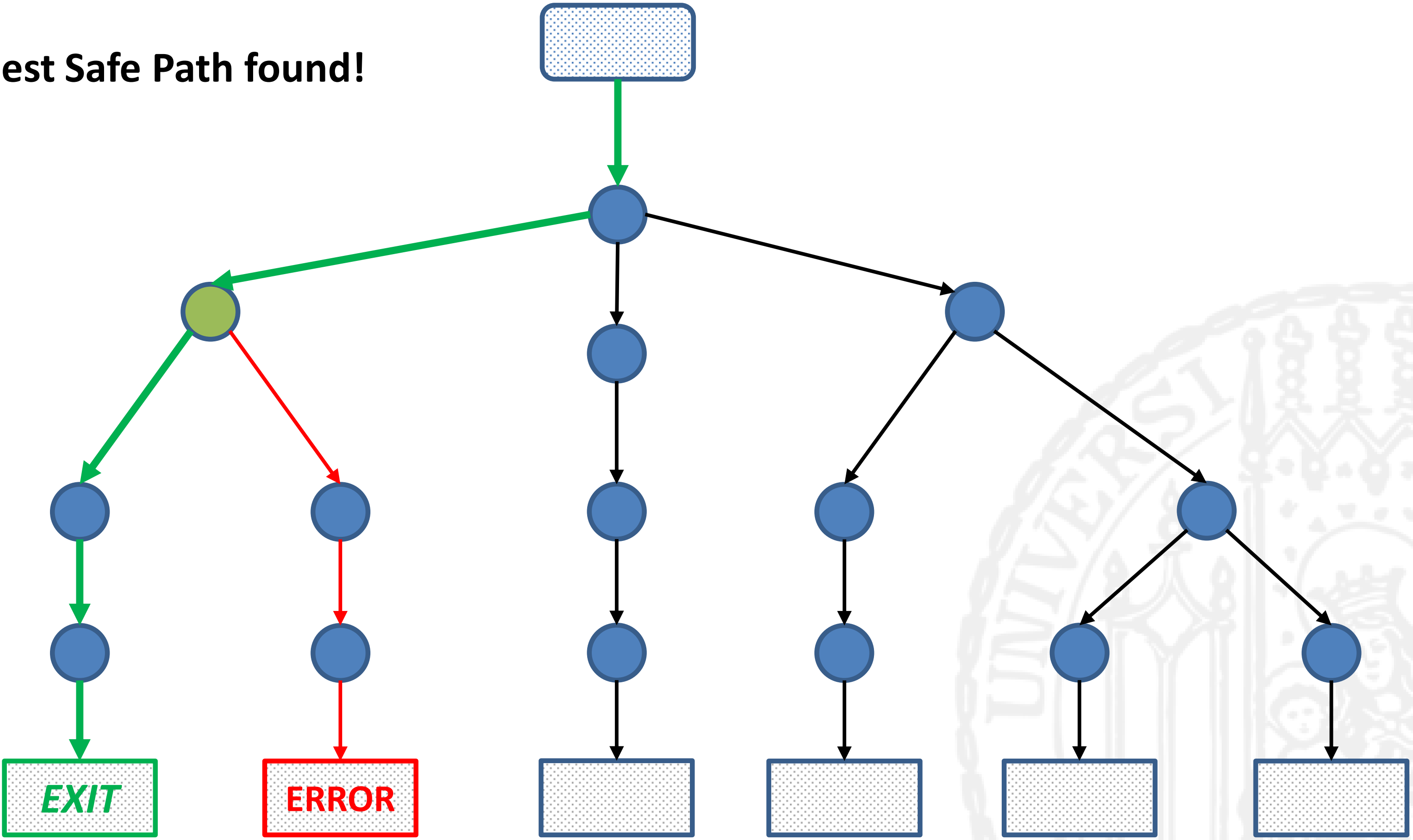
Path Generation

Is it a safe path ?



