Analyzing Dynamic Information with Spy and Roassal: An Experience Report

Alison Fernandez[1], Diego Gabriel Nuñez Duran[1], Alejandro Infante[2], Alexandre Bergel[2]

[1]University of San Simon, Bolivia
[2]Pleiad Lab, DCC, University of Chile

***Abstract*—Dynamic analyses tools are seldom crafted by practitioners. This paper discusses the benefits of supporting the practitioners to build their ad-hoc tool and presents our experience to lower the barrier to gather dynamic information. The experience we present is driven by the combination of the Spy profiling framework and the Roassal visualization engine, two frameworks used in industry and academia. We conclude with two question to discuss at the workshop.**

I. Experiences in Dynamic Analyzes

Analyzing the execution of a software is often a non-trivial activity due to numerous technical aspects to consider. Over the last three years, we have built over a dozen of code execution profilers that cover a wide spectrum of analyses. Our profilers employ Spy, a lightweight profiling framework to gather dynamic information. Collected data is then represented using the Roassal visualization engine.

This paper describes the key points of our frameworks and highlights the experience we have gained. Since Spy and Roassal are written using the Pharo programming language[1] [1], all the examples are given in Pharo. Pharo is a dynamically typed language, with a syntax close to Objective-C and Ruby.

II. Profiling and Visualizing

Spy. Spy is a profiling framework which features the following: (i) profilers made with Spy share the same memory space than the application itself; (ii) instrumentation code may be executed for each method of the base-code; (iii) after the execution, a profile is structured along packages, classes, and methods.

We chose Spy over other non-invasive technologies like aspect oriented programming for the data analysis, mainly because the Spy framework is thought to create profilers. Spy takes care of creating a meta-model that simplifies the querying and organization of the gathered data and also it ensures the consistency of the system. We noticed that simplifying the use of the tools and orienting the user is fundamental for the adoption of the tool in a development process.

As a running example, we will build a profiler to expose dependencies between methods. First we generate a new profiler with the following instruction:

```
Spy generate: 'Dep' category: 'DependencyProfiler'
```

The method `#beforeRun:with:in:` is invoked before invoking a method of the base code. We therefore have to properly define it to incoming calls:

```
DepMethod>>beforeRun: methodName with: args in: receiver
  | caller |
  hasBeenExecuted := true.
  caller := self spySender.
  caller ifNotNil: [
    self incomingCalls add: caller.
    caller outgoingCalls add: self ]
```

The class `DepMethod` offers facilities to store data related to a method, such as `incomingCalls` and `outgoingCalls`. The only missing bit is the initialization of the `hasBeenExecuted` variable:

```
DepMethod>>initialize
  super initialize.
  hasBeenExecuted := false.
```

The `Dep` profiler may now be employed to profile method invocations for a given program execution.

Roassal. Roassal is a visualization engine. Roassal features the following: (i) a visualization lives in the same memory space than the represented domain; (ii) support polymetric views [4] and a charting library to visually represent software-related metrics; (iii) offer a large range of interactions to let the user navigate and browse a visualized domain.

Roassal supports a number of domain-specific languages, including Mondrian [5], a dedicated language for rendering polymetric views. Consider the following Mondrian script:

```
1  Dep>>visualizeOn: aView
2    | executedMethods b |
3
4    executedMethods := self allMethods select:
         #hasBeenExecuted.
5    b := RTMondrian new.
6    b view: aView.
7
8    b nodes: executedMethods.
9
10   b shape line color: (Color blue alpha: 0.2).
11   b edges
12     moveBehind;
13     connectToAll: #outgoingCalls.
14
15   b layout force.
16
17   b normalizer
18     normalizeSize:
19       [ :m | m numberOfOutgoingCalls + m
         numberOfIncomingCalls ];
20     normalizeColor:
21       [ :m | m numberOfOutgoingCalls / (m
         numberOfIncomingCalls + m numberOfOutgoingCalls + 1)
         ]
```

