A tale about software profiling, debugging, testing, and visualization

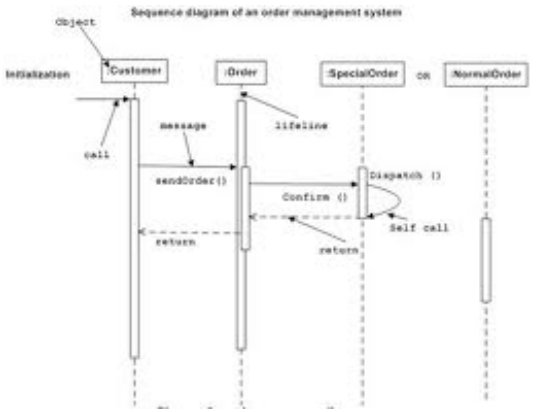
Alexandre Bergel et al.
University of Chile & Object Profile

bergel.eu
objectprofile.com

OBJECT PROFILE

Three-polar identity disorder

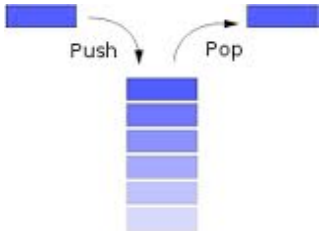
```go
// explode splits s into an array of UTF-8 sequences, one per Unicode
// Invalid UTF-8 sequences become correct encodings of U+FFF8.
func explode(s string, n int) []string {
	if n <= 0 {
		n = len(s)
	}
	a := make([]string, n);
	var size, rune int;
	na := 0;
	for len(s) > 0 {
		if na+1 >= n {
			a[na] = s;
			na++;
			break;
		}
		rune, size = utf8.DecodeRuneInString(s);
		s = s[size:len(s)];
		a[na] = string(rune);
		na++;
	}
	return a[0:na];
}
```



textual description

interaction of objects

stack, heap, ...



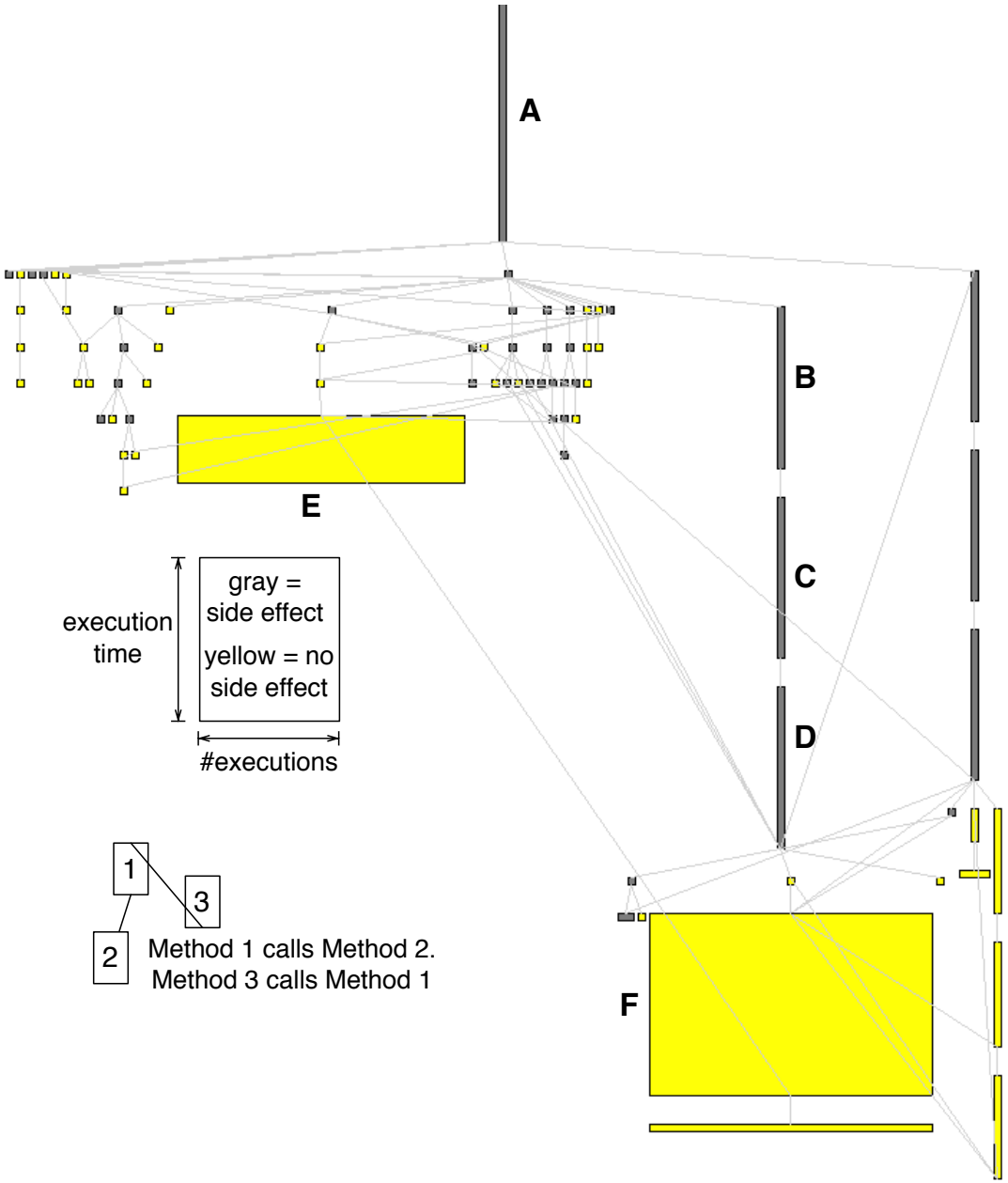
Execution profiling with Kai

Traditional code profilers are driven by the method stack, discarding the notion of sending messages

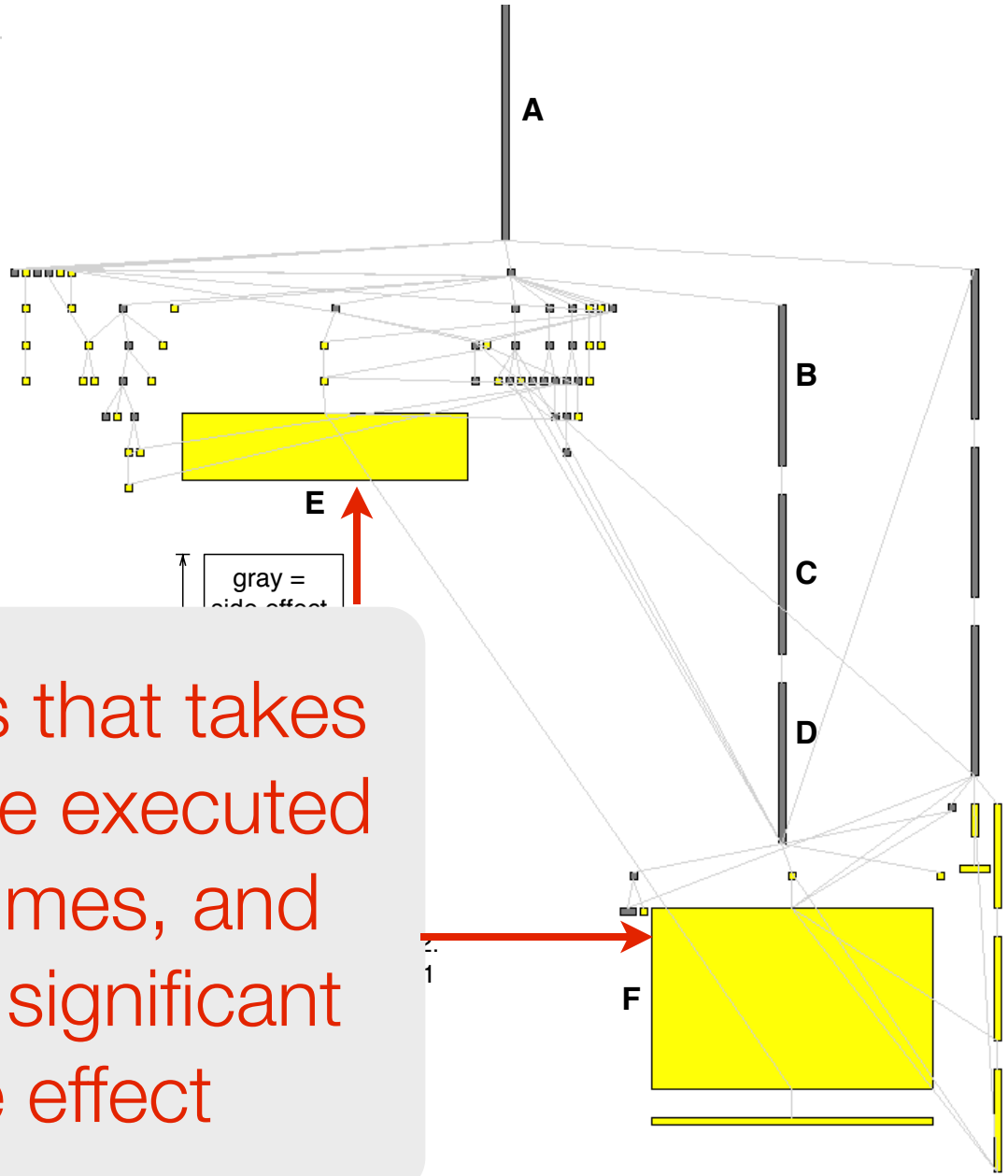
How to answer to "Is there a slow method that is called too often?"

Visually compare *time* and the number of *executions*

Profiling blueprint



Profiling blueprint



methods that takes times, are executed many times, and without significant side effect

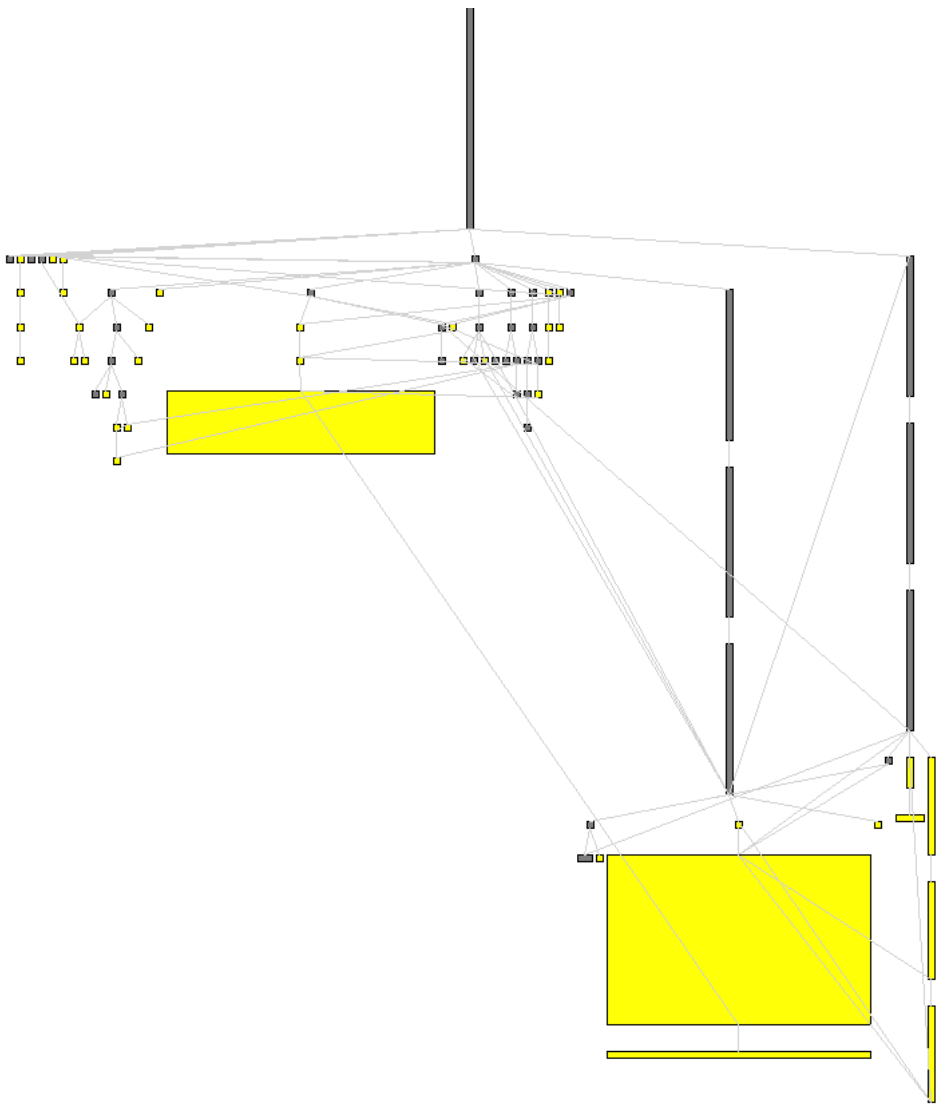
Adding a memoization

```
ROElement>>bounds
  "Return the bounds of the element"


  ^
      self position extent: (shape extentFor:
self)
```