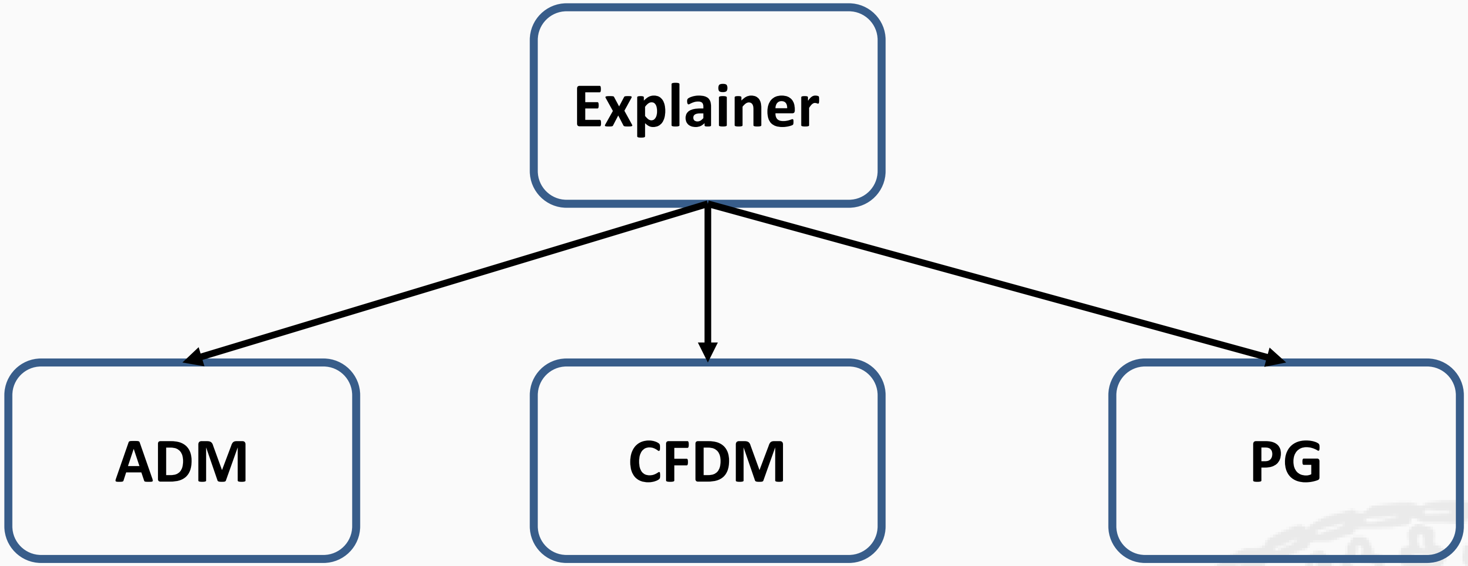
Path Generation

Closest Safe Path found!



