HAL open science

Tool demo: fine-grained run-time reflection in Python with Reflectivipy

Vincent Aranega, Steven Costiou, Marcus Denker

To cite this version:

Vincent Aranega, Steven Costiou, Marcus Denker. Tool demo: fine-grained run-time reflection in Python with Reflectivipy. [Research Report] Inria. 2021. hal-03463035

HAL Id: hal-03463035

https://inria.hal.science/hal-03463035v1

Submitted on 2 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Tool demo: fine-grained run-time reflection in Python with Reflectivipy

Vincent Aranega
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France
vincent.aranega@inria.fr

Steven Costiou
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France
steven.costiou@inria.fr

Marcus Denker
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France
marcus.denker@inria.fr

Abstract

Reflectivipy is a Python implementation of sub-method, partial behavioral reflection (SPBR). SPBR provides selective reflection operations applicable to sub-elements of methods (*e.g.*, sub-expressions). SPBR helps in run-time code instrumentation with various application, from advanced debugging to hot patching of running programs. In this tool paper, we briefly describe SPBR and its Reflectivipy API and implementation. We illustrate Reflectivipy through two examples: first we build and demonstrate a basic object-centric debugger and describe how SPBR favors its implementation and second, we hot patch a running REST server.

CCS Concepts: • **Software and its engineering** → *Maintaining software*; Object oriented development; *Development frameworks and environments*; *Object oriented frameworks*; *Software maintenance tools*; *Software notations and tools.*

Keywords: debugging, code instrumentation, reflection, AST transformation, hot-patching

1 Introduction

Sub-method, partial behavioral reflection [3] (or SPBR) is a reflection technique for fine-grained instrumentation of object-oriented programs. The technique consists of annotating AST nodes with meta-behavior to instrument programs. At run time, execution of annotated nodes trigger that meta-behavior, thus executing the instrumentation. In theory, any node can be annotated and any run-time information can be requested. Consequently, it is possible to instrument sub-expressions, (*e.g.*, a message send in an assignment) and to request any contextual meta-data to be used in the instrumentation (*e.g.*, the receiver of the message).

The technique has successfully been integrated to Pharo [1] as Reflectivity [3], inspired by its early implementation as Reflex [12, 14]. Reflectivity has been used in more than 20 research projects to implement fine-grained and customized program instrumentation. Many of these projects are about exploring, designing, and implementing new debugging tools, hot-patching or unanticipated live-program adaptation techniques, and dynamic analysis tools. In all these projects, SPBR is the fundamental support for implementing fine-grained run-time instrumentation.

Today, efforts are being made to implement SPBR in other technologies to bring its benefits to a larger audience. In this paper, we present Reflectivipy, a python implementation of the technique. We illustrate the use of Reflectivipy on two examples: the implementation of a simple object-centric debugger, and the hot-patching of Flask[1], a web framework.

2 Sub-method, partial behavioral reflection

SPBR 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.

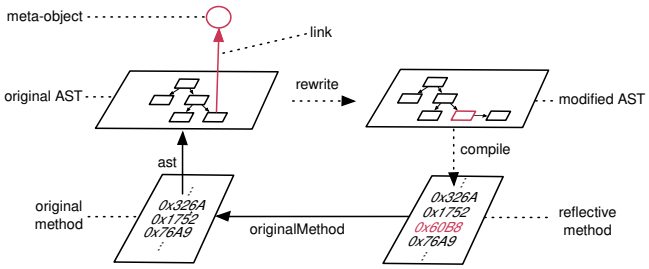

Figure 1. Run-time method instrumentation with SPBR.

Installing metalinks leads to the code being dynamically transformed, recompiled, and installed. This allows the system to be annotated at run time (Figure 1). The main abstractions we are working with are:

- **Metalink.** The object that associates a meta-object with on one or more AST Nodes.
- **Meta-object.** The meta-object is called by the operations that the link is installed on.
- **Reification.** Metalinks can ask for information to be passed to the meta-object. These are, *e.g.*, the arguments of a message send or the name of a variable.
- **AST Node.** A node in the AST models an operation, such as message sends, assignments, etc. MetaLinks are non-textual annotations on the AST node.