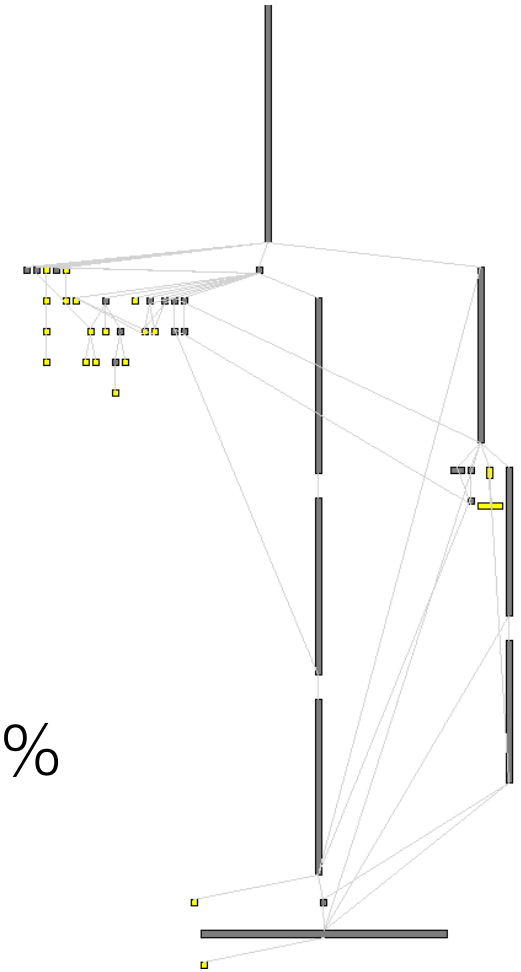
Adding a memoization

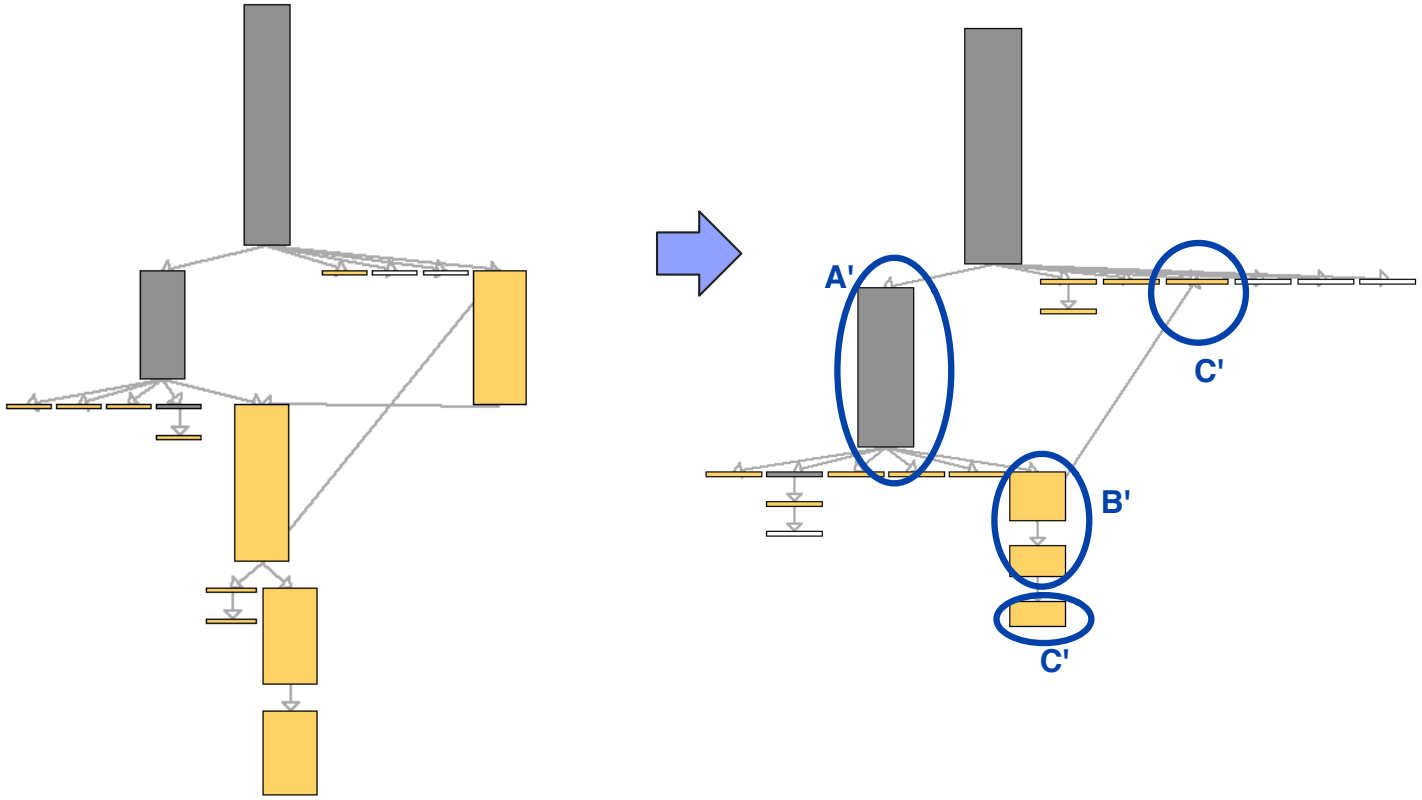
```
ROElement>>bounds
  "Return the bounds of the element"

  boundsCache ifNotNil: [ ^ boundsCache ].

  ^ boundsCache :=
      self position extent: (shape extentFor:
self)
```

Effect of the memoization



speedup 43%

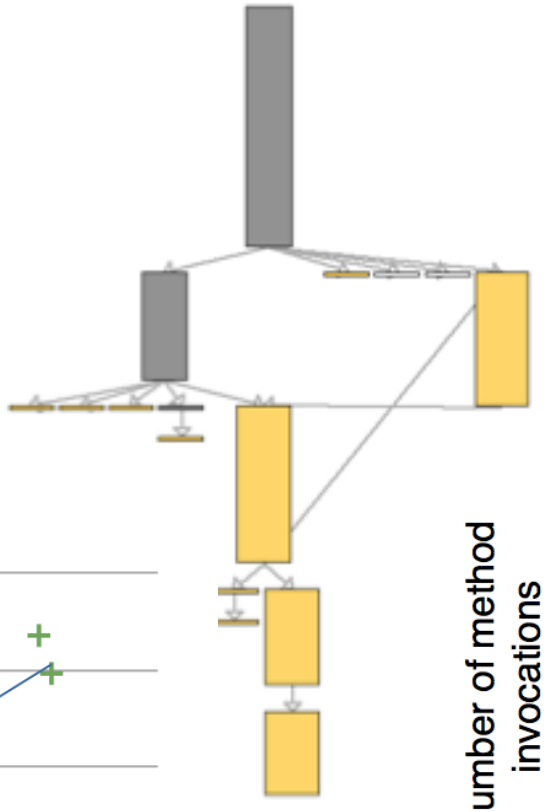


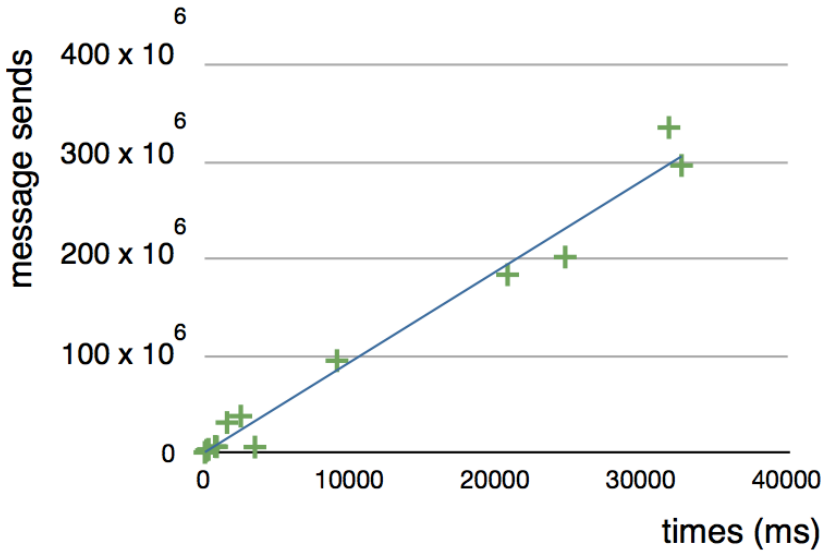

A'
C'
B'
C'

Execution profiling blueprints. *Software: Practices and Experience, 2012*

Visualizing Dynamic Metrics with Profiling Blueprints. *Proceedings of the TOOLS, 2010*

Counting Messages as a Proxy for Average Execution Time in Pharo. *Proceedings of ECOOP, 2011*





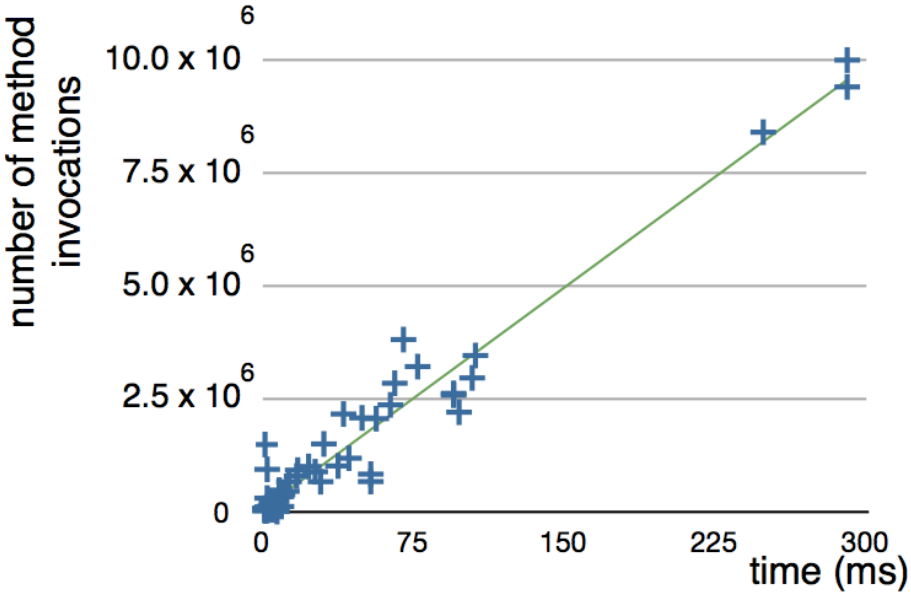
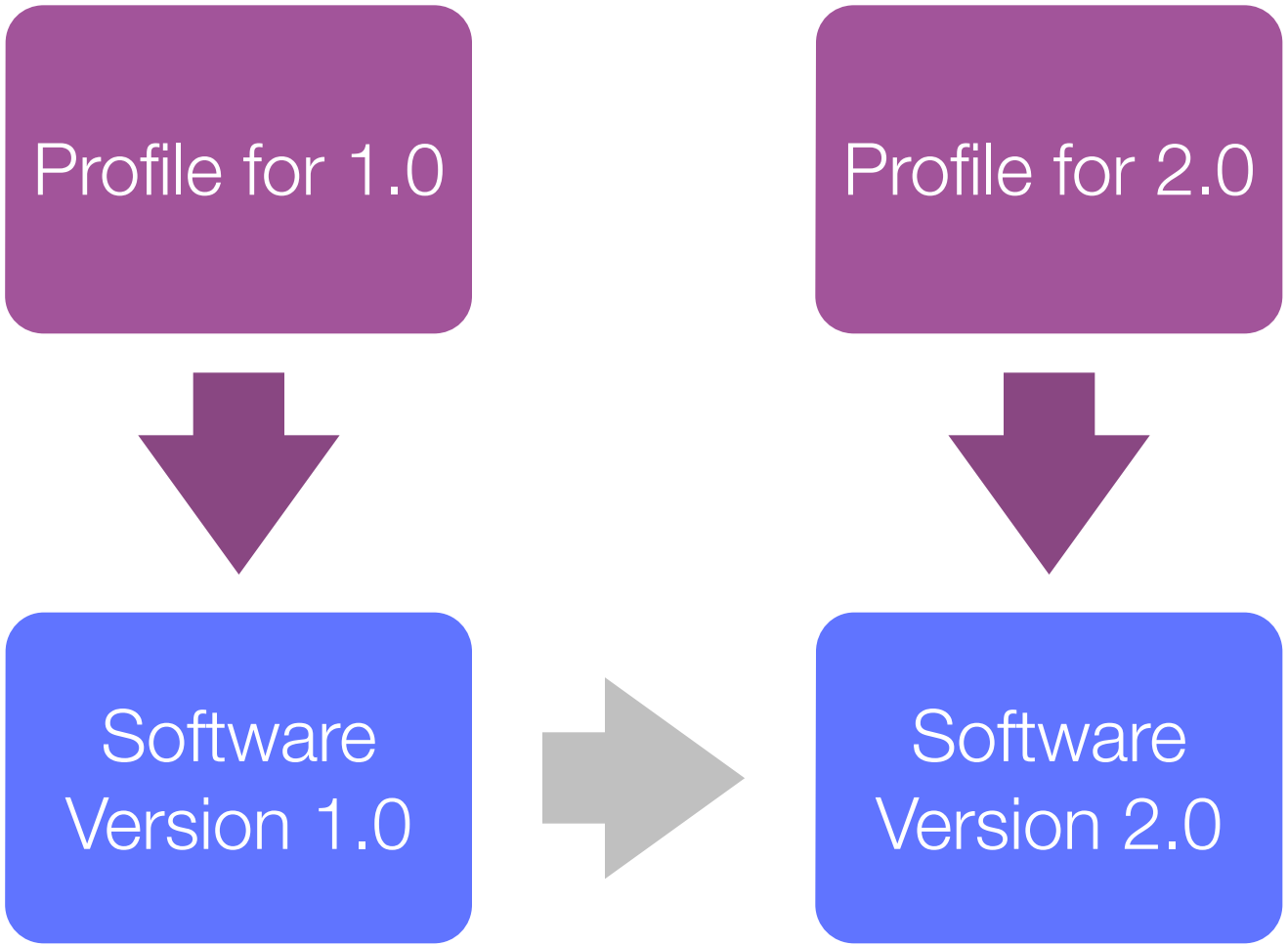


Fig. 1. Linear regression for the 16 Pharo applications.

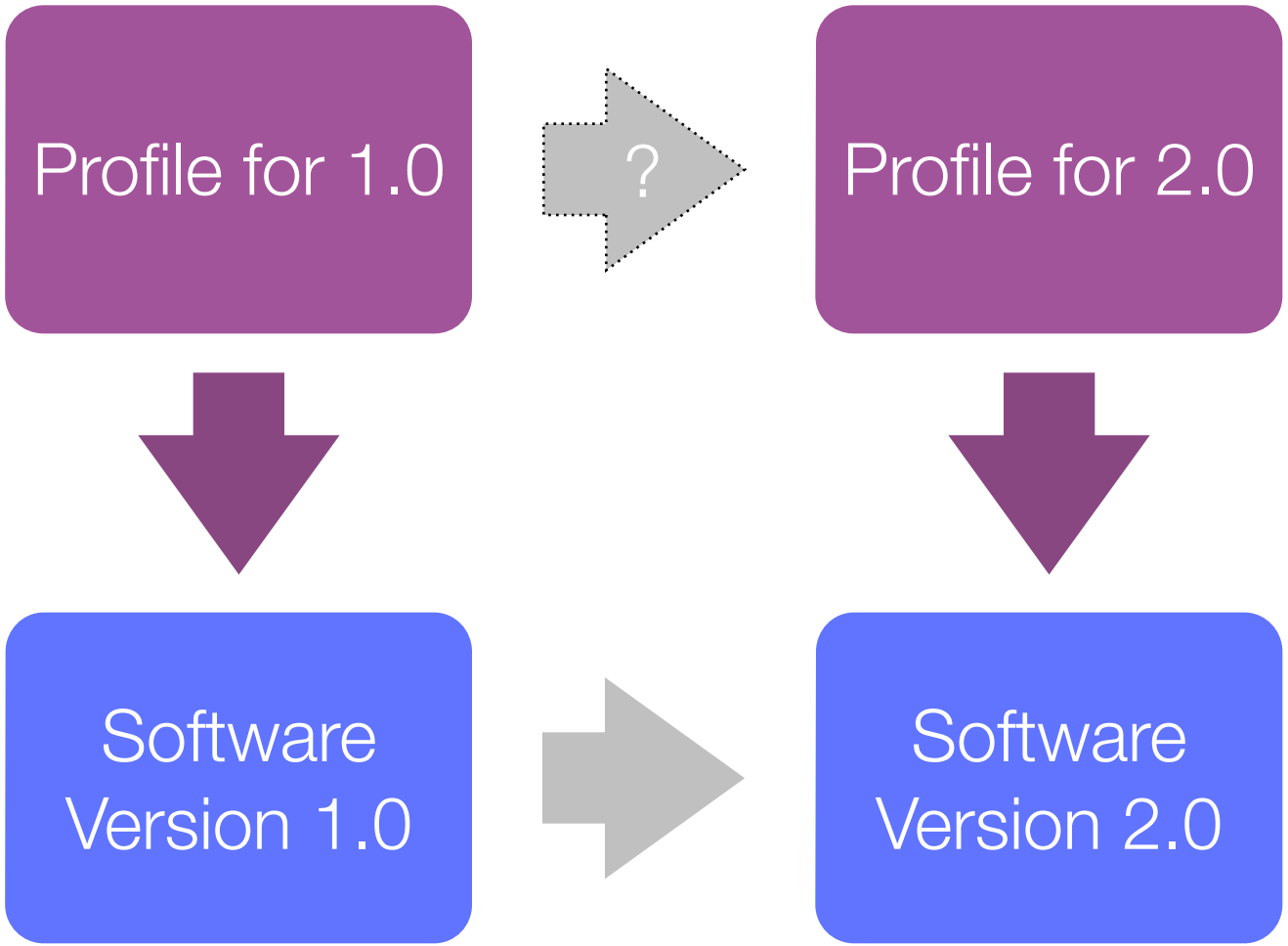
Tracking Performance Evolution

Understanding of performance evolve across multiple software revisions is difficult

