Squeak Makes a Good Python Debugger

Bringing Other Programming Languages into Smalltalk's Tools

Fabio Niephaus
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
fniephaus@acm.org

Tim Felgentreff
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tim.felgentreff@hpi.de

Tobias Pape
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tobias.pape@hpi.de

Robert Hirschfeld
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.de

ABSTRACT

Interactive debuggers are indispensable in many software development scenarios. However, they are often hard to extend and more importantly, their capabilities are limited to an application programming interface (API) provided by the runtime executing the corresponding programming language.

We propose an approach that allows to use the live tools of a Smalltalk environment for other programming languages. The approach is based on interpreter-level composition, ultimately making a full-fledged integrated development environment (IDE) part of the language execution process. This allows to directly control interpreters of foreign languages from Smalltalk. It also enables tool reuse and provides the ability to rapidly build new tools.

As an example, we demonstrate how we have combined Squeak/Smalltalk and PyPy's Python implementation. We then reused Squeak's debugger, so that it enables edit-and-continue style debugging of Python applications — something that is currently not supported by Python's PDB or any Python IDE.

CCS CONCEPTS

• **Software and its engineering** → *Integrated and visual development environments*; *Software testing and debugging*; Virtual machines;

KEYWORDS

Smalltalk, Python, debuggers, integrated environments, virtual machines

ACM Reference format:
Fabio Niephaus, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. 2017. Squeak Makes a Good Python Debugger. In *Proceedings of Programming Experience Workshop, Brussels, Belgium, April 2017 (PX'17),* 8 pages.
DOI: 10.475/123_4

PX'17, Brussels, Belgium

123-4567-24-567/08/06…$15.00
DOI: 10.475/123_4

1 BACKGROUND AND MOTIVATION

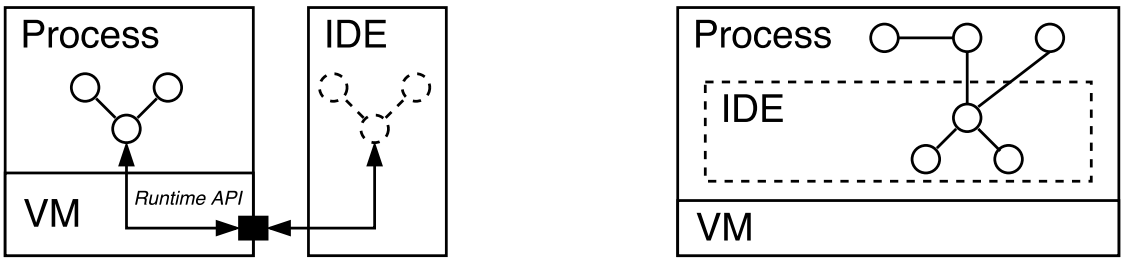
Programming environments play a central role in software development. Developers have come to expect language-specific amenities from their development environments, ranging from syntax highlighting, refactoring support, context-specific auto-completion, to interactive debugging and live code reloading. Most environments offer at least some of these features. Furthermore, there is a renewed push in current research to allow developers to use live, run-time data that can be explored and manipulated to understand and extend software systems [6, 27], through edit-and-continue style debugging and coding against live data. However, efforts towards that goal are often tied to specific domains or programming languages (or both).

IDE systems such as Eclipse or NetBeans try to provide a framework to build development tools for a variety of programming languages. Their architecture is set up to minimize the dependency on any particular language feature, to support a range of different languages. This has the advantage that code can be reused between tools for different languages, at the cost of not integrating deeply with any language.