[1] https://flask.palletsprojects.com/en/2.0.x/

3 Reflectivipy in a Nutshell

Reflectivipy is a Python implementation of sub-method, partial behavioral reflection [3][2]. In this section, we briefly describe its API and illustrate its usage through basic examples. Then, we describe its implementation and we summarize the key points of this implementation's requirements.

3.1 Reflectivipy API

We illustrate Reflectivity's API by logging the execution of the print(...) instruction from a foo() method:

```
class ExampleClass(object):
  def foo(self):
    print('Executing foo')
```

We first create a class whose instances will play the role of meta-objects that will do the logging:

```
class MetaLogger(object):
  def log(self, sender, node):
    print("I'm here, called by", sender, "from node", node)
```

We then instantiate a metalink:

```
metalink = reflectivipy.MetaLink(MetaLogger(), selector='log
     ', control='before', arguments=('sender', 'node'))
```

The metalink is configured with a meta-object, here an instance of our MetaLogger. The selector represents the message to be sent to the meta-object when the metalink fires. The *ctrl* argument controls when the metalink must fire, relatively to the AST node. It can be before, instead or after the execution of that node. Finally, the *args* argument specifies the reifications to pass as arguments to the meta-behavior. Various reifications are proposed[3] (*e.g.*, sender, node, value) and the reification list is extensible by the end-user.

The metalink has to annotate an AST node. To access the AST of a method, Reflectivipy proposes the reflective_ast_for_method() function. This function takes as arguments an object and a selector, and returns the AST of the method corresponding to the selector. This AST can then be navigated to retrieve a particular node.

```
rf_ast = reflectivipy.reflective_ast_for_method(
     ExampleClass, 'foo')
node = rf_ast.body[0].body[0]
```

The returned method is bound to the object that is passed as parameter. If that object is an instance of a class, the accessed AST will be bound to this instance, and metalinks annotating that AST will only fire for that particular instance.

Finally, the link() function attaches a metalink to an AST node, and unlink() method uninstalls a specific link:

```
link(metalink, node)
a = ExampleClass()
a.foo() #calls log() before the first statement of foo()
metalink.uninstall()
```

[2]https://github.com/StevenCostiou/reflectivipy
[3]https://github.com/StevenCostiou/reflectivipy/blob/master/reflectivipy/reifications.py#L130-L143

3.2 Implementation

Reflectivipy is based on run-time AST transformation[4]. It manipulates ASTs to insert message nodes whose receiver is the metalink's meta-object, and whose selector is the metalink's message selector. We describe how these AST manipulations implement SPBR and we summarize its requirements.

AST transformations. When a metalink is installed, it triggers a transformation of the AST node it annotates:

- It isolates the node to surround it by a meta-level call,
- it introduces intermediate variables nodes to store reifications specified by the metalink, that are passed to the meta-level call as parameters,
- it introduces a message send to the metalink's meta-object (*i.e.*, the meta-level call).

The new AST generated by this transformation is compiled to a new method. By default, transformations are effective for all instances of the class defining the original method. The new method then replaces the original one in the attributes of that class. When the original method is bound to a specific instance, the new method replaces the original one in that instance's attribute dictionary (which scopes the transformation to that instance).

```
1 # Original method
2 def hello():
3   return "Hello world"
4 # Method after AST transformation
5 def hello():
6   temp_Return_4 = __rf_method__.lookup_link(139842966411728)
      .metaobject.return_me()
7   return temp_Return_4
```

Listing 1. New method after AST transformation.

Listing 1 illustrates this transformation. Imagine that we define a metalink configured to replace a node execution by sending the message return_me to a given meta-object. Installing this metalink on the constant node "Hello world" (line 3) of the hello() method (line 1) will transform that method to another hello() method (line 5). The call to the meta-object is inserted line 6, and its result is stored into a new temporary variable. This new temporary variable is returned instead of the original "Hello world" value.