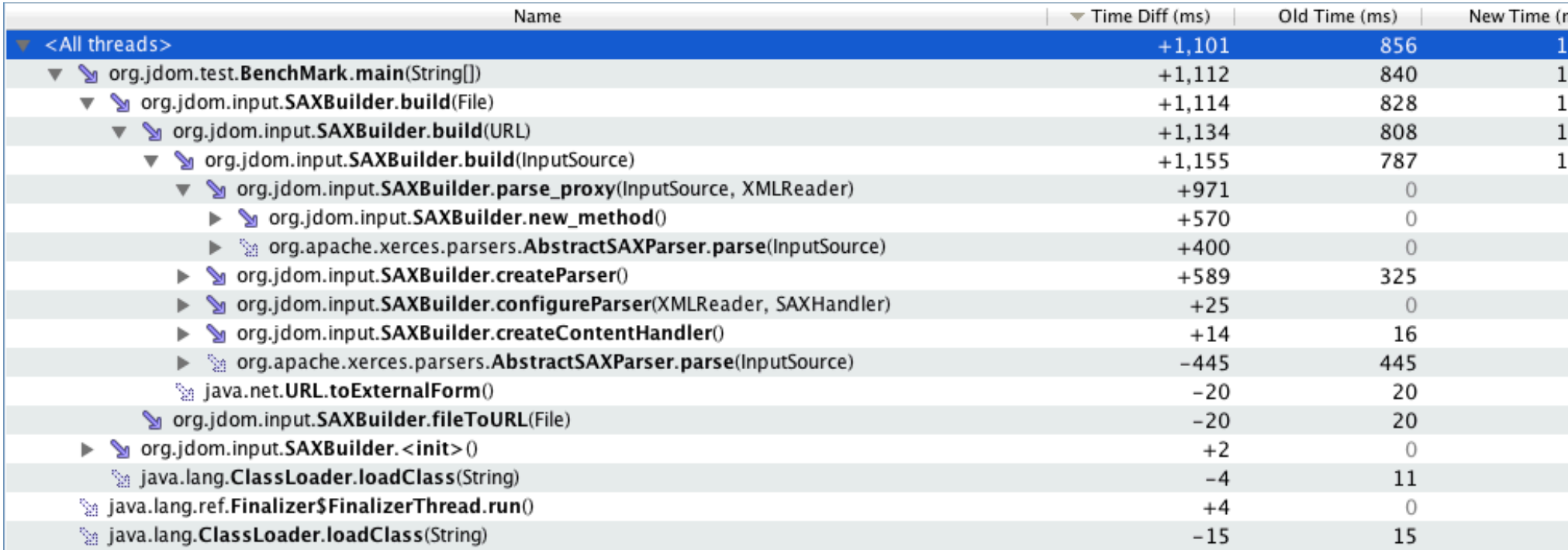
Measuring performance evolution



Measuring performance evolution

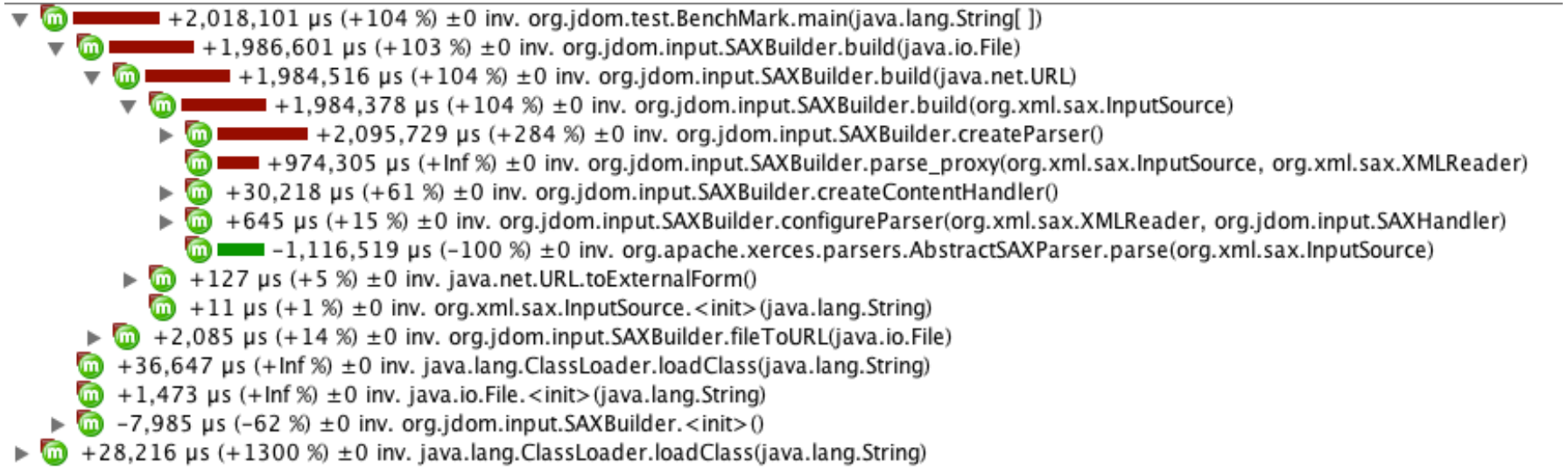


Yourkit



Name	Time Diff (ms)	Old Time (ms)	New Time (r
<All threads>	+1,101	856	1
org.jdom.test.**BenchMark.main**(String[])	+1,112	840	1
org.jdom.input.**SAXBuilder.build**(File)	+1,114	828	1
org.jdom.input.**SAXBuilder.build**(URL)	+1,134	808	1
org.jdom.input.**SAXBuilder.build**(InputSource)	+1,155	787	1
org.jdom.input.**SAXBuilder.parse_proxy**(InputSource, XMLReader)	+971	0	
org.jdom.input.**SAXBuilder.new_method**()	+570	0	
org.apache.xerces.parsers.**AbstractSAXParser.parse**(InputSource)	+400	0	
org.jdom.input.**SAXBuilder.createParser**()	+589	325	
org.jdom.input.**SAXBuilder.configureParser**(XMLReader, SAXHandler)	+25	0	
org.jdom.input.**SAXBuilder.createContentHandler**()	+14	16	
org.apache.xerces.parsers.**AbstractSAXParser.parse**(InputSource)	-445	445	
java.net.**URL.toExternalForm**()	-20	20	
org.jdom.input.**SAXBuilder.fileToURL**(File)	-20	20	
org.jdom.input.**SAXBuilder.<init>**()	+2	0	
java.lang.**ClassLoader.loadClass**(String)	-4	11	
java.lang.ref.**Finalizer$FinalizerThread.run**()	+4	0	
java.lang.**ClassLoader.loadClass**(String)	-15	15	

JProfiler


+2,018,101 µs (+104 %) ±0 inv. org.jdom.test.BenchMark.main(java.lang.String[ ])
+1,986,601 µs (+103 %) ±0 inv. org.jdom.input.SAXBuilder.build(java.io.File)
+1,984,516 µs (+104 %) ±0 inv. org.jdom.input.SAXBuilder.build(java.net.URL)
+1,984,378 µs (+104 %) ±0 inv. org.jdom.input.SAXBuilder.build(org.xml.sax.InputSource)
+2,095,729 µs (+284 %) ±0 inv. org.jdom.input.SAXBuilder.createParser()
+974,305 µs (+Inf %) ±0 inv. org.jdom.input.SAXBuilder.parse_proxy(org.xml.sax.InputSource, org.xml.sax.XMLReader)
+30,218 µs (+61 %) ±0 inv. org.jdom.input.SAXBuilder.createContentHandler()
+645 µs (+15 %) ±0 inv. org.jdom.input.SAXBuilder.configureParser(org.xml.sax.XMLReader, org.jdom.input.SAXHandler)
-1,116,519 µs (-100 %) ±0 inv. org.apache.xerces.parsers.AbstractSAXParser.parse(org.xml.sax.InputSource)
+127 µs (+5 %) ±0 inv. java.net.URL.toExternalForm()
+11 µs (+1 %) ±0 inv. org.xml.sax.InputSource.<init>(java.lang.String)
+2,085 µs (+14 %) ±0 inv. org.jdom.input.SAXBuilder.fileToURL(java.io.File)
+36,647 µs (+Inf %) ±0 inv. java.lang.ClassLoader.loadClass(java.lang.String)
+1,473 µs (+Inf %) ±0 inv. java.io.File.<init>(java.lang.String)
-7,985 µs (-62 %) ±0 inv. org.jdom.input.SAXBuilder.<init>()
+28,216 µs (+1300 %) ±0 inv. java.lang.ClassLoader.loadClass(java.lang.String)