Most combinations of IDEs and runtimes share a common debugging architecture (see Figure 1a). When the program under development is running, the tools connect to it through some *runtime API* to offer inspection and debugging depending on the underlying capabilities of the runtime. For example, the Java Virtual Machine (JVM) Tools Interface (JVMTI) offers read-only[1] inspection capabilities for a running Java program. It can stop and inspect, force early returns with a particular return value, or set local variable values in the top frame. The HotSpot virtual machine (VM) also allows restarting a frame that is already running and hot code swapping for frames that are not active on the stack. Microsoft's Common Language Runtime (CLR) has a dedicated Interface (`ICorDebug`) that allows edit-and-continue active stack frames;[2] and `ICorDebugEval` allows the IDE to inject and execute arbitrary code.[3]

An IDE constructed in this way can only reflect on the execution state through the runtime API. To inspect and modify objects, representations for them are reproduced in the context of the IDE. Objects and properties that cannot be transferred through the runtime API, cannot be inspected or modified.

[1] https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html
[2] https://msdn.microsoft.com/en-us/library/ms231220.aspx
[3] https://blogs.msdn.microsoft.com/jmstall/2006/01/04/partition-of-icordebug/



(a) Eclipse is separate from the language process.

(b) Squeak/Smalltalk is part of the language process.

Figure 1: Architectural comparison of Eclipse and Squeak/Smalltalk.

On the other hand, live programming systems, such as Squeak/Smalltalk, are well-suited to provide full access to the runtime state and provide developers with means to adapt their tools at development time according to their needs. In such systems, the IDE is entirely contained in the running process, and thus has full reflective access to the state of the system and can directly inspect and manipulate all objects therein, as well as provide edit-and-continue style debugging (see Figure 1b).

However, due to the deep integration with the runtime, tool implementations in live, always-on programming environments like Squeak are very language-specific, and even the means to write tools do not carry over very well. Thus, even those tools that have been reproduced for other programming systems have been written from ground up.

We propose an architecture for constructing a multi-language runtime that attempts to combine the benefits of both approaches: a common code-base for tool developers and live, immediate access to the running application for inspecting and manipulating state.

Our contributions are as follows:

- An architecture to compose multiple languages within the same live programming environment with reflective capabilities for full execution control from within the runtime.
- An implementation of said architecture using PyPy [21] and RSqueak/VM [5, 12].
- An implementation of a debugger that works for both, Python and Squeak/Smalltalk.

In Section 2 we introduce our approach. Then, we demonstrate how this approach can be implemented with an example in Section 3. Afterwards, we discuss advantages and disadvantages of our approach in Section 4. Related work is then mentioned in Section 5. Finally in Section 6, we conclude the paper and describe future work.

2 APPROACH

Various interpreted programming languages provide only very limited debugging support which can be a burden for developers when trying to understand a misbehavior that occurs in their application. On the other hand, Smalltalk is not only a programming language, but also an IDE. This means, that the IDE is an actual part of the process running Smalltalk. This gives developers full control over the running applications and therefore allows for very comprehensive debugging tools.

We propose the idea to use a Smalltalk environment as an IDE for other languages and perform the integration on interpreter-level. This way, our architecture avoids the n-to-m problems of interfacing multiple languages by mapping all languages into the language of the live environment. This can be achieved by composing a Smalltalk interpreter with another language's interpreter. Interpreter composition is especially convenient when both languages are implemented in the same interpreter framework, such as RPython [1], as demonstrated in Section 3.

However, this kind of composition states the problem of how to run two or more interpreters at the same time. Smalltalk has a concept of processes which can be scheduled dynamically [14]. As we demonstrate later, it is possible to make the execution of a non-Smalltalk interpreter part of a Smalltalk-level process. This way, the Smalltalk scheduler decides when to continue with the other interpreter loop, which ensures that the developer can interact with the Smalltalk environment at the same time.

In such a setup, it is now possible to control interpreters of a foreign language with tools written in Smalltalk. For this to work, the VM needs to be extended in such a way that it exposes a number of interpreter-controlling primitives which can then be called accordingly from within a Smalltalk environment. In order to be able to execute a program written in another language, primitives for example need to exist which provide an entry point as well as an end point for the corresponding interpreter loop. Similarly, primitives for restarting and stepping in call frames need to be implemented to enable debugging support.