Implementation

Explainer

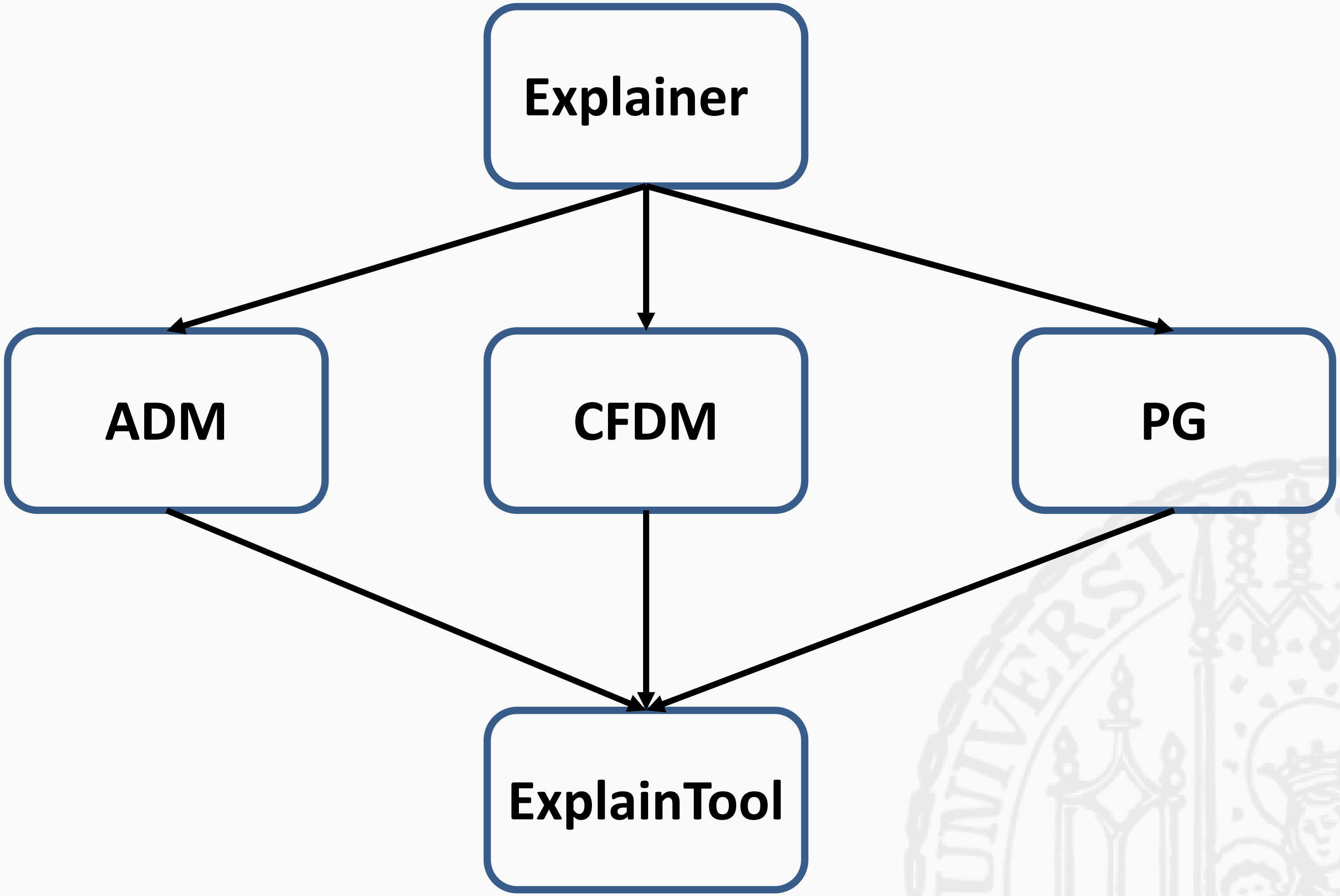


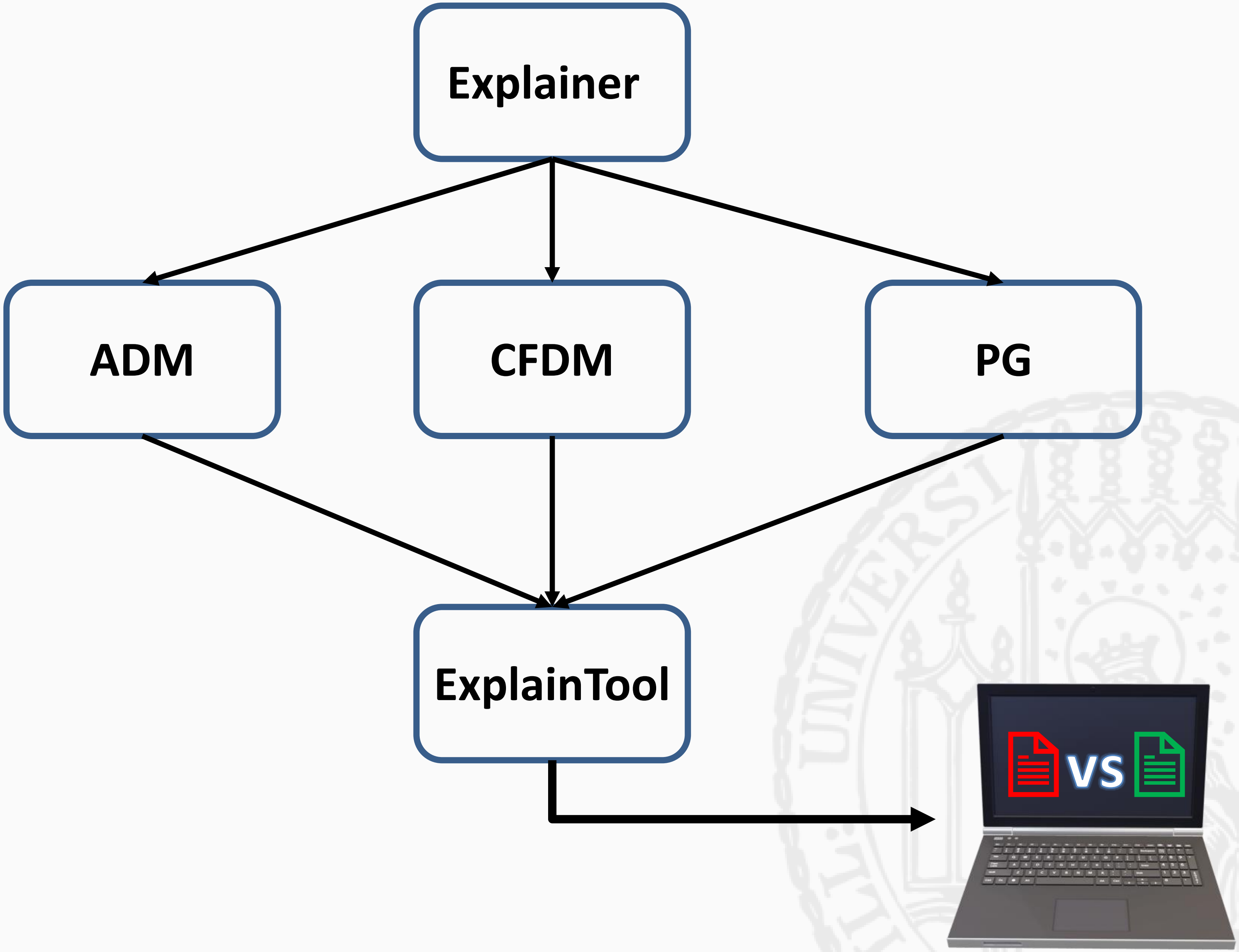
Explainer

ADM

CFDM

PG


Explainer
ADM
CFDM
PG
ExplainTool



Explainer

ADM

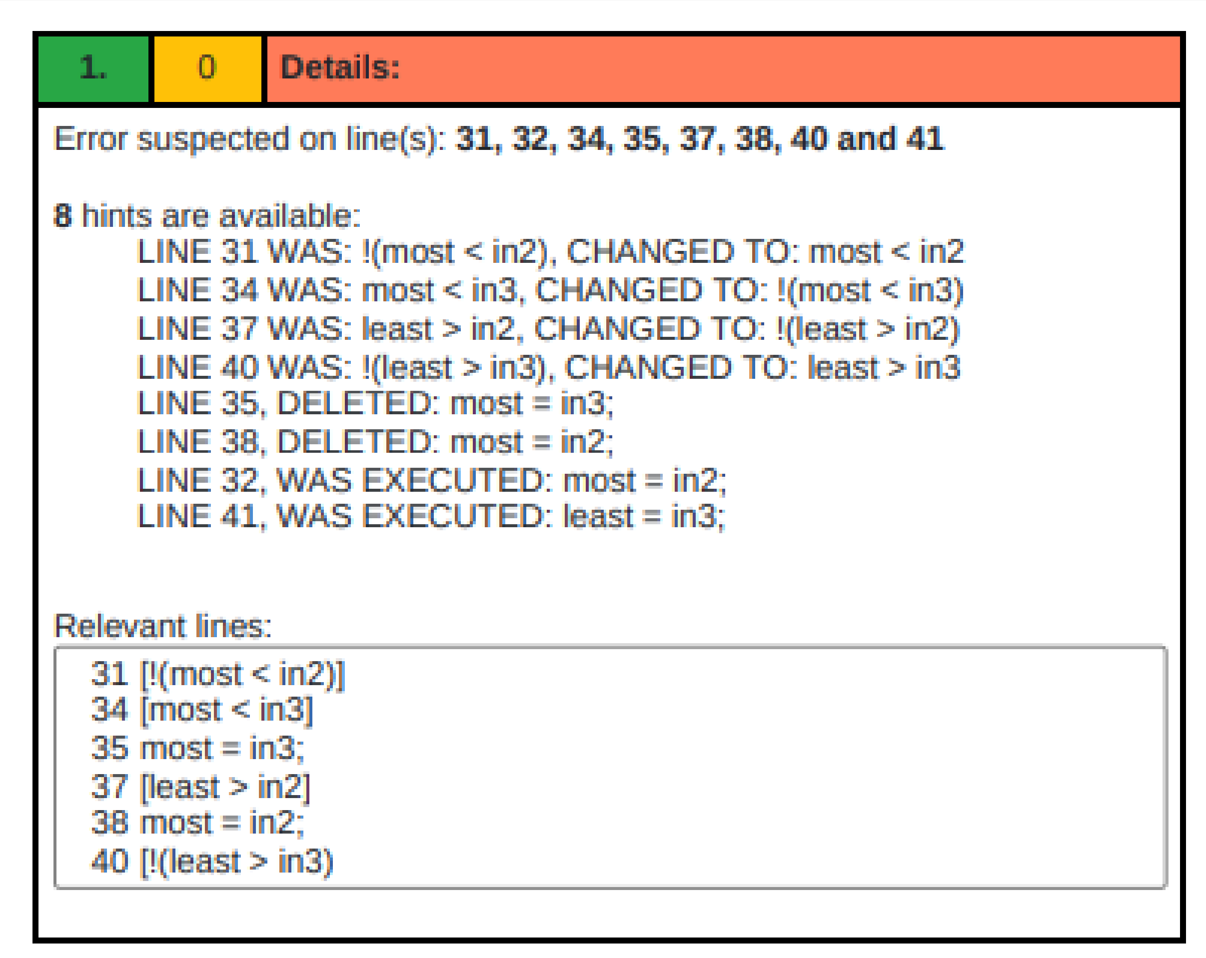
CFDM

PG

ExplainTool

Presentation of the Differences

=



1.	0	**Details:**

Error suspected on line(s): **31, 32, 34, 35, 37, 38, 40 and 41**

8 hints are available:

```
        LINE 31 WAS: !(most < in2), CHANGED TO: most < in2
        LINE 34 WAS: most < in3, CHANGED TO: !(most < in3)
        LINE 37 WAS: least > in2, CHANGED TO: !(least > in2)
        LINE 40 WAS: !(least > in3), CHANGED TO: least > in3
        LINE 35, DELETED: most = in3;
        LINE 38, DELETED: most = in2;
        LINE 32, WAS EXECUTED: most = in2;
        LINE 41, WAS EXECUTED: least = in3;
```