Measuring performance evolution

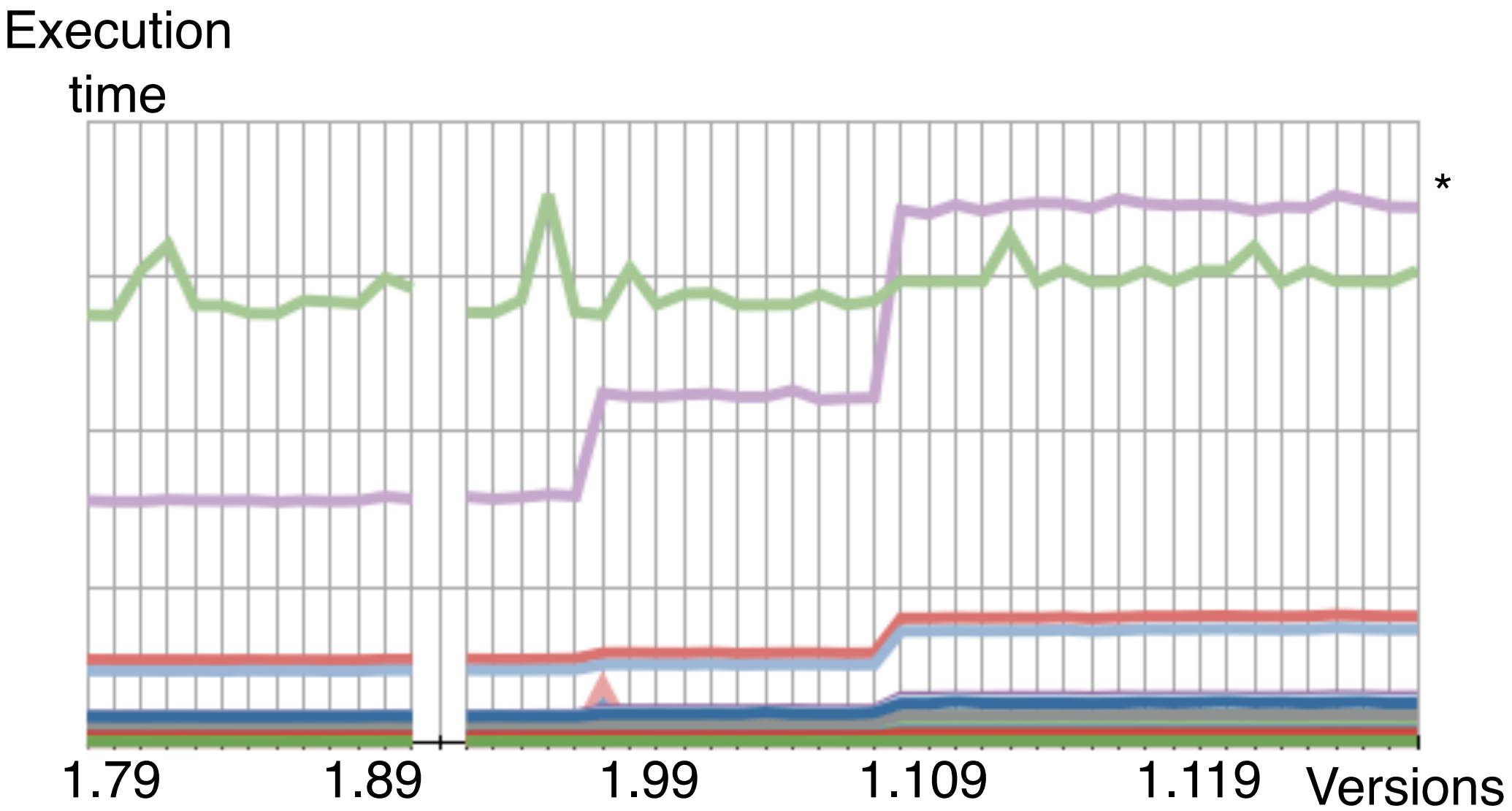
Variation have to be manually tracked

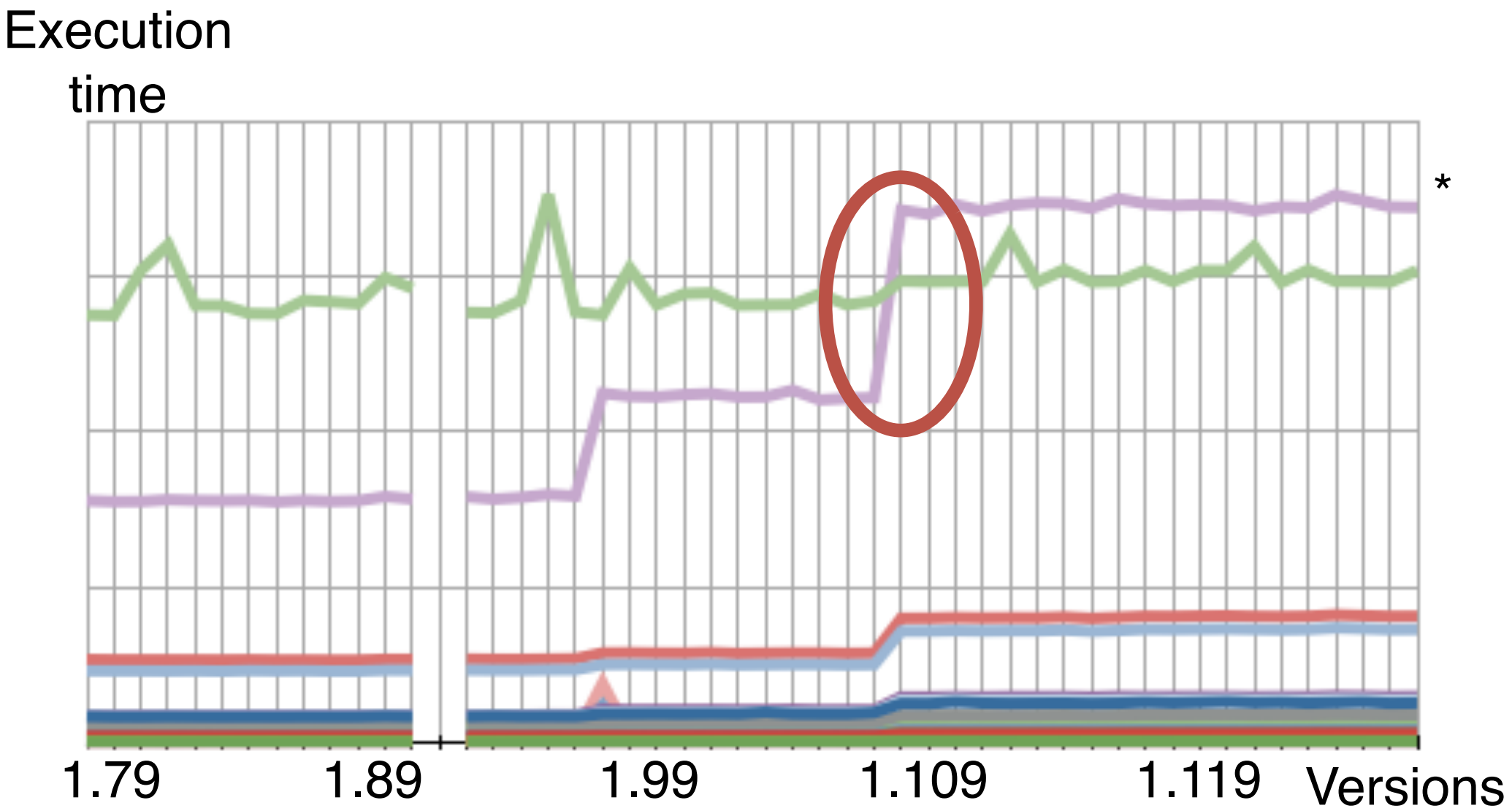
Relevant metrics are missing

Poor visual representation

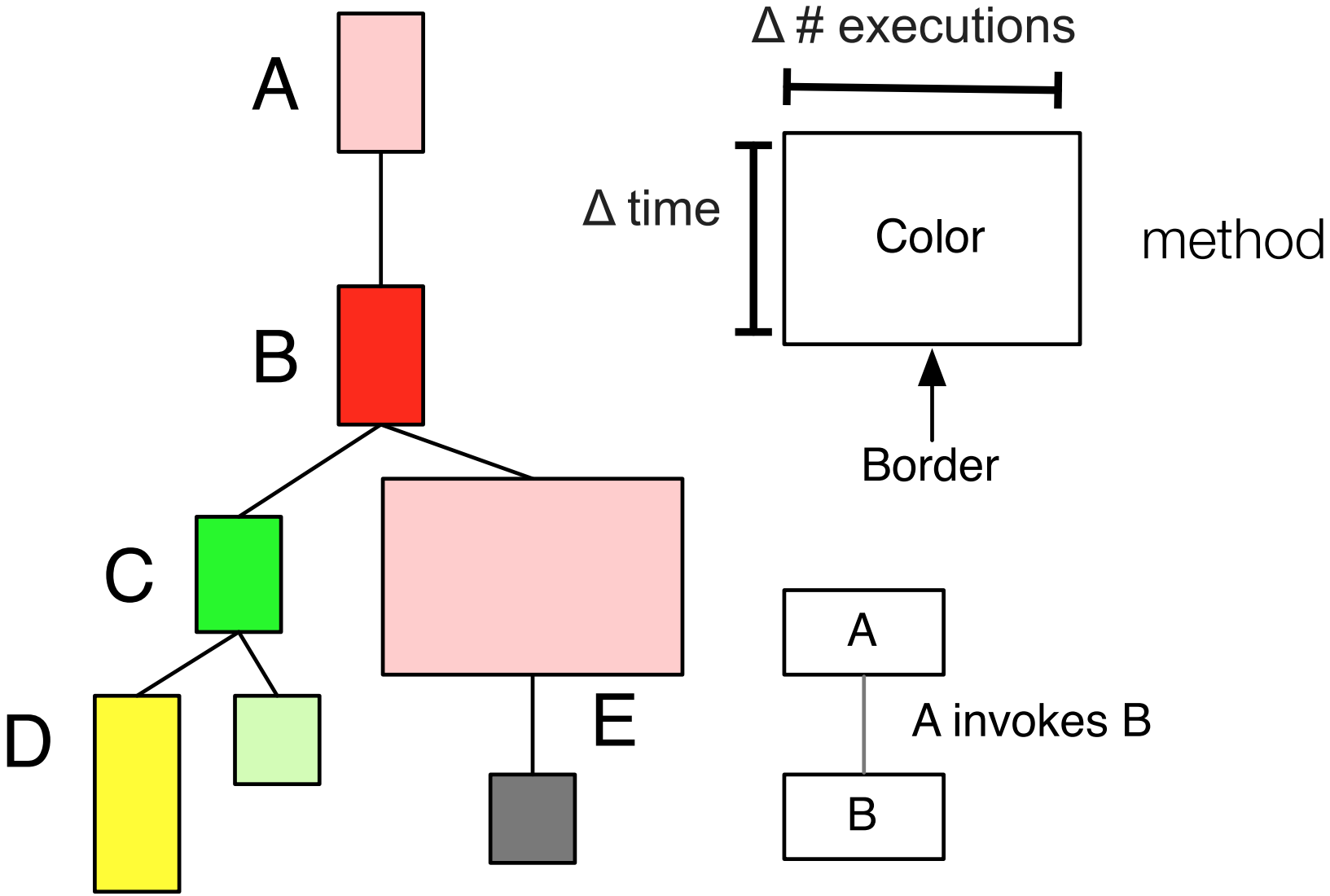
Framework to measure evolution

```
data := Rizel new
          setTestsAsBenchmarks;
          trackLast: 100 versionsOf: 'Roassal';
          run.


data export...
```

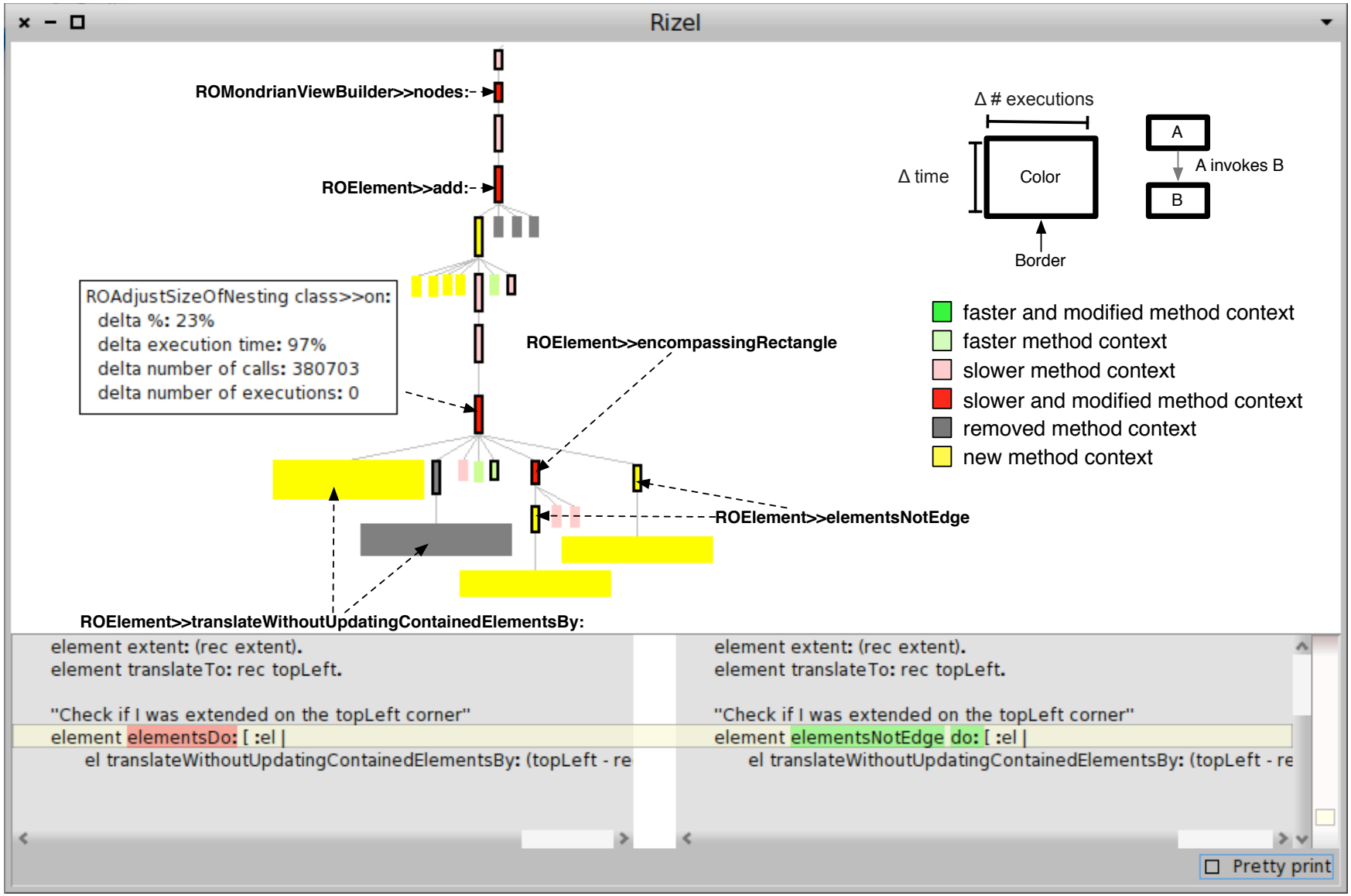

Execution
time
*
1.79
1.89
1.99
1.109
1.119
Versions


Execution time
*
1.79
1.89
1.99
1.109
1.119
Versions

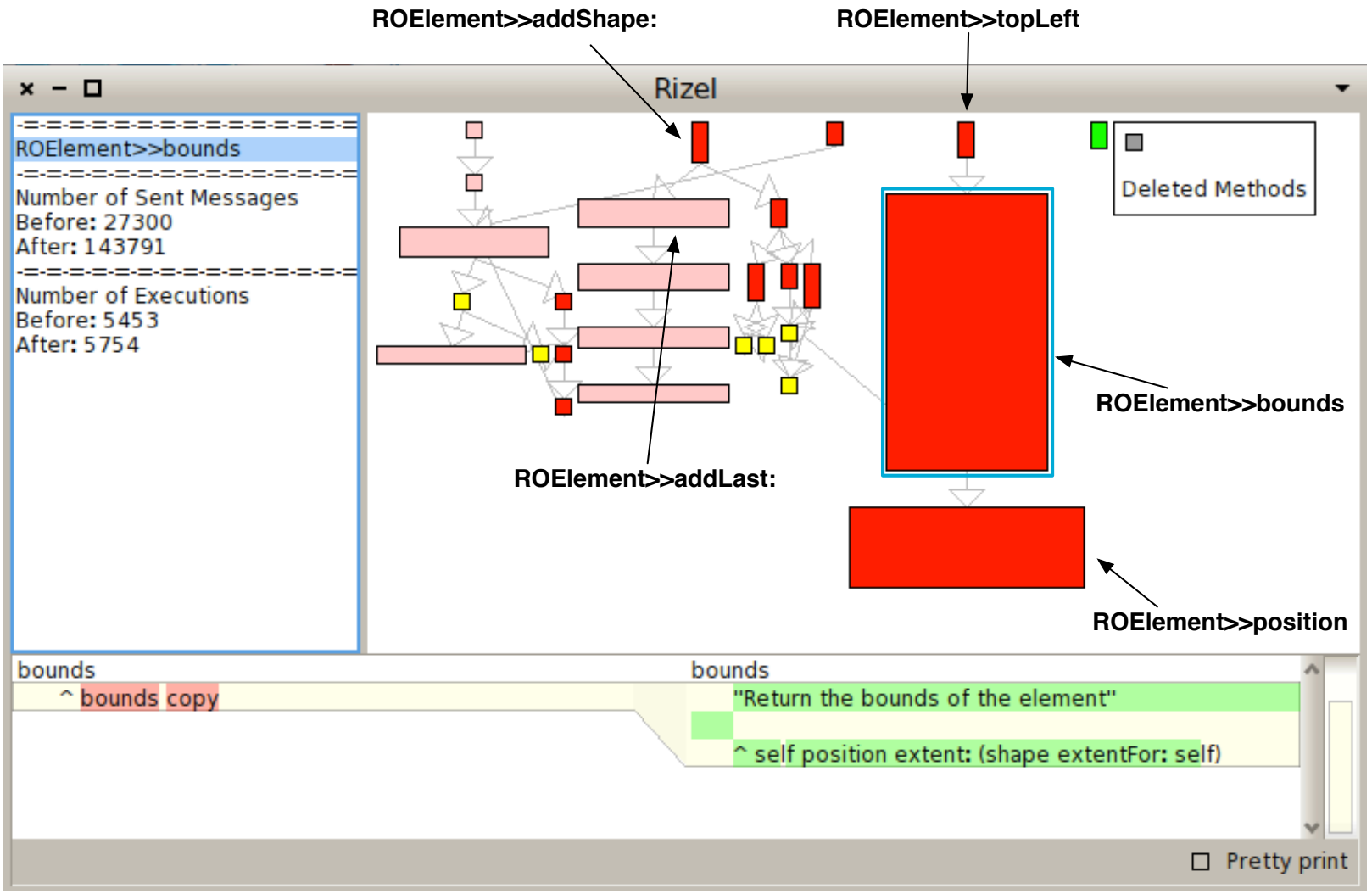
Performance Evolution Blueprint



Rizel in action



Rizel in action



Test coverage with Hapao

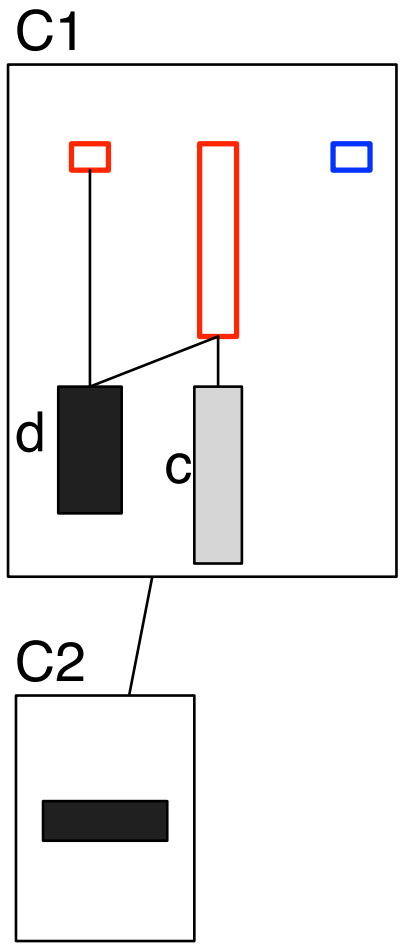
Traditional code coverage *tools* have a *binary view* of the world

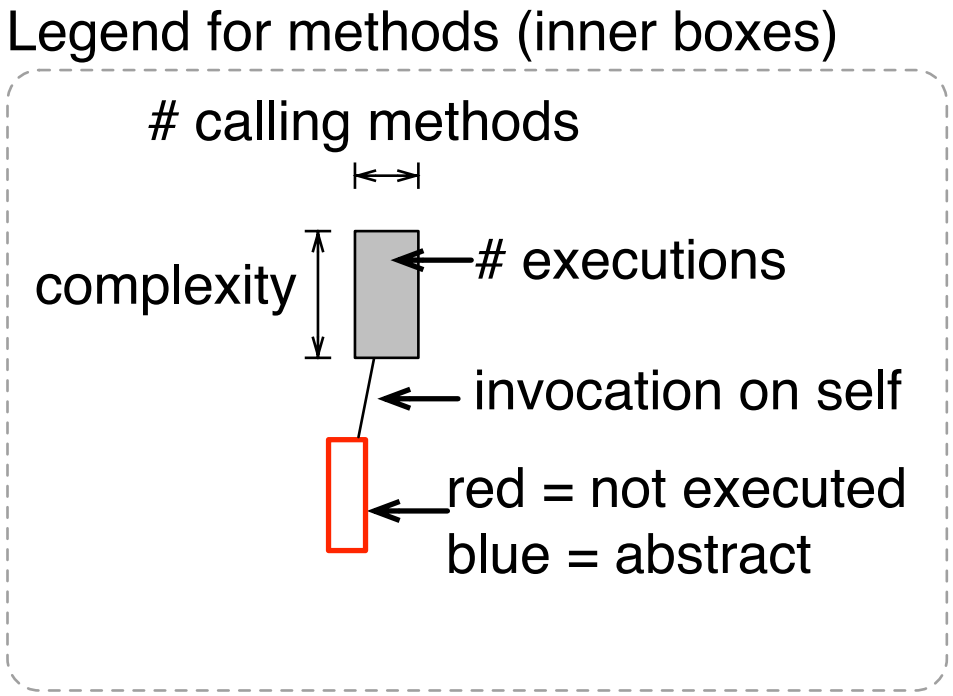
Is my code well covered or not?