Squeak already contains various tools originally demonstrated in Smalltalk-80, including an interactive debugger [13]. Since Squeak's tools are designed to work as part of a framework, it is straightforward to adopt them, so they can be used for other languages as well. In addition, new tools, such as application-specific debugging tools, can quickly be built in Smalltalk [25].

3 IMPLEMENTATION

We have applied our approach in Section 2, so that we can use Squeak/Smalltalk as an IDE for Python. The resulting architecture of our implementation is depicted in Figure 2. On VM-level, there is an interpreter loop for each language as well as a VM plugin with a set of primitives which allow Smalltalk to interact with the Python bytecode loop. In the Smalltalk image, we have introduced different classes in order to bridge between Python and Smalltalk. In the following, we explain details of this implementation.

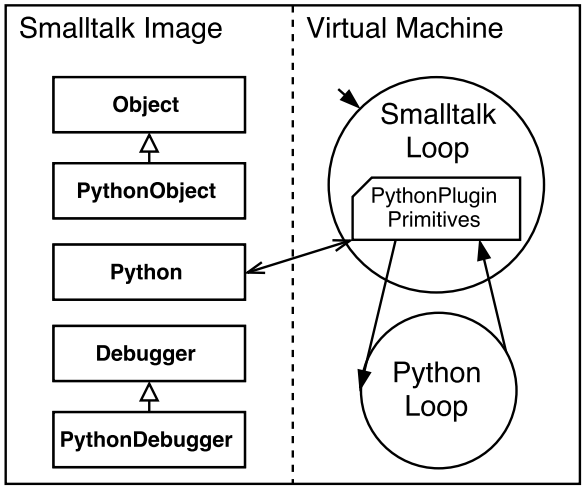


Figure 2: Architecture of the Smalltalk and Python composition.

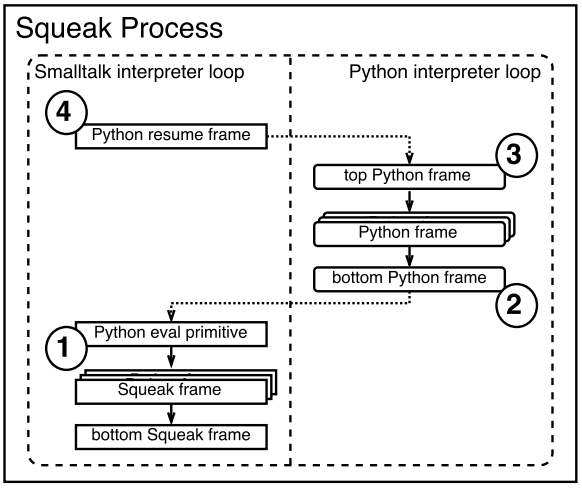


Figure 3: Mixed-stack Smalltalk and Python execution.

3.1 VM-level Implementation

First, we need to build a VM that is capable of executing both, Smalltalk and Python code. This can be achieved by composing a Smalltalk interpreter with a Python interpreter. RSqueak/VM [5] is a RPython-based implementation of Squeak [16], a Smalltalk implementation derived from Smalltalk-80 and a live programming environment. PyPy [21] is a Python implementation and the first RPython VM.

By simply combining the two interpreters, we can create a virtual machine with support for both programming languages. However, to use Squeak/Smalltalk as a live development environment for Python, we need to be able to run both interpreters concurrently.

Smalltalk implements co-operative multitasking through *processes* [14]. We leverage this and integrate the execution of Python bytecodes with a Smalltalk-level process, leaving the decision when to run the next bytecodes up to the Smalltalk scheduler. This allows us to interact with the Squeak/Smalltalk environment as usual while a Python program is running, as well as to interrupt this process to inspect it from Smalltalk.