[1]http://pharo.org

```
22          using: { Color blue . Color red };
23        alphaColor: 0.7.
24    b build.
25    ^ b
```

The method `#visualizeOn:` is defined on the class `Dep`, our profiler. The `self` pseudo-variable therefore refers to a profile. Line 4 gets all the methods executed during the execution. Line 5 creates the Mondrian interpreter. Line 8 associates a node to each executed method. Line 10 sets a translucent blue color to lines. For each executed method, Line 11 creates lines from a method and the methods it calls. Line 15 selects a force based layout. Lines 18-19 normalizes the size of each node: a small method is a method that receives / sends a few calls and a big method receives / sends many. Lines 20 - 22 gives a color ranging from blue to red to each method: blue indicates many receiving calls and red many emitting calls.

The profiler and the visualization may be simply invoked using the expression:

```
Dep
  profile: [ "Code to profile here" ]
  onPackagesNamed: "Packages to instrument"
```

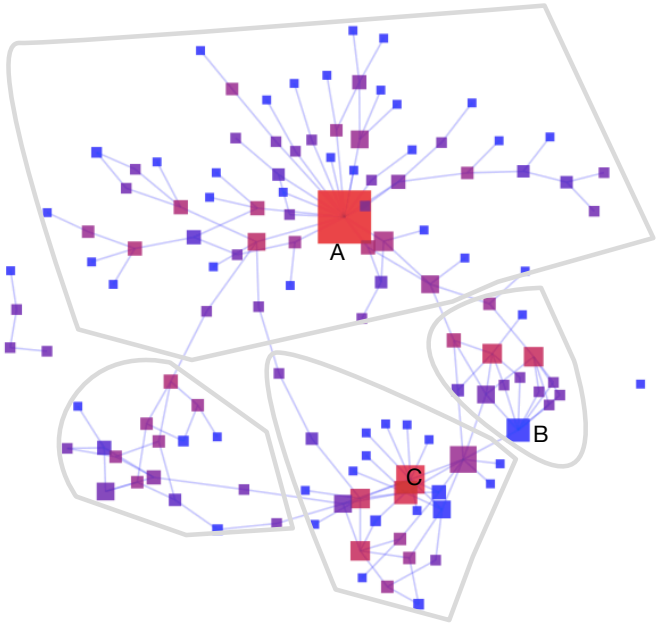

Fig. 1: Execution example: each box is a method; lines are dependencies between methods; blue box is a method that receive messages; red box is a method that send messages.

Figure 1 shows a relatively short program execution that involves only 128 methods. The visual aspect of each method and their connections illustrate the differences between methods in the way they are involved in the computation. Several method invocation chains may be found. The red and large method at the center of the figure, marked `A`, invokes many methods as indicated by the size and color of `A`. This method is the entry point of the execution, *i.e.,* the `main` method. Inspecting its source code (accessible by simply clicking on the element) reveals a particularly long method.

Heavily interconnected methods has been manually marked on the figure. Each of the four groups belongs to a particular component made of several classes. The method `C` is another entry point of a component.

Blue methods are methods that do not call any method (such as variable accessors and call to primitives). Method `B` is a primitive method, invoked by several other methods.

In Figure 2 we can appreciate the execution of the tests of Roassal and Glamour and we can see the most evaluated methods are bigger and that could hint the developer about critic methods that call many methods or are highly called.