Which method should you test first in order to increase the coverage?

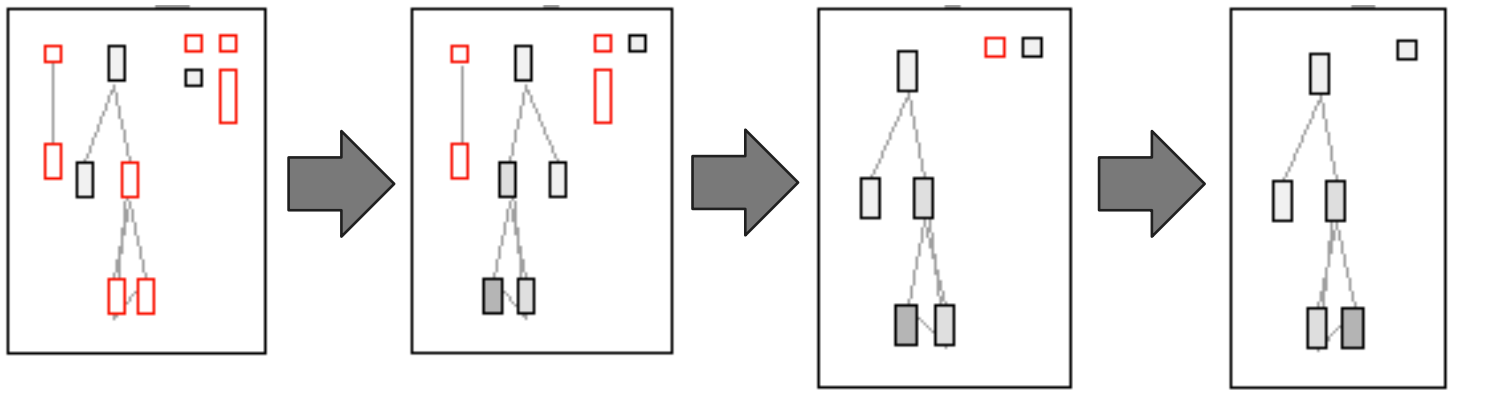
Visual *qualitative assessment* of the coverage

Test blueprint





Successive improvement

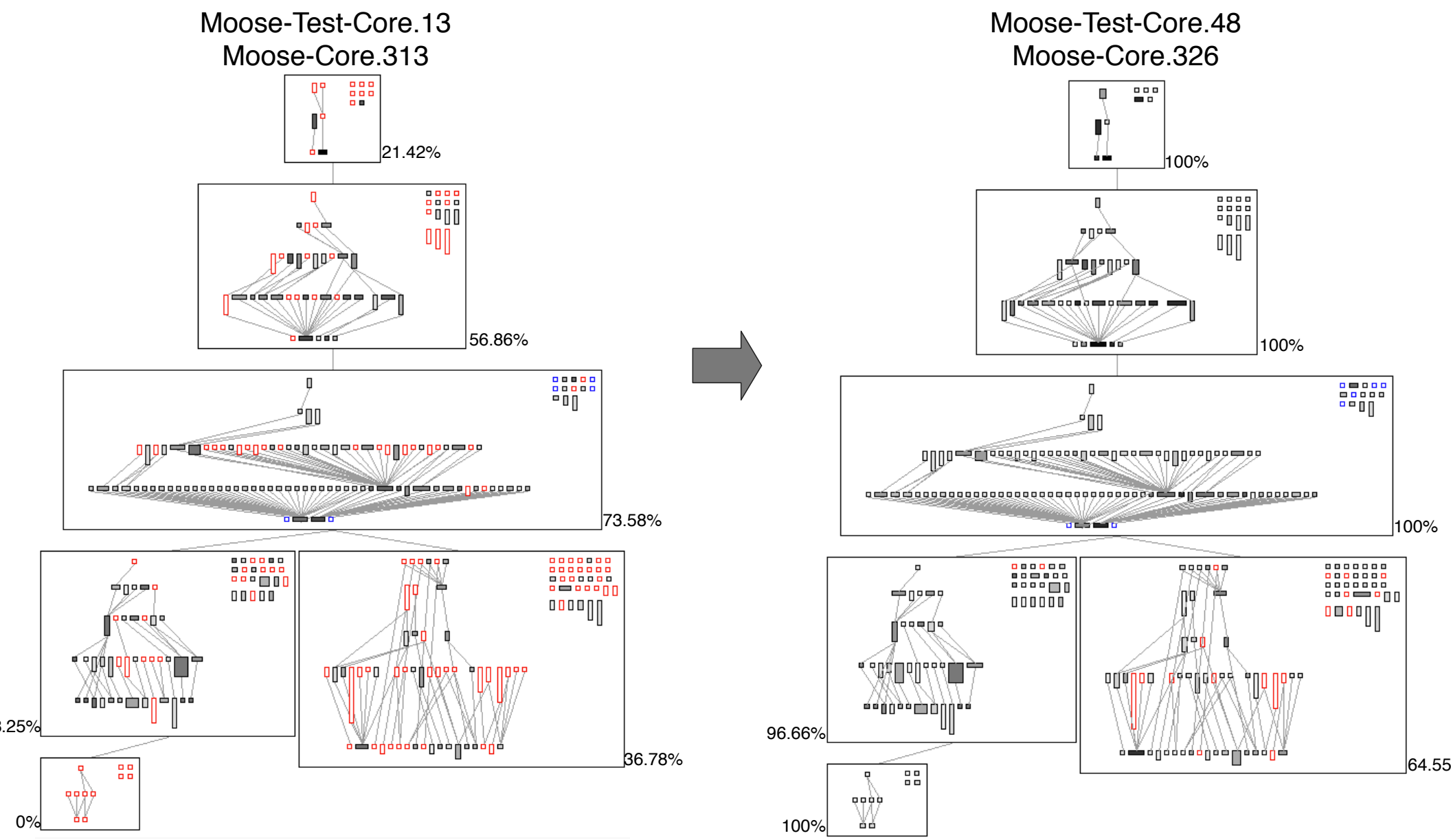


Version 2.2
27.27%

Version 2.3
54.54%

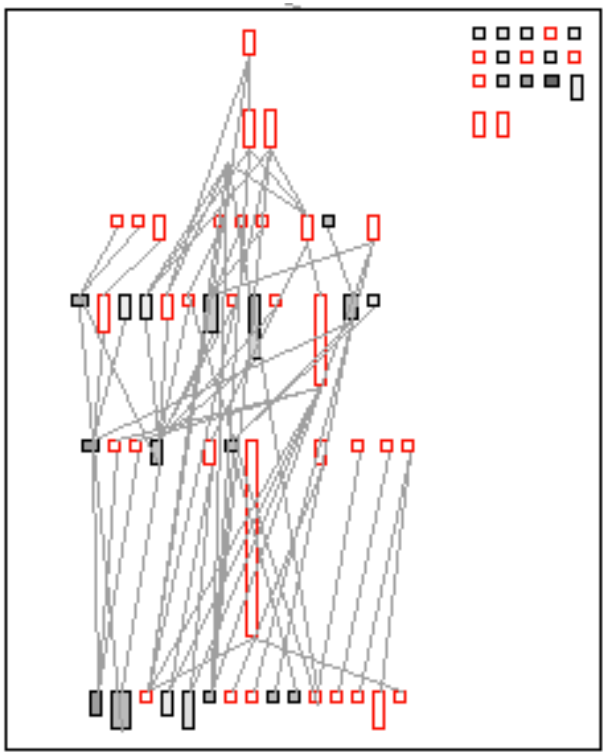
Version 2.4
87.71%

Version 2.5
100%

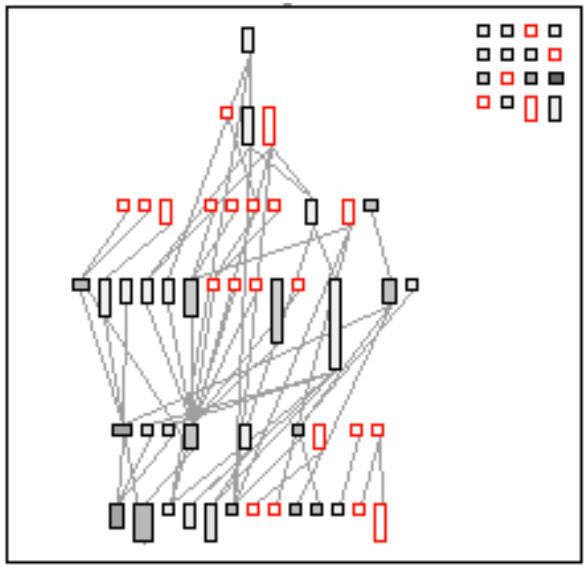

Moose-Test-Core.13
Moose-Core.313
21.42%
56.86%
73.58%
.25%
36.78%
0%
Moose-Test-Core.48
Moose-Core.326
100%
100%
100%
96.66%
64.55
100%