Reflective methods. Each transformed method is bound to a reflective method object (__rf_method__ in Listing 1). Reflective methods preserve the original methods, provide metalinks access and manage their (un)installation. The first time an AST is transformed, a reflective method is created

[4]AST transformations are done with the default Python NodeTransformer class https://docs.python.org/3.10/library/ast.html#ast.NodeTransformer, compilation and installation of transformed methods are done with standard Python tools and libraries.

and associated to the method compiled from that AST. Reflective methods are in charge of:

- Transforming and compiling ASTs,
- restoring original methods whose last link is removed,
- maintaining a connection between an original AST and its latest transformation,
- providing run-time access to installed metalinks.

Metalinks are referenced in reflective methods by a unique id, and accessed at run time through id lookups (line 6). To maintain a connection between ASTs and their transformed version, reflective methods use *twin ASTs*.

Twin ASTs. When an original AST is transformed, that AST is preserved and tied to the transformed AST (*i.e.*, its twin). The source of transformations is always an original AST and the metalinks that annotate it. Each transformation always produces a new twin that replaces the previous one. This separates the AST that developers annotate (the original) and the transformed one (the twin) that is executed at run time. This allows reflective methods to keep track of installed metalinks, and to dynamically (un)install metalinks.

Implementation requirements. Table 1 presents SPBR implementation requirements. Introspection is necessary to access ASTs at installation time, and to reify information at run time. Run-time recompilation is necessary to dynamically replace methods, since metalinks are installed after the program started. To define and install *on-the-fly* (or unanticipated) metalinks, the language must support *live interaction*, such as live programming [10] or online debugging [7]. To dynamically define new meta-objects, the language must support the addition of classes at run time (or Dynamic Software Update (DSU) [9, 15]). Per-instance metalinks require an intercession mechanism able to modify methods for single instances. In Python, methods can be modified in the local dictionary of the instances themselves. Otherwise, it requires instance migration from the instance's base class to a subclass where the transformed methods are installed [**?**].

Table 1. SPBR implementation requirements.

Requirement	Target language support
AST annotation	Access to methods' ASTs
Reifications	Access to the execution stack
Metalink installation	Run-time recompilation
On-the-fly metalinks	Live interaction or DSU
Per-instance metalinks	Instance migration, intercession

4 Reflectivipy in practice

We illustrate Reflectivipy by implementing an object-centric breakpoint and hot-patching of a running REST web server.

4.1 Object-centric Debugging for Python

Object-centric debugging [11] 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 on 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. We implemented the object-centric do_halt breakpoint command. It takes two arguments: a string referring to an instance's method (*e.g.*, my_instance.my_method), and the identifier of the AST node on which we want to install the breakpoint. Listing 2 shows the implementation of the do_halt command.

We first acquire the AST of the method pointed by the arguments of the command (summarized line 3). Because the first argument points to a method bound to a specific instance, the acquired node is also bound to that instance. Therefore, the AST transformation will be local to that instance, and the breakpoint will be scoped to that instance.

Then, we instantiate a metalink (line 4). This metalink will send the halt message to an OCBreakpoint meta-object before the target node. The metalink will also reify itself (arguments=['link']) and pass that reification as an argument of the halt message. We install the metalink (line 5).

```
1 class OCPdb(Pdb):
2   def do_halt(self, args):
3     node = #parses arguments to lookup the selected node
4     metalink = MetaLink(OCBreakpoint(), 'halt', arguments=['
       link'], control='before')
5     link(metalink, node)
```

Listing 2. The do_halt command to set object-centric breakpoints.

At run time, when the target instance will execute the method transformed by the metalink, the metalink will call the halt method of its associated OCBreakpoint meta-object.

This meta-object is implemented as in Listing 3. In the halt method, we create an instance of the OCPdb debugger (line 3). This OCPdb is an extension of the basic Pdb Python debugger we built to handle object-centric breakpoints.

We first use the control parameter of the metalink (*before, after or instead)* to compute a line adjustment for debugger display (line 4). When the execution breaks, OCPdb uses this line adjustement to find which line to display from the original method instead of the Reflectivipy transformed version.