Relevant lines:

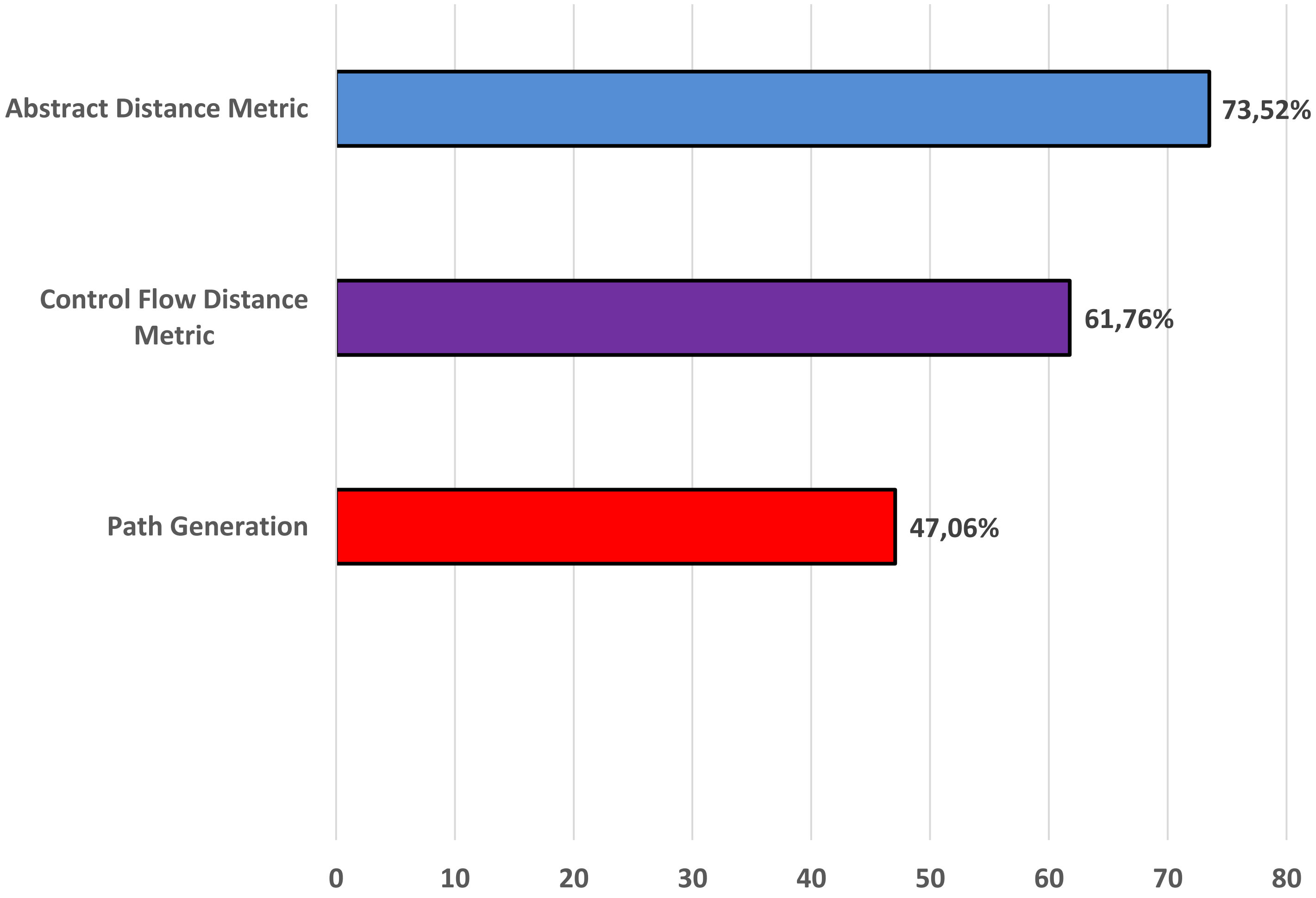
```
31 [!(most < in2)]
34 [most < in3]
35 most = in3;
37 [least > in2]
38 most = in2;
40 [!(least > in3)
```