Reducing code complexity

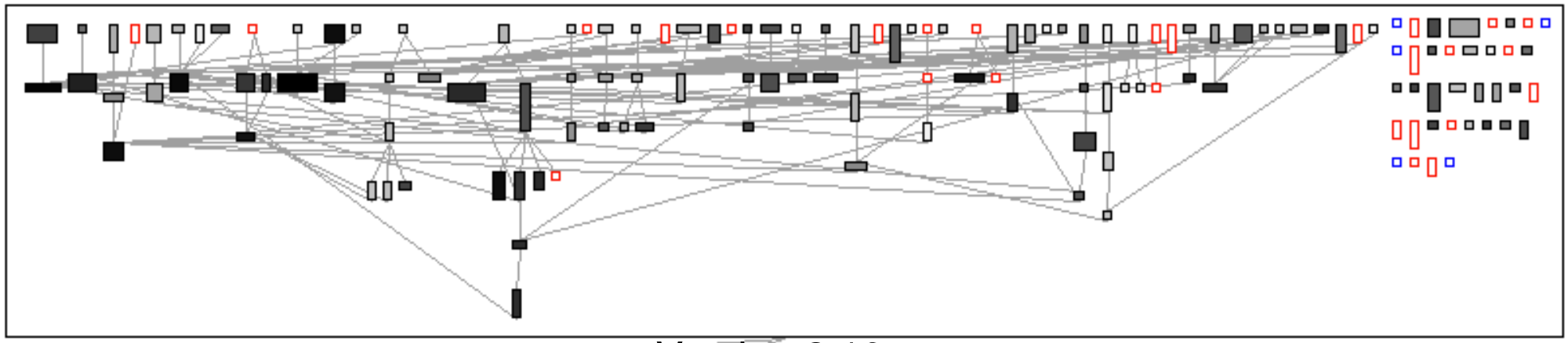
Version 1.58.1
Coverage: 40.57%



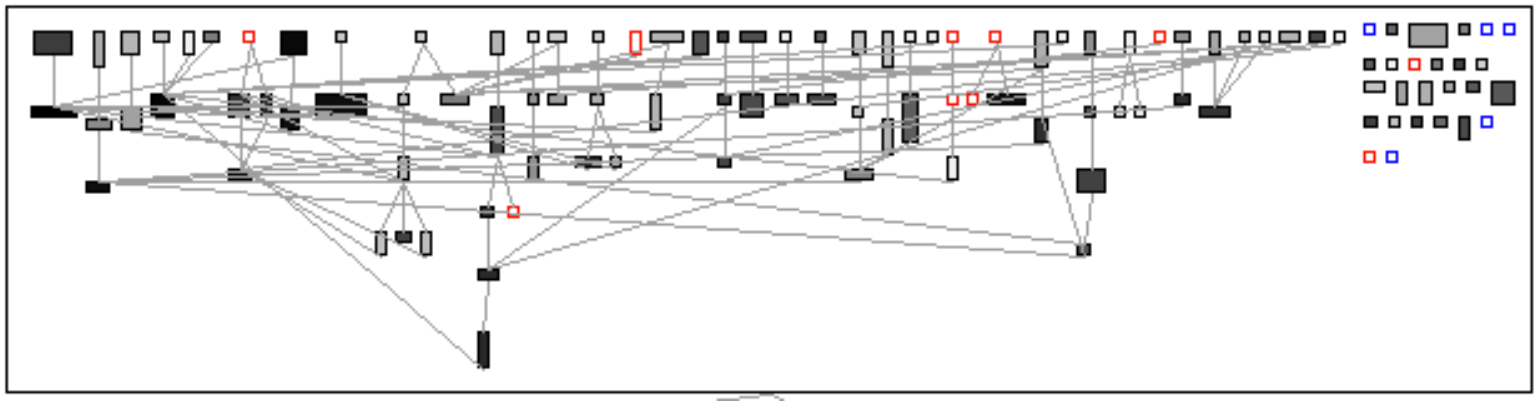
Version 1.58.9
Coverage: 60.60%



Reducing code complexity

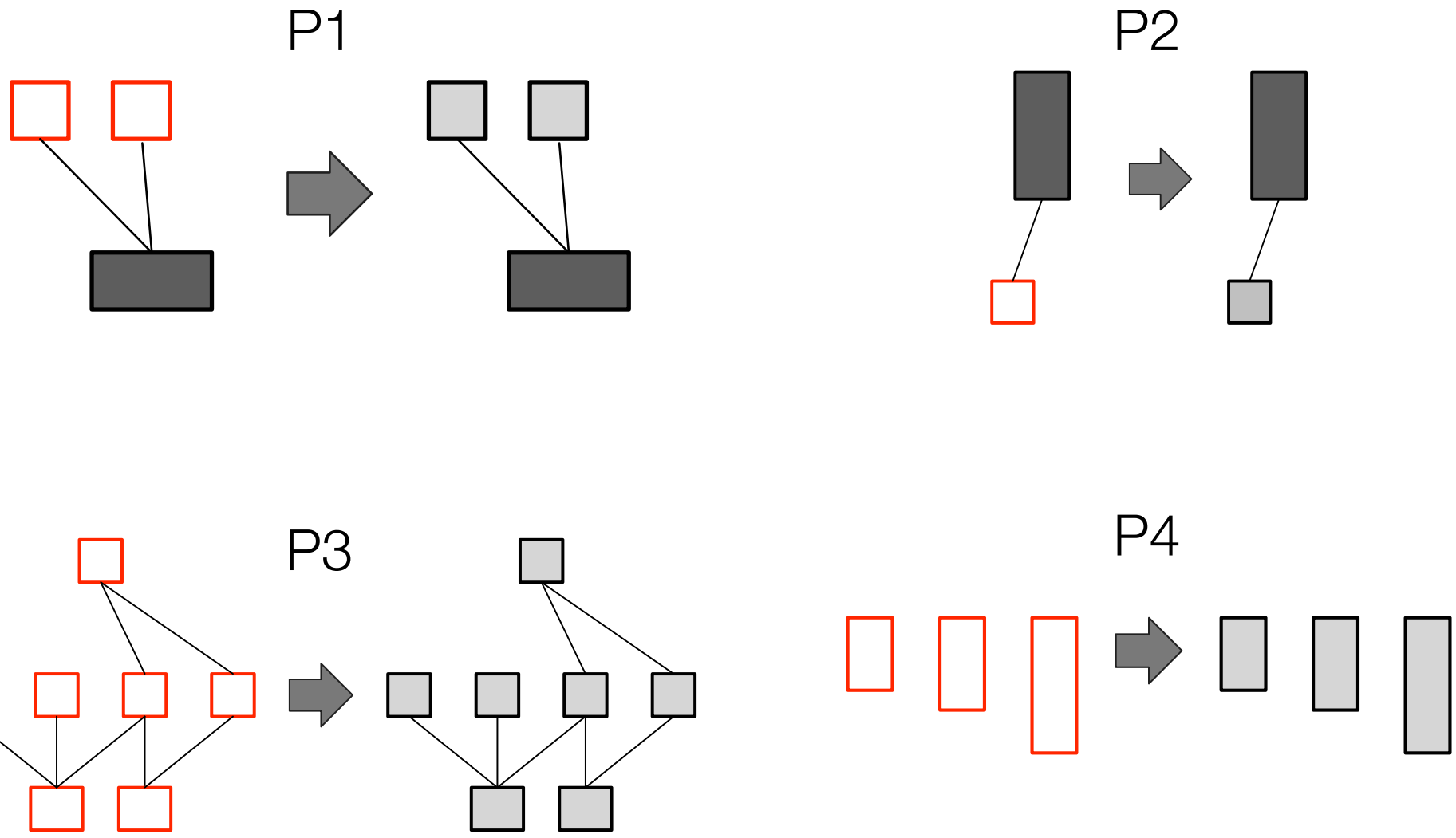


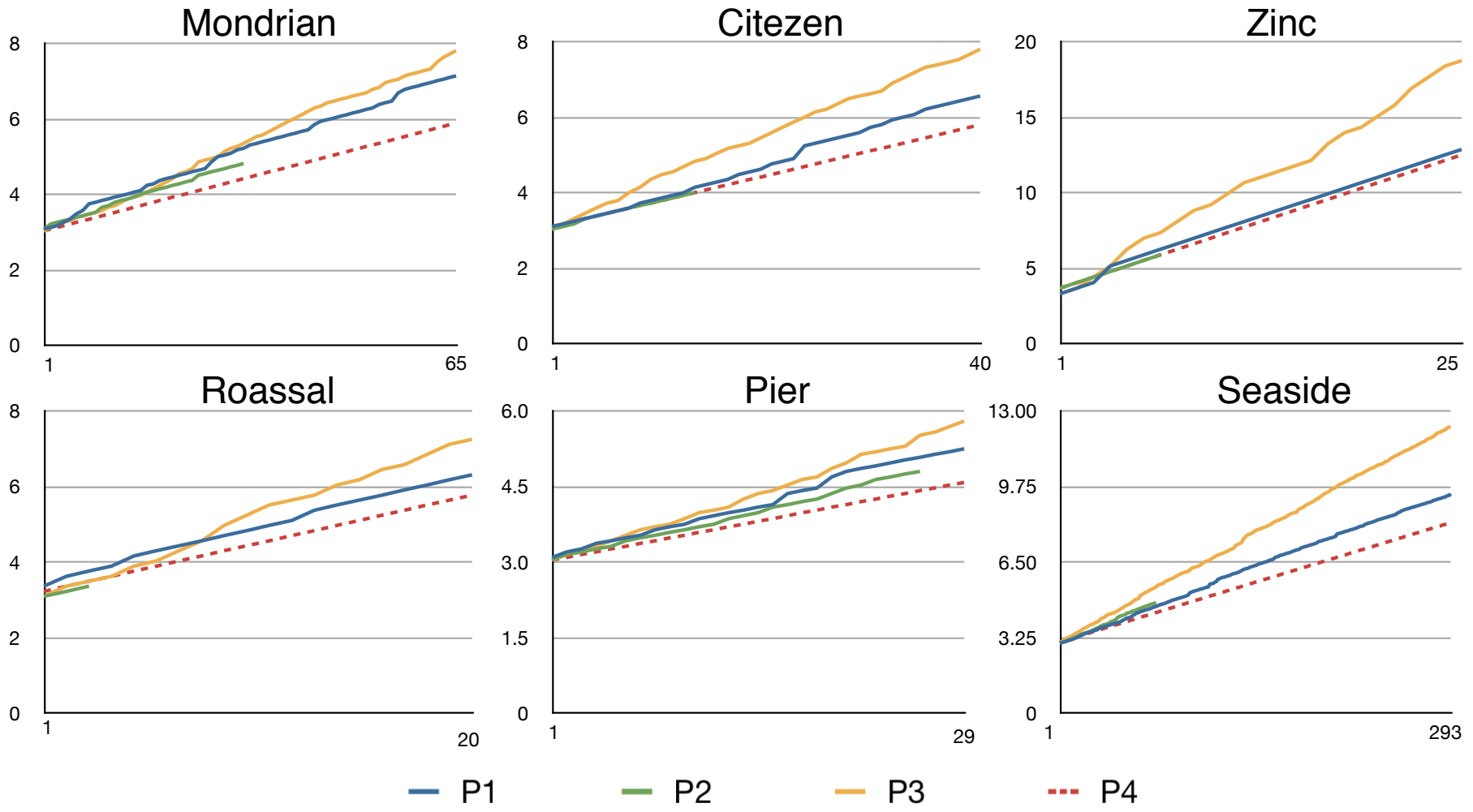
Version 2.10

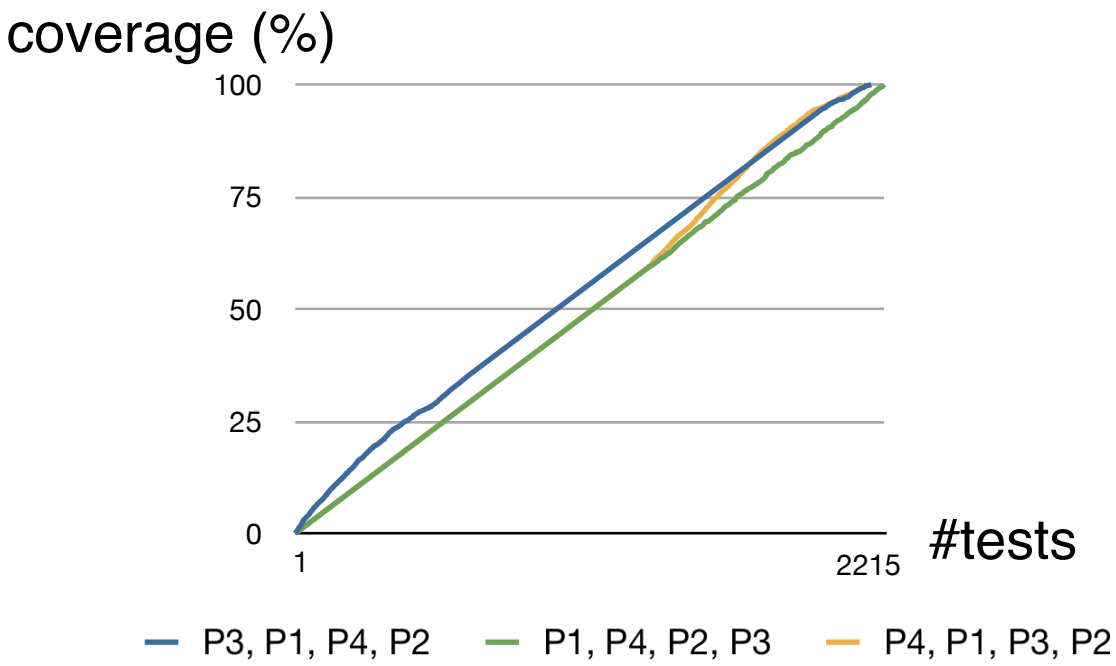


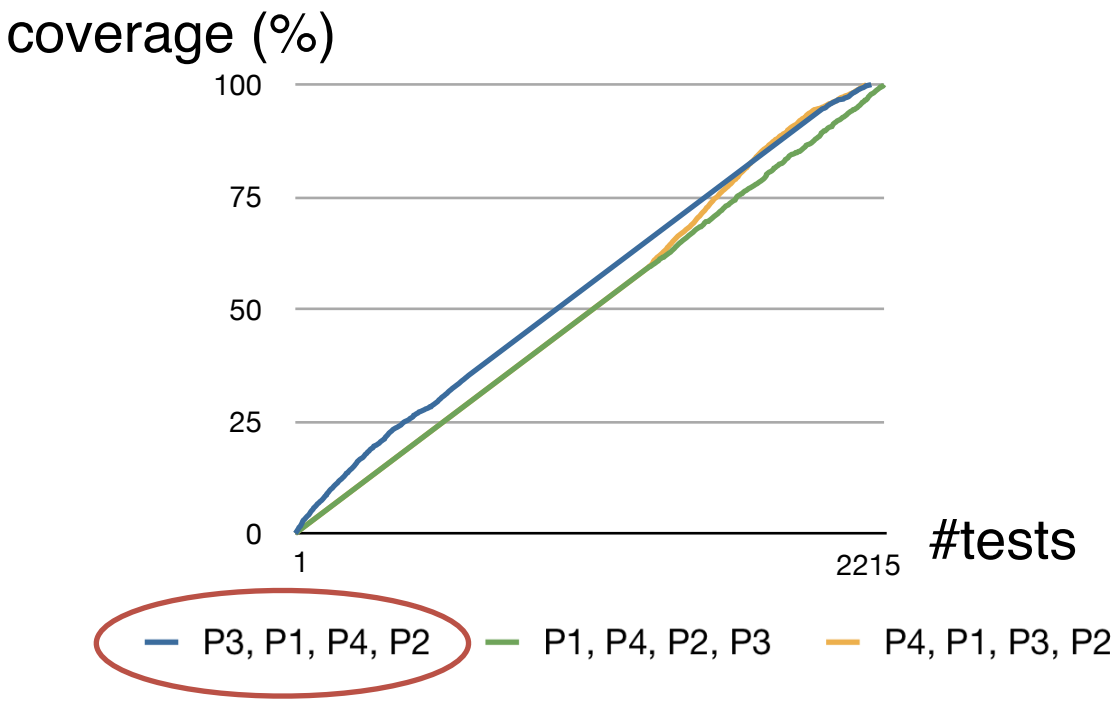
Version 2.17

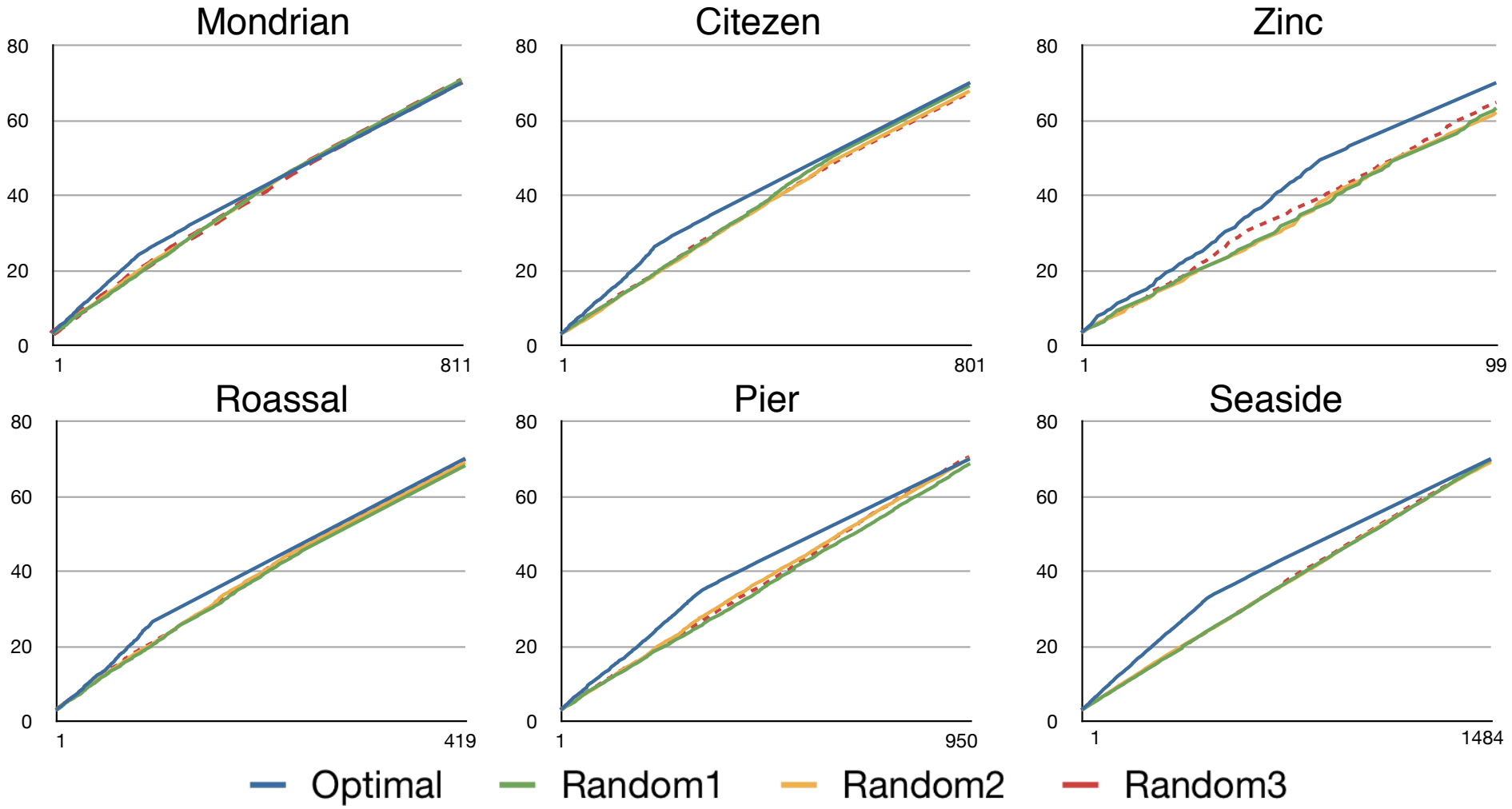
4 patterns



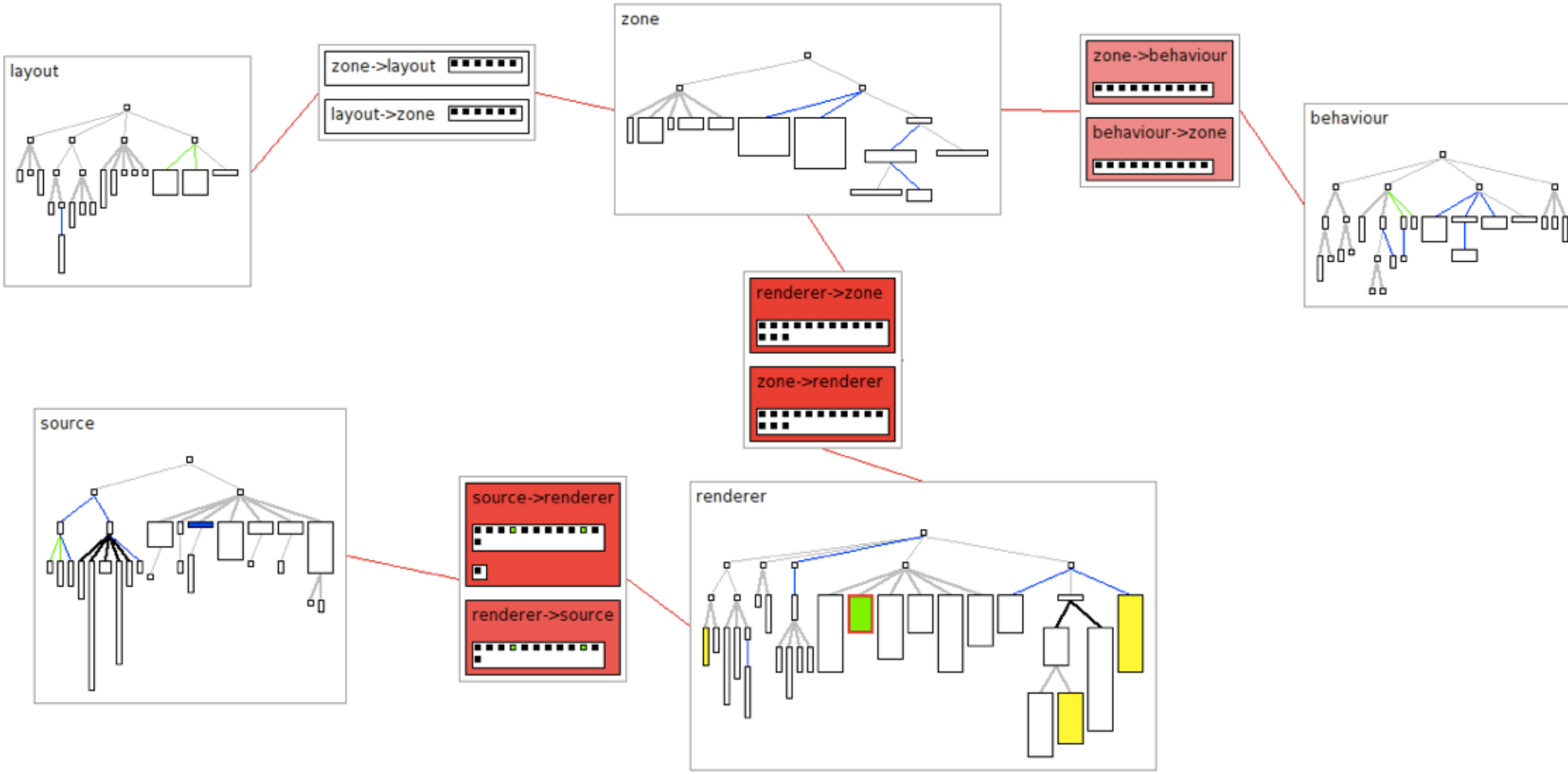

Mondrian
8
6
4
2
0
1
65
Citezen
8
6
4
2
0
1
40