We create a mixed stack of Python and Smalltalk frames that is managed like any other Squeak process (cf. Figure 3). The core responsibility of the VM in our approach is to maintain the sender-relationship ②–① and ④–③ across interpreter loops.

We switch from Squeak to Python by executing a primitive ① to enter Python code. This creates a Python frame ② that executes in the Python interpreter loop. When this frame returns, the VM will return from the primitive ① and transfer control back to its sender.

While the Python interpreter is running, we maintain a counter of how many Python bytecodes have been executed to decide when to transfer control back to Squeak in order to give other processes a chance to run (e.g., so that the UI process can handle user input). Since Python frames are not visible to Squeak and thus the Squeak scheduler cannot switch directly back to Python frames at a later time, we create an entry point for the scheduler by putting a Squeak frame on the top of the stack. When the Python interpreter loop switches back to Squeak, the top Python frame ③ creates an artificial suspended Squeak "resume" frame ④. When the Squeak scheduler resumes this artificial frame later, it immediately returns control back to the top Python frame ③.

Since interpreters maintain some execution state directly on the stack, we use *Stacklets*,[4] which are provided by the RPython standard library. Stacklets allow us to implement minimal coroutines for regions of the C stack to switch between the two interpreter loops. Since this is a feature of the RPython framework and not of any specific language implementation, the same principle can also be applied to any other interpreter written in RPython.

Similar to other Smalltalk VMs, RSqueak/VM can be extended with plugins. Not only have we added VM primitives that can be called from within the Smalltalk environment to control the PyPy interpreter loop. Since RPython is a subset of Python which compiles to C, we were able to build the entire composition in a single `PythonPlugin`. The code of the plugin for example also patches the PyPy bytecode loop as well as the class which represents Python frames internally before the actual translation begins. After translating this new virtual machine, we can open a Squeak/Smalltalk image and make it aware that the VM also supports the Python programming language.

3.2 Bridging between Squeak and PyPy

We start by providing a class `PythonObject`. This class is special, because the VM will automatically expose objects of the Python object space as instances of this new class. In addition, all primitives of the `PythonPlugin` are able to automatically convert primitive data types between Python and Smalltalk. Python strings are therefore for example converted to Smalltalk `ByteStrings` and vice-versa. Now that the VM can inject any kind of Python object into an image, we need be able to interact with these objects. For this reason, we have added appropriate primitives to the virtual machine. The most important primitive is the `pythonEval` primitive which can

[4] A lightweight threading mechanism in the spirit of *tasklets*.

evaluate Python expressions and execute Python code. Similar to Python's `compile` builtin, it expects Python source code, a corresponding filename, as well as a string which describes the mode and can either be "eval" or "exec". When a Python expression is evaluated, the primitive returns the result. Otherwise, it will execute the Python code and return `nil`, or fail if for example a syntax error occurred. To clearly separate methods that call primitives from methods that can be called on `PythonObjects`, we add a new class called `Python`. To its metaclass, we add for example a method named `primEval:filename:mode:` which can be used to call the `pythonEval` primitive. As an example, we could call the following to get a new Python `object` instance:

```
Python
  primEval: 'object()'
  filename: '<string>'
  mode: 'eval'
```

We have also modified the method lookup for these kinds of wrapped Python objects. First, the lookup is done on the Python side. If it fails, the lookup continues in the `PythonObject` class inside the Smalltalk environment. This allows us to implement and override methods that are necessary for the tool support in Squeak/Smalltalk while also preserving the original Python behavior. As an example, the following method is used to facilitate that `PythonObjects` can understand the message `class`:

```
PythonObject>>class
    ^ self __class__
```

Sending `__class__` to `self` will retrieve the value of the corresponding attribute of the Python object. If a selector name resolves to a Python *callable* instead of resolving to an attribute, it is called directly. This allows to call Python methods and to pass arguments in. For example, sending `append: 'some text'` to a Python list object will call `append('some text')` on the actual object. Multiple arguments can be passed using Squeak/Smalltalk's keyword syntax. For example, `index: 'a'startingAt: 2` will call `index('a', 2)` on the Python object—the message name is only considered up to the first colon, so we can make up readable keywords.