Evaluation

Evaluation

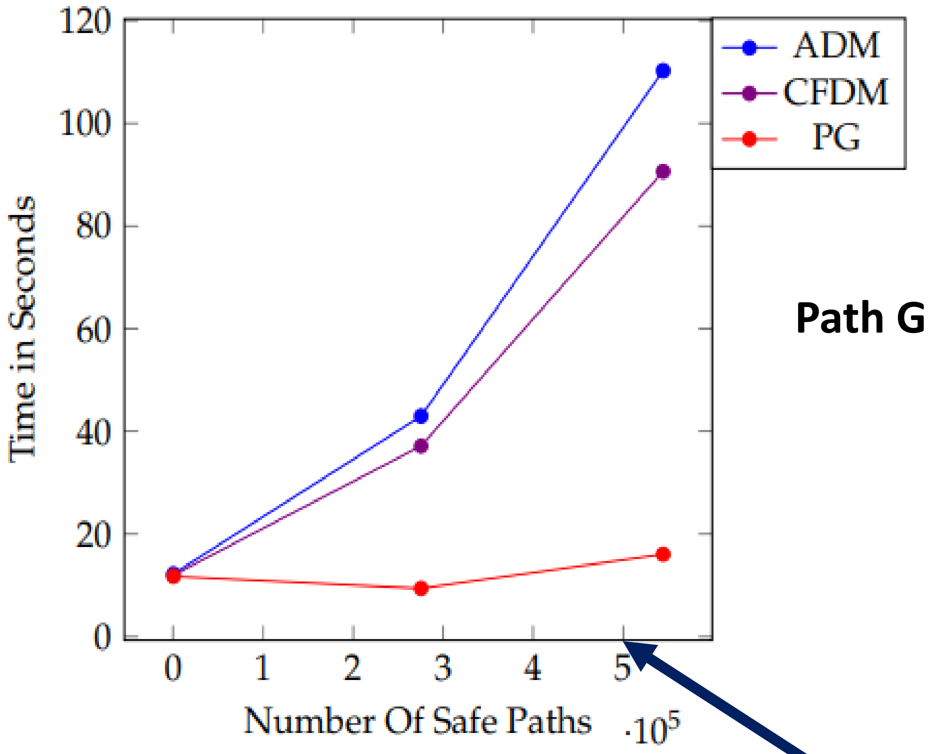
- We performed Quantitative and Runtime analysis of the three techniques

Overview of the Results



Runtime

Runtime



Path Generation is clearly faster

500.000 different successful executions

Ranking Function

$$Rank\colon [0,1] \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{R}$$

$$\mathrm{Rank}(\boldsymbol{successRate}, Hotlines, Differences) = \boldsymbol{successRate} * \frac{Hotlines}{Differences}$$

successRate = the possibility for the actual fault to be included in the set of differences

Ranking Function

$$Rank\colon [0,1] \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{R}$$

$$\text{Rank}(successRate, \boldsymbol{Hotlines}, Differences) = successRate * \frac{\boldsymbol{Hotlines}}{Differences}$$

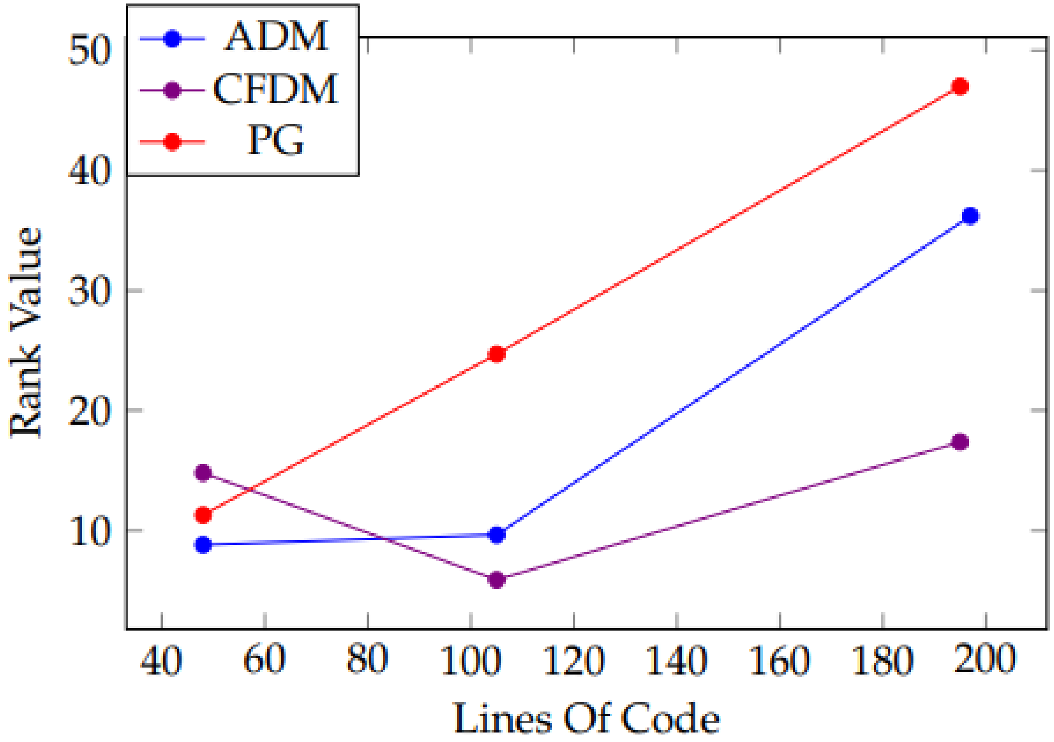
Hotlines >>> the more lines of code a program has, the more Code that the developer must go through looking for the fault

