Reflectivipy: building python debuggers with sub-method, partial behavioral reflection

Steven Costiou, Vincent Aranega, Marcus Denker

Reflectivipy: building python debuggers with sub-method, partial behavioral reflection

Steven Costiou[1], Vincent Aranega[2], and Marcus Denker[1]

[1] Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
{steven.costiou,vincent.aranega, marcus.denker}@inria.fr

Abstract. Building debugging tools is hard and requires powerful tools and libraries. In object-oriented technologies, it is common to use fine-grained reflection to implement debuggers. In this tool presentation, we describe how partial behavioral reflection applied to sub-elements of a method helps in the implementation of advanced debugger features. As an example, we present an implementation of object-centric breakpoints in python.

Keywords: Debugging · Object-centric · Python · Reflection

1 Introduction

Debugging is a general concern in software engineering. Therefore we need to build debuggers, and for that we require support from languages and their infrastructure. Sub-method, partial behavioral reflection [1] is a reflection technique for fine-grained instrumentation of object-oriented programs.

The technique consists in annotating AST nodes at run time with *metalinks*. Metalinks describe calls to a meta-object to be executed for the operation defined by the AST node. They define when to call (before, after, instead) and which information to reify and pass to the meta-object. That meta-object implements and executes instrumentation behavior, such as debugging operation.

Installing metalinks leads to the code being dynamically transformed, recompiled, and installed. This allows the system to be annotated at run time (Figure 1).

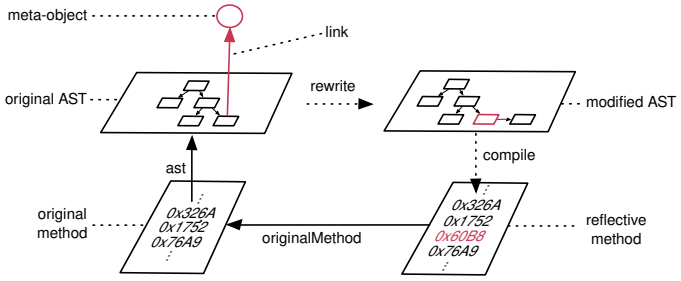


Fig. 1: Run-time method instrumentation with sub-method, partial behavioral reflection

The technique has successfully been implemented and integrated with Pharo as Reflectivity [1]. It has been used in more than 20 research projects to implement fine-grained and customized program instrumentation. Many of these projects are about designing and implementing new debugging tools. In all these projects, sub-method, partial behavioral reflection is the fundamental support for implementing debuggers. In this paper, we present Reflectivipy, our python implementation of the technique. We briefly describe its API, and how it is used to implement an advanced debugger able to target one specific object in a running program. The debugger provides breakpoints scoped to a specific object, with two granularities: on a method or on any sub-expression of a method. The debugger relies on Reflectivipy and IPDB[3].

2 Reflectivipy in a Nutshell

Reflectivipy is our python implementation of sub-method, partial behavioral reflection [1][4]. The flow for the creation and installation of a metalink using `reflectivipy` is the following:

[3] https://ipython.readthedocs.io/en/stable/api/generated/IPython.core.debugger.html
[4] https://github.com/StevenCostiou/reflectivipy

(1) Select an AST node of a method using: reflective_ast_for_method(*classOrObject*, *methodName*).
(2) Create a metalink and configure it using:
MetaLink(*meta-object*, *meta-objectMethodName*, *"before"|"instead"|"after"*).
(3) Install the metalink on the selected AST node using: link(*configuredLink*, *aSelectedNode*).

3 Object-centric Debugging for Python

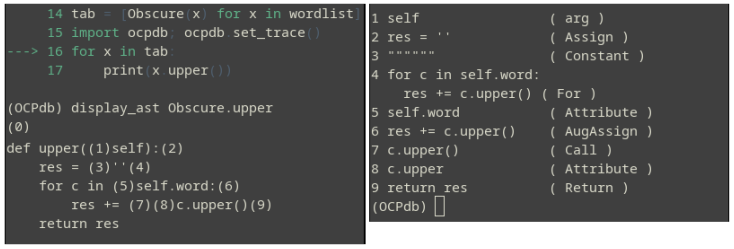
Object-centric debugging [2] scopes breakpoints to specific objects and their interactions instead of breaking the execution for all objects of the same kind. This helps developers to focus their debugging investigations to precise parts of their program. In this demonstration, we illustrate how we use Reflectivipy to implement a breakpoint that interrupts an execution when a precise object receives a message. This breakpoint builds a meta-object that breaks the execution, configures a metalink to call that meta-object, and installs this metalink on an AST of an object's method.

Implementing an object-centric breakpoint with Reflectivipy. To build our object-centric breakpoint, we attach a metalink to the object we want to debug. We install the metalink on a (sub)expression (*i.e.*, an AST node) of a method bound to the object to debug. When the metalink is activated, *e.g.*, when the execution reaches the aforementioned expression, it automatically calls the Python special method `set_trace()` that interrupts the execution.

Installing an object-centric breakpoint. We first have to set a first breakpoint to interrupt the execution and activate the object-centric debug mode. To do that, we insert the breakpoint instruction `ocpdb.set_trace()` in the code we want to debug.

The command `display_ast` command (Figure 2) labels all nodes of a method with a number. Each number correspond to a given AST node. We can install breakpoints on a statement (*e.g*, AST node 6) or on an expression (*e.g*, AST node 7).

Fig. 2: Labelling the AST for an existing `upper` method.



With the `halt` command, we install an object-centric breakpoint. It takes two arguments: (1) the method of a specific object and (2) the number of the node where installing the breakpoint in that method. Let us use it to debug a program that transforms a list of names from lowercase to uppercase. The program output produces ALBERT JODIE MIKE JOHN CARMEN NaLLELY GINA. One of the name is not transformed correctly. To debug it, we install an object-centric breakpoint on the nallely string object to interrupt the execution when upper is called on that object: halt tab[5].upper, 6

We resume the execution which stops exactly on the sixth element of the list (the nallely string object). A program state inspection reveals that the second lowercase letter is not a latin letter, but an UTF-8 letter looking like a latin letter. On a large collection and in a complex program, stopping on the right object is tedious without object-centric breakpoints. In that case, we have to express breakpoint conditions to find the object, which is difficult.

4 Conclusion

We presented Reflectivipy, our Python implementation of sub-method partial behavioral reflection. We used it to implement a breakpoint that scopes to specific objects with a sub-expression granularity. The technique offers strong support for build debugging tools. However, exposing ASTs is tedious and we do it by labeling nodes which is impractical. We need more tools dedicated to ASTs to support the building debuggers based of sub-method, partial behavioral reflection.

References

1. S. Costiou, V. Aranega, and M. Denker. Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use. *The Art, Science, and Engineering of Programming*, 4(3), Feb. 2020.
2. J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012.