Zinc
20
15
10
5
0
1
25
Roassal
8
6
4
2
0
1
20
Pier
6.0
4.5
3.0
1.5
0
1
29
Seaside
13.00
9.75
6.50
3.25
0
1
293
P1
P2
P3
P4

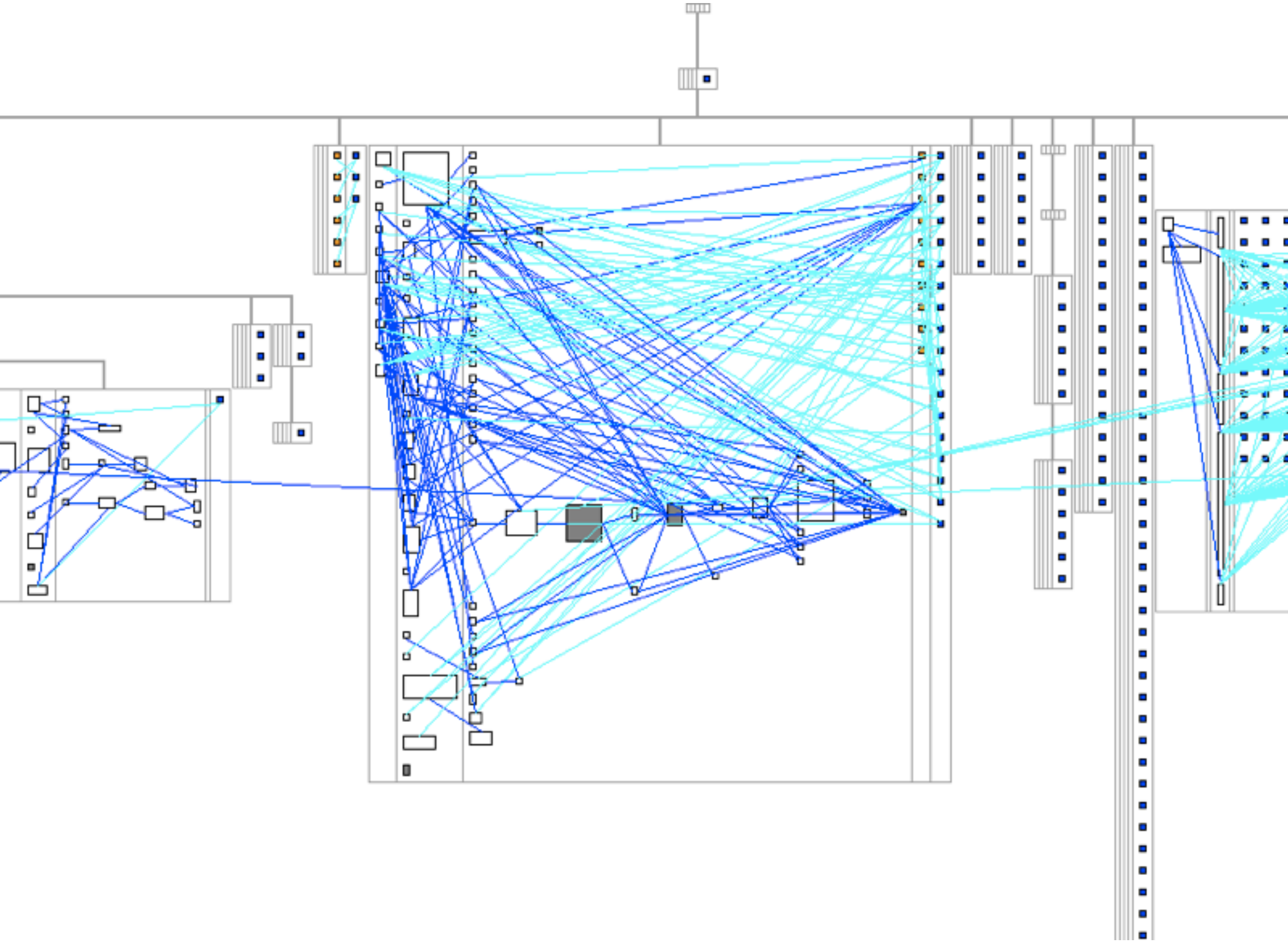

coverage (%)
100
75
50
25
0
1
2215
#tests
P3, P1, P4, P2
P1, P4, P2, P3
P4, P1, P3, P2

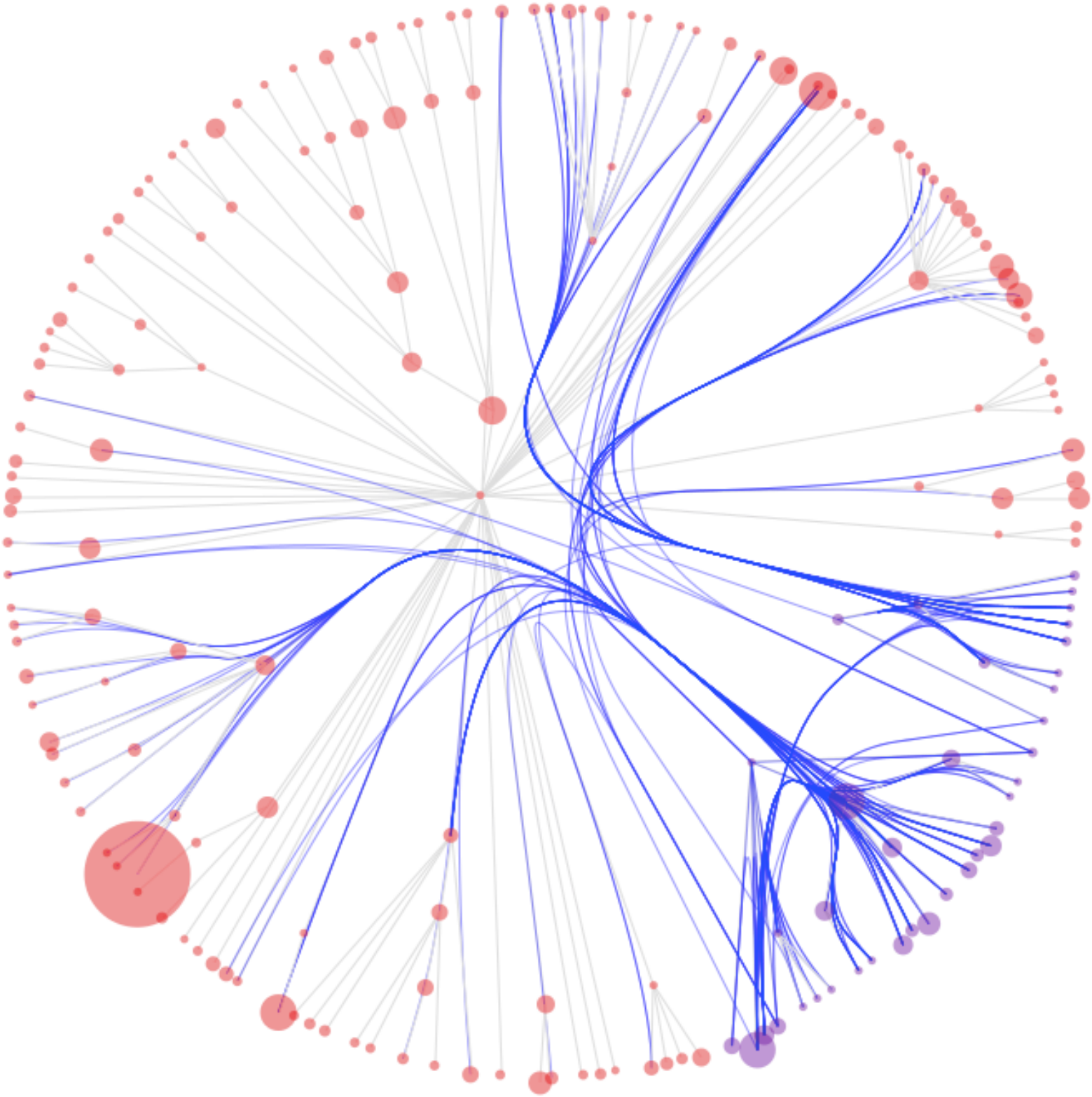

coverage (%)
100
75
50
25
0
1
2215
#tests
P3, P1, P4, P2
P1, P4, P2, P3
P4, P1, P3, P2

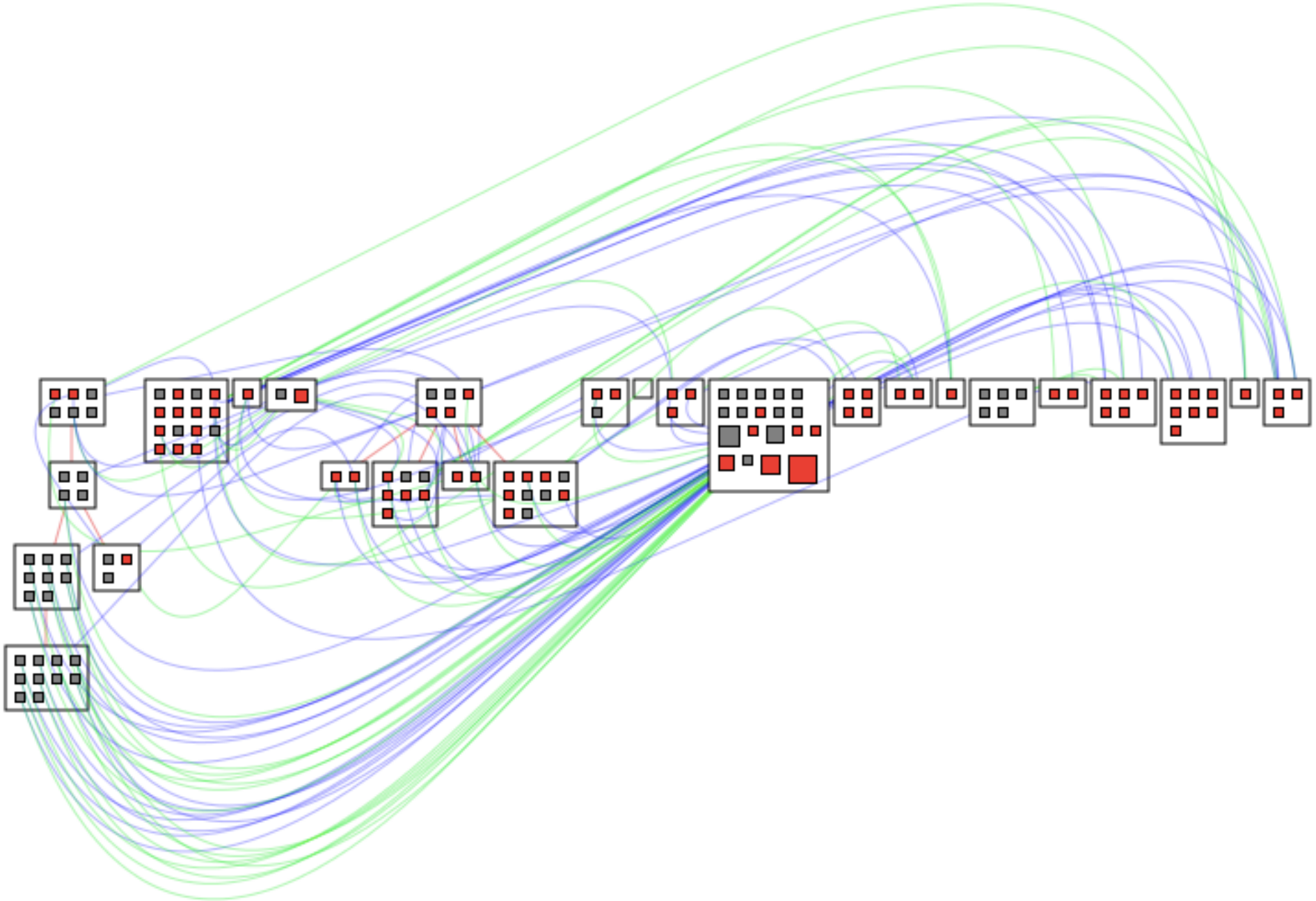

Mondrian
Citezen
Zinc
Roassal
Pier
Seaside
80
60
40
20
0
1
811
801
99
419
950
1484
Optimal
Random1
Random2
Random3