Ranking Function

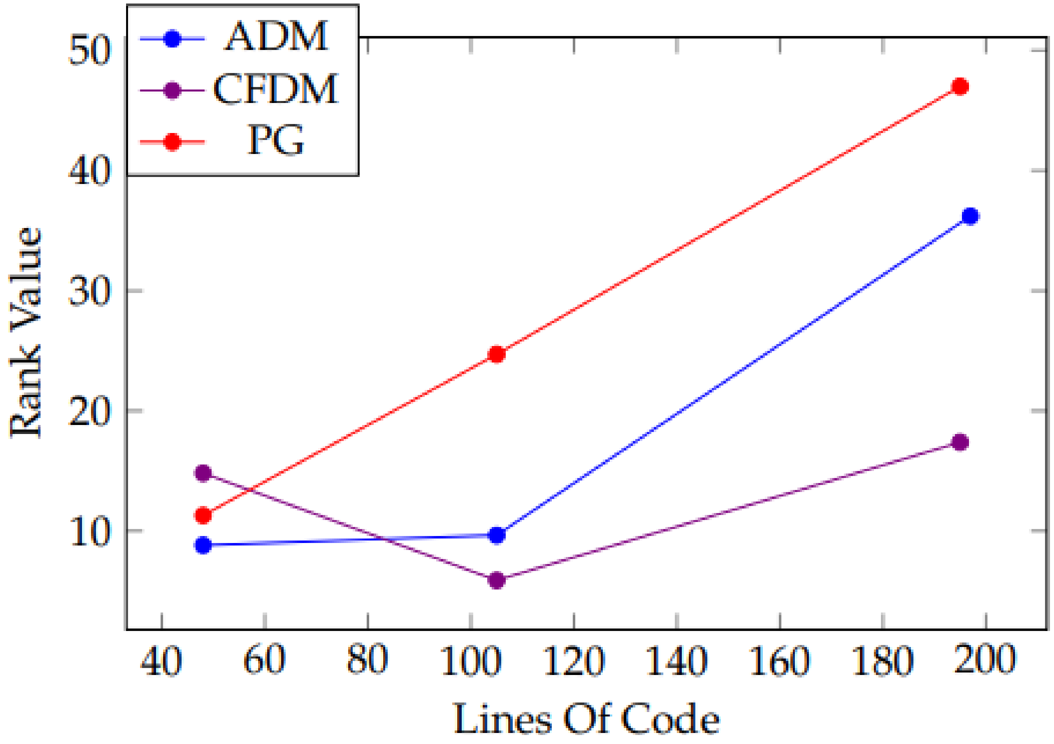
$$Rank\colon [0,1] \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{R}$$

$$\mathrm{Rank}(successRate, Hotlines, \boldsymbol{Differences}) = successRate * \frac{Hotlines}{\boldsymbol{Differences}}$$

Differences >>> the number of differences between the counterexample and the the closest to the counterexample found successful run


ADM
CFDM
PG
Rank Value
10
20
30
40
50
Lines Of Code
40
60
80
100
120
140
160
180
200