Charting. A profiler may offer several visualizations. Grapher is another domain-specific language offered by Roassal. The following script plots the methods along the number of incoming and outgoing calls:

```
1  b := RTGrapher new.
2
3  ds := RTDataSet new.
4  ds interaction popupText.
5  ds dotShape circle color: (Color red alpha: 0.3).
6  ds points: (self allMethods select: #hasBeenExecuted).
7  ds dotSize: [ :m | m numberOfOutgoingCalls + m
       numberOfIncomingCalls ] min: 5 max: 20 using: #yourself.
8  ds x: #numberOfOutgoingCalls.
9  ds y: #numberOfIncomingCalls.
10 b add: ds.
11
12 b axisX title: '#Out'.
13 b axisY title: '#In'.
```

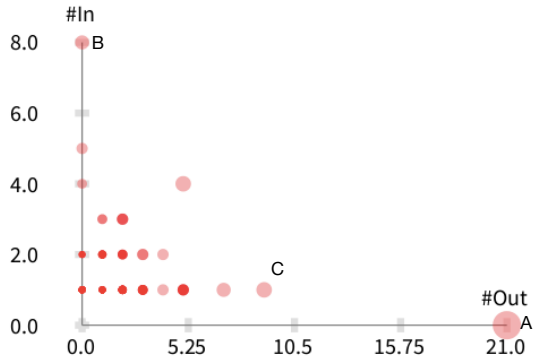

Fig. 3: Method scatterplot

Figure 3 illustrates the result of the script: each dot is a method; size of a dot is the sum of incoming and outgoing calls; horizontally is the number of outgoing calls; vertically is the number of incoming calls. Methods `A`, `B`, and `C` are marked in the scatterplot.

We have used three metrics so far (`numberOfOutgoingCalls`, `numberOfIncomingCalls`, and the sum of these two). A metric is defined as a simple block function, taking as input a Spy element. Advanced metrics about source code are offered by the Moose platform for software and data analysis[2].