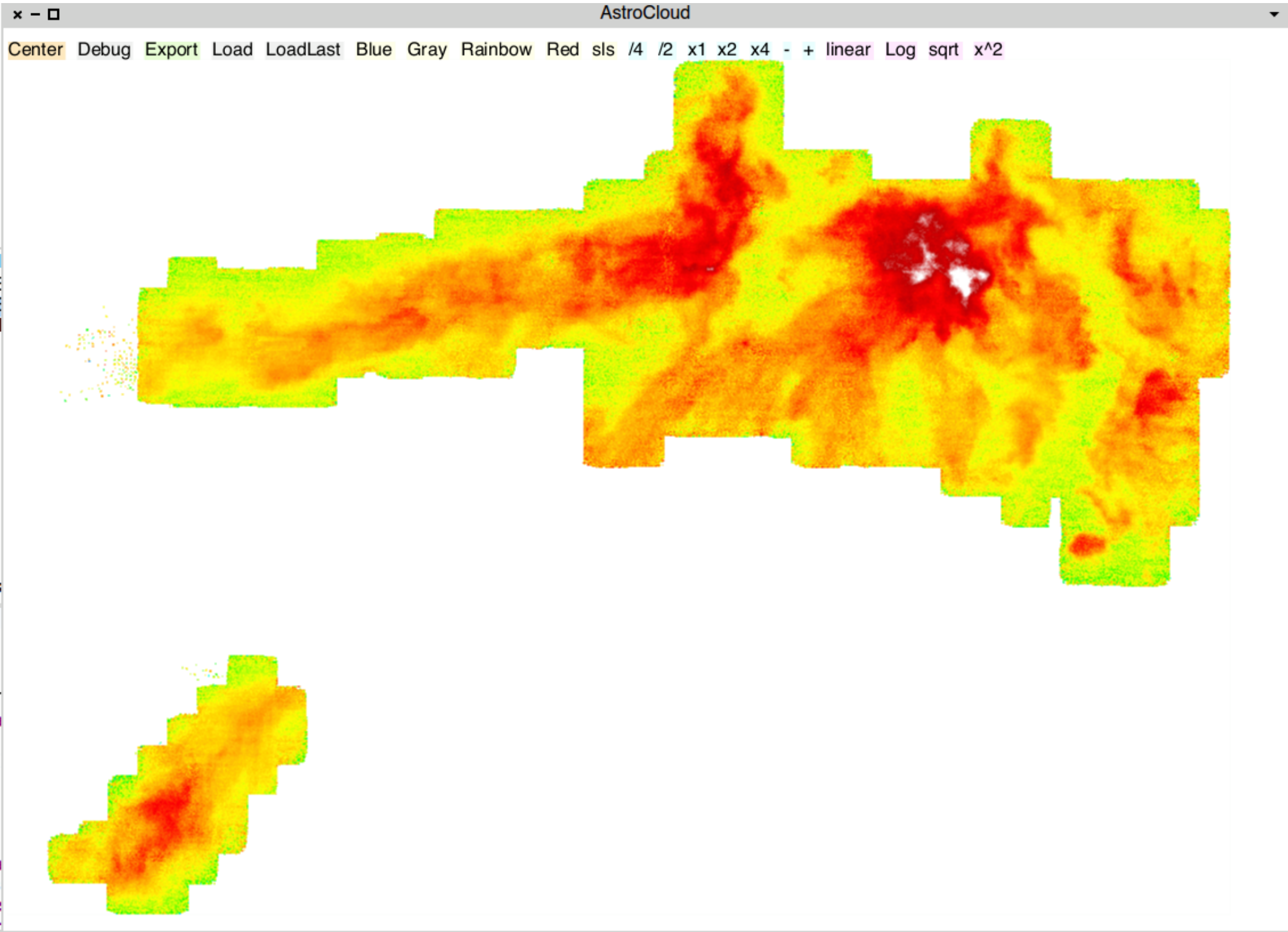
Agile Visualization Engine

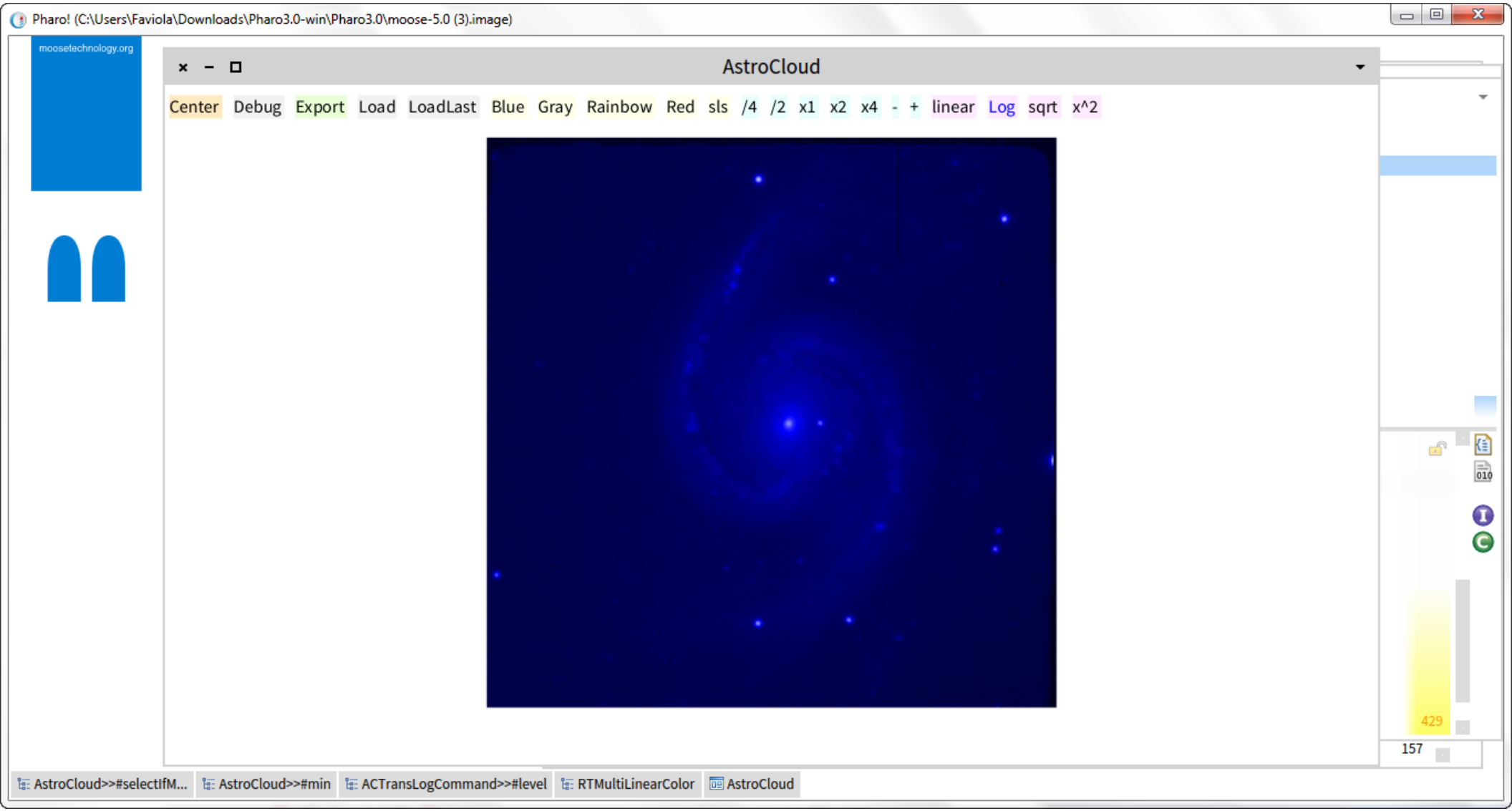

layout
zone->layout
layout->zone
zone
zone->behaviour
behaviour->zone
behaviour
renderer->zone
zone->renderer
source
source->renderer
renderer->source
renderer








AstroCloud
Center Debug Export Load LoadLast Blue Gray Rainbow Red sls /4 /2 x1 x2 x4 - + linear Log sqrt x^2

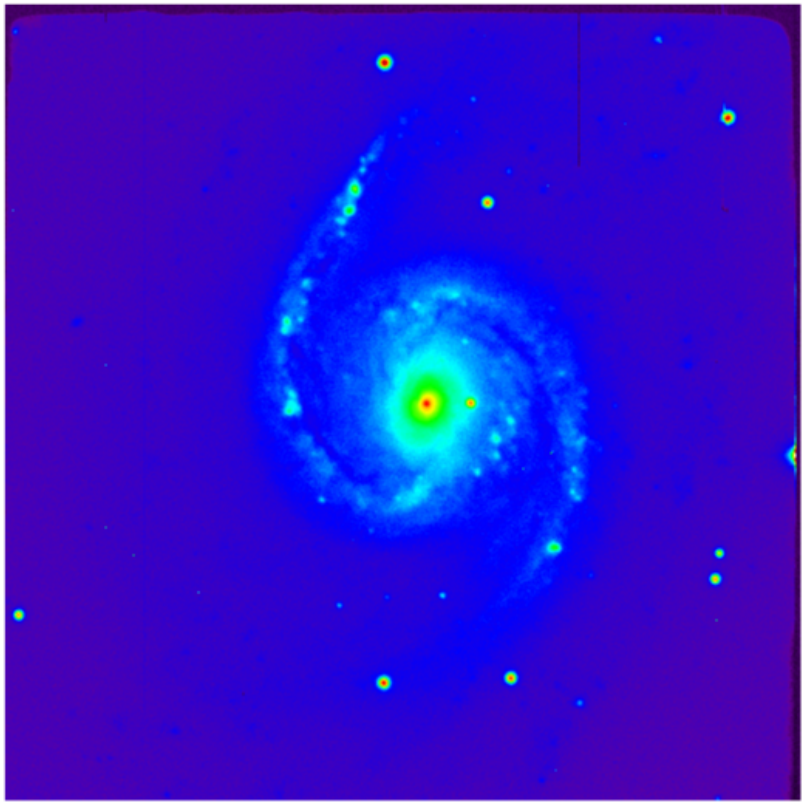

Pharo! (C:\Users\Faviola\Downloads\Pharo3.0-win\Pharo3.0\moose-5.0 (3).image)
moosetechnology.org
AstroCloud
Center Debug Export Load LoadLast Blue Gray Rainbow Red sls /4 /2 x1 x2 x4 - + linear Log sqrt x^2
429
157
AstroCloud>>#selectIfM...
AstroCloud>>#min
ACTransLogCommand>>#level
RTMultiLinearColor
AstroCloud

Pharo! (C:\Users\Faviola\Downloads\Pharo3.0-win\Pharo3.0\moose-5.0 (3).image)

moosetechnology.org

AstroCloud

Center Debug Export Load LoadLast Blue Gray Rainbow Red sls /4 /2 x1 x2 x4 - + linear Log sqrt x^2



429

157

AstroCloud>>#selectIfM... AstroCloud>>#min ACTransLogCommand>>#level RTMultiLinearColor AstroCloud

DEEP INTO Pharo

Pharo is a dynamic object-oriented programming language. Pharo's model and syntax are uniform, simple and expressive. These properties, when added to a powerful and flexible programming environment, regularly attract new developers. The community around Pharo has been steadily increasing over the years. This community is actively creating exciting and innovative software artifacts helping the development of advanced software systems. Pharo leverages the software building experience to its best by offering open and object-oriented programming environments and libraries.

The book covers a large spectrum of topics ranging from central language aspects such as blocks and exceptions to package management and graphics oriented frameworks. Recent frameworks like Roassal and Petit Parser are covered. This book contains unique material often presented in a tutorial form with many experiences to carry on.
Everybody will learn something reading this book : programmers familiar with Pharo will enjoy the highlights made of some particularly beautiful aspects of Pharo as well as discovering new and powerful frameworks. Practitioners making their debut with Pharo will board for a wonderful journey in the realm of objects.

Deep into Pharo not only presents some internal aspects of Pharo but it presents important libraries that proved to be important for a business and development perspective.

Square Bracket Associates

ISBN 978-3-9523341-6-4 90000

9 783952 334164

DEEP INTO Pharo

ALEXANDRE BERGEL - DAMIEN CASSOU - STÉPHANE DUCASSE - JANNIK LAVAL

DEEP INTO Pharo

ALEXANDRE BERGEL - DAMIEN CASSOU
STÉPHANE DUCASSE - JANNIK LAVAL

ESUG 2013 Edition

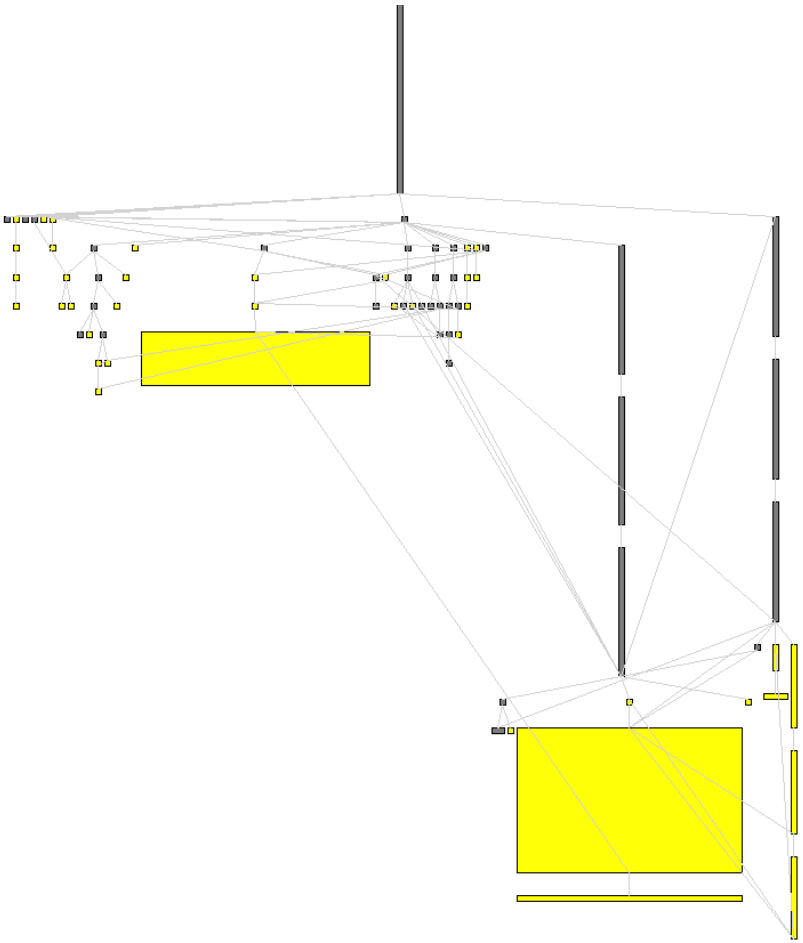
Conclusion

Little innovation in the tools we commonly use

- Profilers, debuggers, testing tools have not significantly evolved

Fantastic opportunities for improvement

Kai, Rizel, Hapao, Roassal are just a beginning



ObjectProfile.com
facebook.com/ObjectProfile
@ObjectProfile

OBJECT PROFILE