We then use OCPdb to set a breakpoint in the previous executed frame (line 5), *i.e.*, in the executing instrumented method. At run time, it configures the calling frame so that

the execution immediately breaks when the halt method returns to that frame.

```
class OCBreakpoint(object):
  def halt(self, link):
    ocpdb = OCPdb()
    ocpdb.line_adjust = self.adjustment[link.control]
    ocpdb.set_trace(sys._getframe().f_back)
```

Listing 3. Object-centric breakpoint implementation.

Object-centric debugging workflow. When a developer wants to set an object-centric breakpoint, she needs to select an AST node and an object. To enable this, we extended the Python debugger REPL with two new commands: `do_halt` (described above) and `display_ast`. This command displays the AST of a method with labels, to which developers refer to for selecting a node. The object-centric workflow is:

1. Use the native Python `set_trace()` method to set a first breakpoint to halt the program in a place where to find an object of interest.
2. Select an AST node from a method bound to that object, using the `display_ast` command.
3. Install an object-centric breakpoint with the `do_halt` command, with as parameters the method reference and the selected node identifier.

An object-centric debugging example. To demonstrate our debugger, we debug a program that transforms a shuffled list of names from lowercase to uppercase. Listing 4 shows the main part of the program. A tuple of string names is defined (more than 1000 names) then each name is wrapped into an Obscure object providing an upper() method (lines $1-2$). The collection is shuffled (line 3) and each name is displayed in uppercase in the loop under (lines $4-5$).

```
1 wordlist = (..., "jodie", "john", "gina", "nallely")
2 tab = [Obscure(x) for x in wordlist]
3 random.shuffle(tab)
4 for x in tab:
5     print(x.upper())
```

Listing 4. Program under debug.

The program execution outputs a list of a thousand names (Figure 2) where one name is not set as uppercase: the name `nallely` is not transformed properly.

Standard breakpoints in upper() would break for each object of the collection. Conditional breakpoints could be used, but our objects are simple. It might be harder to express stopping conditions in more complex programs. Let us use an object-centric breakpoint!

We first set a breakpoint before the for loop. In the debugger, we use display_ast (Figure 3) to label all nodes and select the node we want to stop on. We want to set an object-centric breakpoint on the AugAssign node in the upper method (node 6 in Figure 3) for the nallely string object. Since the nallely string is the last object in our collection, we do:

```
halt tab[-1].upper, 6
```

This will interrupt the execution when upper is called on the nallely string, just before node 6 is executed.

We resume the execution which stops exactly when the nallely string object is accessed in the upper method of the Obscure class (Figure 4). 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 (Figure 5)!

4.2 Hot-patching a REST server using Reflectivipy

In this example, we hot-patch a running Flask REST server by dynamically changing the behavior of a sub-expression of the server code. This code (Listing 5) defines an end-point (line 3) on a root URL answering a simple *"Hello World"*.

```
1 app = Flask('simpleHelloWorld')
2 class Container(object):
3     @app.route("/")
4     def hello():
5         return "Hello World"
```

Listing 5. A simple REST endpoint in Flask.

We want to change the value of the string returned by the default server route /. However, Flask installs all routes as specific end-points that cannot be redefined at run time. Consequently, to redefine the end-point, we have to access a python shell to get the server's process, patch Flask to allow run-time end-point redefinition, then patch the end-point.

A meta-behavior to ignore sub-expressions. After gaining access to the running server process[5], we patch the add_url_rule method responsible to install end-points. The Flask code preventing end-point redefinition is the last if check in Listing 6. It raises an exception if the new end-point is different from the previous one (line 6).

```
1 @setupmethod
2 def add_url_rule(self, ...) -> None:
3   ...
4   if view_func is not None:
5         old_func = self.view_functions.get(endpoint)
6         if old_func is not None and old_func = view_func:
7             raise AssertionError(...)
8         self.view_functions[endpoint] = view_func
```

Listing 6. Flask mechanism to prevent endpoint redefinition.