[2]http://moosetechnology.org

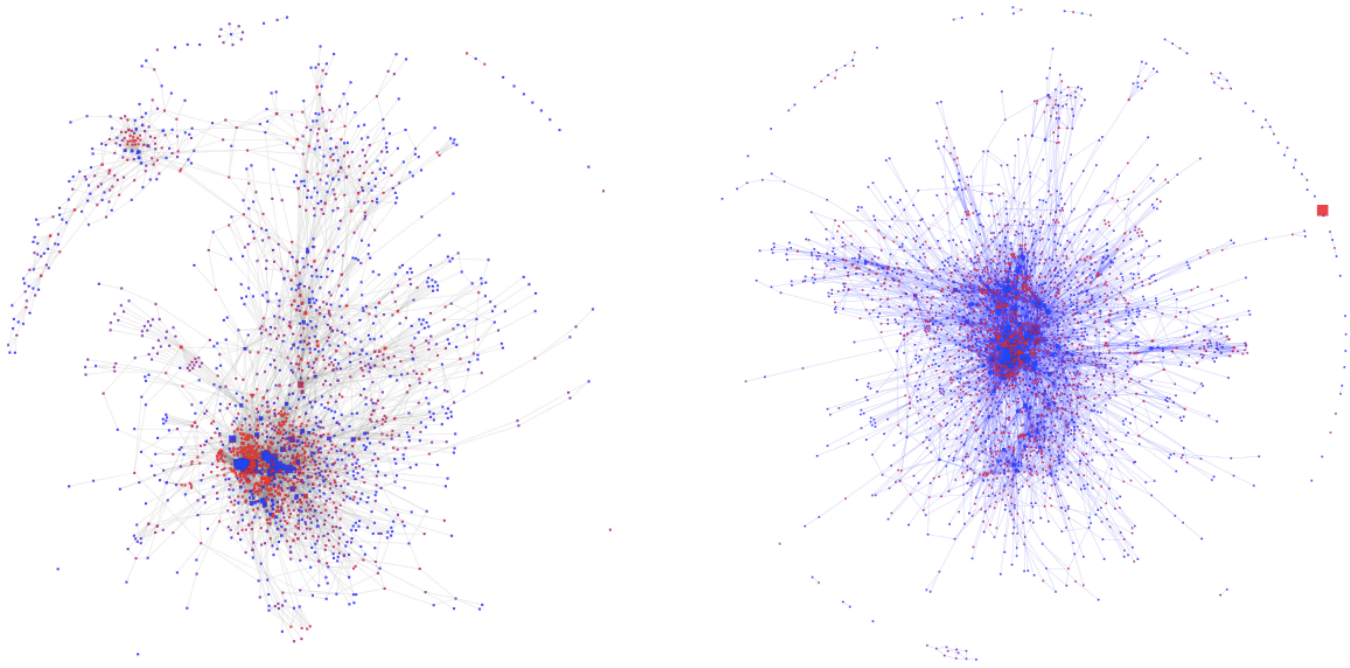

Fig. 2: Visualization of some large executions

III. Gained Experience

Several profilers made with Spy and Roassal are used in industry and in open source communities. This adoption has been key to evolve Spy and Roassal to offer a satisfactory usability experience.

Drilling down. Each visual element represents an element of the profile, itself representing a structural element of the application (*i.e.,* a package, a method or a class). Thanks to the Glamour [1] and GTInspector [3] frameworks, source code behind each visual element is one-click away. Source code is displayed within the same window than the visualization, to minimize cognitive context switching.

Execution overhead. The overhead introduced by the profiler is about 80%, which means that an application run about twice slower when being profiled. This overhead is acceptable is many of the case we have dealt with.

Memory space. The profiler, the visualization and the source code application live in the same memory space. Having the profiler, the visualization and the application sharing the same memory space has not been perceived as being a problem.

Excessive memory consumption may happen. The Pharo system analyzers easily indicate where the memory or the CPU time is being spent on. For example, in case of a large amount of visual nodes may be a burden for some graph layout algorithms. In such a case, the computation may be simply interrupted and the visualization scripting adequately adjusted.

Integration in the IDE. Profilers are essentially made to assist software engineers in a particular task. Making sure that a profiler is easily inserted in a development workflow is essential to ease its adoption. For example, a source code editor may be opened on any represented source code entities. A method represented in a visualization may be modified and the profiler be run again to update the visualization. This approach has been addressed by some IDEs like Eclipse. Visual Studio, and others.

In addition to source code editing, our profiler seamlessly integrates the Pharo debugging framework and the inspector framework. Debugging [6] and inspecting [3] are two essential operations in Pharo. Integrating Spy in the debugger enables domain-specific debugging [2].

Easy to customize. Spy favors the creation of multiple small and focused profilers. Most of the profilers we have built are below less one hundred lines of code. Roassal favors short scripts and rapid development cycles for data visualization. Following the trends of live programming, visualization scripts are iteratively tried and adjusted. In our experience, the same programming fashion applies when analyzing profiling data.

IV. Concluding Words

Both Spy and Roassal are keys in many software analyze activities carried out both in an industrial and academic context. Spy is made to extract information from a program execution while Roassal is designed to map metrics and properties to inter-connected objects.

During the workshop, we hope to discuss on the following questions:

- *How to expose execution analyses data to average programmers?* – Measuring test coverage and software performance are probably the most well-known execution profiling techniques. However, much more than

be extracted from a program under execution. The way we expose dynamic execution to programmers is by using focused visualizations, however we are looking for alternative way to expose dynamic analyses.

- *How to efficiently insert program execution analysis in University curriculum?* – Both Spy and Roassal are used in various lectures on advanced programming. The number of shipped examples, related research papers, and the documentation make both frameworks accessible to undergraduate students with little knowledge about profiling and visualization. However, students and young engineers do not think about crafting their own tool for their punctual need. To conclude, we are convinced that is a skill that would enhance the way of handle a Software Engineering problem.

ACKNOWLEDGMENTS

This work was partially supported by FONDECYT project 1120094 - Chile and by Program U-Apoya, University of Chile.

REFERENCES

[1] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[2] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The Moldable Debugger: A framework for developing domain-specific debuggers. In Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 102–121. Springer International Publishing, 2014.

[3] Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. The Moldable Inspector: a framework for domain-specific object inspection. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.

[4] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[5] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[6] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012.