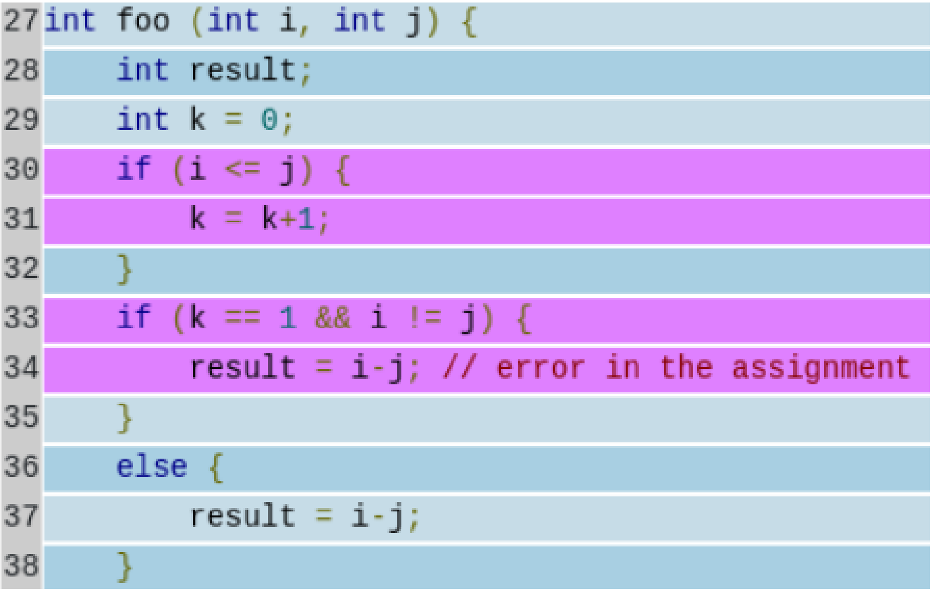
Ranking



Ranking

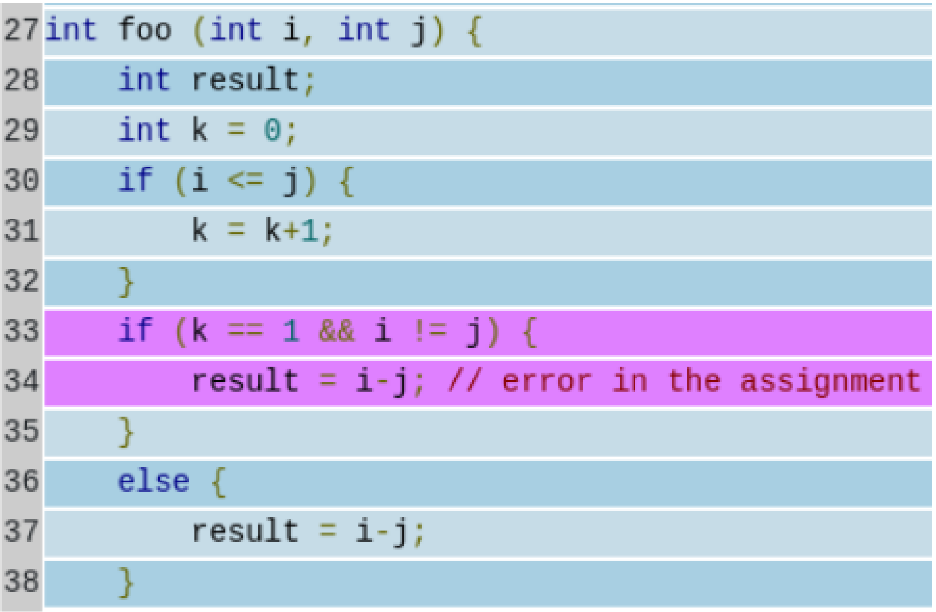
Path Generation has the lead because it gives a better explanation about the fault

Evaluation



```
int foo (int i, int j) {
    int result;
    int k = 0;
    if (i <= j) {
        k = k+1;
    }
    if (k == 1 && i != j) {
        result = i-j; // error in the assignment
    }
    else {
        result = i-j;
    }
```

Suspicious lines using ABD



```
int foo (int i, int j) {
    int result;
    int k = 0;
    if (i <= j) {
        k = k+1;
    }
    if (k == 1 && i != j) {
        result = i-j; // error in the assignment
    }
    else {
        result = i-j;
    }
```

Suspicious lines using Path Generation

Future Work & Conclusion

Future Work

- Use of differently structured distance metrics
    - ❑SSA-based distance metrics

- Combine distance metrics with another fault localization technique:
    - ❑Distance Metric finds the closest successful run
    - ❑Tarantula locates the exact position of the fault

Conclusion

- Distance Metrics for fault-localization purposes can be a great assistance to the developer

Conclusion

- Distance Metrics for fault-localization purposes can be a great assistance to the developer
- Big programs ⋙ huge number of safe paths

  ⋙ slow execution time

Conclusion

- Automated Path Generation technique is the least promising to find the fault

Conclusion

- Automated Path Generation technique is the least promising to find the fault
- PG is fastest out of all three and if it finds the fault, it produces a much better explanation

Thank you for your attention!

Q & A

References

[1] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In R. N. Taylor and M. B. Dwyer, editors, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, pages 73–82. ACM, 2004.

[2] A. Groce. Error explanation with distance metrics. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2004.

[3] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 347–351. ACM, 2005.

[4] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.