We use Reflectivipy to ignore this last condition, allowing us to change an existing endpoint. Listing 7 shows how a "pass" meta-behavior is defined and how it is installed to ignore the end-point redefinition check. At line $1-3$, we define the "pass" meta-behavior, which does nothing. It will be installed on each AST node that we want to ignore at run time. At line $4-5$, we access to the inner if node defined

[5]We use *pyrasite-shell* https://pyrasite.readthedocs.io/en/latest/

line 6 – 7 in Listing 6. At line 6, we configure a metalink designed to execute our "pass" meta-behavior instead of executing the node it is installed on. At line 7 we install our metalink on the if check node we want to ignore.

```
1 class Mb(object):
2   def pass(self):
3     ...
4 ast = reflective_ast_for_method(Flask, 'add_url_rule')
5 node = ast.body[0].body[-1].body[1]
6 passLk = MetaLink(Mb(), selector='pass', control='instead')
7 link(passLk, node)
```

Listing 7. Patch: ignoring the Flask end-point redefinition.

Fine-grained hot-patching. Now that we patched the framework, we can modify end-points. In Listing 8, we define a new meta-behavior that returns the "Goodbye World!" string (lines 1 – 3). We select the original endpoint AST node (lines 5 – 6) then we configure and install a metalink on that node (line 7 – 8). Line 9, we uninstall the ignore link used to patch the framework to restore the original Flask behavior.

```
1 class Mb(object):
2   def bye(self):
3     return 'Goodbye World'
4
5 ast = reflective_ast_for_method(RouteContainer, 'hello')
6 node = ast.body[0].body[0].value
7 metalink = MetaLink(Mb(), selector='bye', control='instead')
8 link(metalink, node)
9 passLk.uninstall()
```

Listing 8. Altering a sub-expression of the original endpoint.

Our end-point now replaces the original one: each access to the default route will show "Goodbye World!" instead of the original text.

5 Discussion

We discuss in this section some limitations of Reflectivipy.

AST node selection. Selecting AST nodes of a method is tedious because it has to be done by manually navigating ASTs. Slight changes in the code can lead to obsolete navigation paths. An interesting solution to explore would be a pointcut language such as AOP pointcuts [5].

On-stack method modification. Reflectivipy cannot modify methods whose execution already started. It requires in-depth execution stack and frame manipulation to inject a new "code object" into the current frame. In Python, this is impossible from within the language itself. It also implies a manipulation of the program counter, which is forbidden in a normal execution. Even if Python proposes a strong intercession mechanism, to alter frame execution at run time we would need to modify the virtual machine.

Asynchronous code instrumentation. Python owns a special syntax to define call/wait asynchronous code. Reflectivipy does not yet support the instrumentation of asynchronous calls. It requires work to ensure that AST transformations do not disturb the asynchronous code execution.

6 Related work

Object-oriented languages usually provide some reflective features. These models of reflection are not operating on an operational level, nor do they allow cross-cutting. This limits their use for the kind of use-cases we show in this paper.

Reflex [14] pioneered Partial Behavioral Reflection. Reflex's metalinks are put on collections of operations over the java bytecode. Similarly, Reflectivity combines an AST-based reflective model of method structure with the idea of Metalinks. Reflectivipy realizes this model in Python.

AOP [5] provides language-level abstractions to model cross-cutting concerns. Pointcuts denote points in the execution of a program that trigger code defined in Aspects. The Metalink model can be seen as a reflective model that is powerful enough to implement an AOP language. A pointcut language could be compiled to metalinks, while the aspect would be the cross-cutting meta-object [4, 13].

Other tools can be used to build features that could be assimilated as run-time reflection. Bytecode instrumentation [6, 8] rewrite bytecode to provide more or less fine-grained instrumentation through which we can achieve introspection and intercession. DynamoRIO [2] instruments binaries, and acts as a run-time control layer between the program and the underlying operating system. It proposes a low-level instruction generator for dynamic run-time manipulation of the binary. Because of that, it can instrument external compiled libraries, while Reflectivipy requires to access a user-readable AST to annotate it and to rewrite it. Such AST manipulation is user-friendly for developers as it is closer to the source code. But it is tedious to implement and we need more tools to expose ASTs and work with them.

7 Conclusion

We presented Reflectivipy, a 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, and to hot-patch a web framework to alter its behavior at run time. As shown with these two examples, Reflectivipy offers strong support for fine-grained instrumentation at run-time. As future work, we plan to build better AST support, to enable on-stack method modifications and instrumentation of asynchronous code.

References

[1] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland. 333 pages. http://books.pharo.org

```
...
ALBERT
CARMEN
JOHN
NaLLELY
MIKE
JODIE
GINA
List size: 1000
```

Figure 2. Erroneous behavior observation

```
(OCPdb) display_ast Obscure.upper
(0)
def upper((1)self):(2)
    res = (3)''(4)
    for c in (5)self.word:(6)
        res += (7)(8)c.upper()(9)
    return res

1 self                ( arg )
2 res = ''            ( Assign )
3 """"""              ( Constant )
4 for c in self.word:
    res += c.upper() ( For )
5 self.word           ( Attribute )
6 res += c.upper()    ( AugAssign )
7 c.upper()           ( Call )
8 c.upper             ( Attribute )
9 return res          ( Return )
(OCPdb)
```

Figure 3. Labelling the AST for an existing `upper` method.

```
(OCPdb) continue
ALBERT
JODIE
MIKE
JOHN
CARMEN
         for c in self.word:
>            res += c.upper()
         return res
```

Figure 4. Execution broke only for one instance.

```
(OCPdb) continue
         res = ''
>        for c in self.word:
             res += c.upper()

(OCPdb) !c
'a'
(OCPdb) ord(c)
120250
(OCPdb) ord('a')
97
```

Figure 5. Inspecting the state of program.

[2] Bernd Bruegge and Allen H. Dutoit. 2004. *Object-Oriented Software Engineering Using UML, Patters, and Java Second Edition.* Prentice-Hall.

[3] Steven Costiou, Vincent Aranega, and Marcus Denker. 2020. Submethod, partial behavioral reflection with Reflectivity: Looking back on 10 years of use. *The Art, Science, and Engineering of Programming* 4, 3 (Feb. 2020). https://doi.org/10.22152/programming-journal.org/2020/4/5

[4] Johan Fabry and Daniel Galdames. 2012. PHANtom: a modern aspect language for Pharo Smalltalk. *Software: Practice and Experience* (2012), n/a–n/a. https://doi.org/10.1002/spe.2117

[5] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. 1997. *Aspect-oriented programming.* Technical Report. Xerox Palo Alto Reserach Center.

[6] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12).* ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2162049.2162077

[7] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2018. Out-Of-Place debugging: a debugging architecture to reduce debugging interference. *The Art, Science, and Engineering of Programming* 3, 2 (Nov. 2018). https://doi.org/10.22152/programming-journal.org/2019/3/3

[8] Philippe Moret, Walter Binder, and Éric Tanter. 2011. Polymorphic bytecode instrumentation. In *Proceedings of the tenth international conference on Aspect-oriented software development (AOSD '11).* ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/1960275.1960292

[9] Luís Pina, Luís Veiga, and Mickael Hicks. 2014. Rubah: DSU for Java on a stock JVM. In *Proceedings of OOPSLA.*

[10] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and live, programming and coding: a literature study comparing perspectives on liveness. *The Art, Science, and Engineering of Programming* 3, 1 (2018). https://doi.org/10.22152/programming-journal.org/2019/3/1

[11] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-Centric Debugging. In *Proceeding of the 34rd international conference on Software engineering (ICSE '12).* https://doi.org/10.1109/ICSE.2012.6227167

[12] Éric Tanter, Noury Bouraqadi, and Jacques Noyé. 2001. Reflex — Towards an open reflective extension of Java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (LNCS)*, Vol. 2192. Springer-Verlag, 25–43.

[13] Éric Tanter and Jacques Noyé. 2005. A Versatile Kernel for Multi-Language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005) (LNCS)*, Vol. 3676. Tallin, Estonia.

[14] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. 2003. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices.* 27–46. https://doi.org/10.1145/949305.949309

[15] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. 2016. Instance Migration in Dynamic Software Update. In *Meta'16.* Amsterdam, Netherlands. https://hal.inria.fr/hal-01611600