With all of the above, we can start adopting and building tools in Squeak/Smalltalk which are able to control and modify a Python program at runtime.

3.3 Adapting Squeak's Debugger

In order to implement a Smalltalk-style debugger for Python, we first start with a subclass of Squeak's `Debugger`.

We want this new `PythonDebugger` to display Python frames on top of the Smalltalk contexts that have triggered the execution of Python code. The Smalltalk debugger is normally opened on `thisContext` which is the current method context. The list of all contexts displayed by the debugger is generated by traversing `MethodContext` objects starting with `thisContext` and following the reference to its sender.

Since Python code is being executed in a Smalltalk process, we need to override the entry point to debug processes and modify the context on which the debugger is opened. We add Python frames on top of `thisContext`, as can be seen in Figure 4 . We have implemented a primitive that returns the top frame of the current execution context in Python. Then, we iterate over all parent frames and generate objects from a new class `PythonContext` which is a subclass of `MethodContext`. These objects are basically only used to hold a reference to the corresponding Python frame. This connection between the two stacks is similar to the implementation of `MethodContexts` in the Cog VM [18], where Smalltalk context objects are connected to their C stack frame counter part. Each `PythonContext` is linked to its parent similar to how `MethodContexts` reference their sender. The sender of the bottommost `PythonContext` is then set to `thisContext` and finally the debugger is opened on the topmost `PythonContext`.

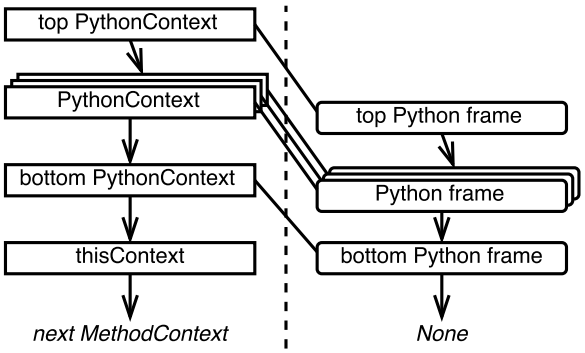


Figure 4: Proxying Python frames for the Smalltalk Debugger.

By overriding the method the debugger calls to retrieve Smalltalk code, we can make the debugger display Python code whenever it encounters a `PythonContext`. In this case, the method uses the referenced Python frame to look up the corresponding Python source file. This is possible, because the Python frame contains the current line number (`frame.f_lineno`), the filename of the Python code (`frame.f_code.co_filename`), as well as the first line in the code (`frame.f_code.co_firstlineno`). With this information, a helper method can read the right file and parse out the corresponding Python code which is then displayed in the code editor of the debugger. To further improve usability, we automatically adjust indentation, because in Python, indentation is part of the syntax.

Moreover, we have overridden the method in `PythonContext` that normally returns a list of temporary Smalltalk variables, so that it returns "`self pyFrame f_locals keys`", namely all keys that are part of the frame's local namespace. This causes the debugger to display the Python locals in the bottom right list.

In total, only six method overrides in `PythonDebugger` were necessary to provide basic support for Python.

Lastly, we need to instruct the Smalltalk environment to use the new debugger. This can be done by making a new `PythonToolSet` the default, which Squeak/Smalltalk will then use to look up which tool to open when an exception occurs.

After a user interrupt, the `PythonDebugger` may look like shown in Figure 5. The Python application running is a HelloWorld "Flask"-based web server. We can now explore the current control flow starting from the `DoIt` Smalltalk frame which started the application, up to the topmost Python frame including the latest line being executed at the time of the user interrupt. The list includes frames

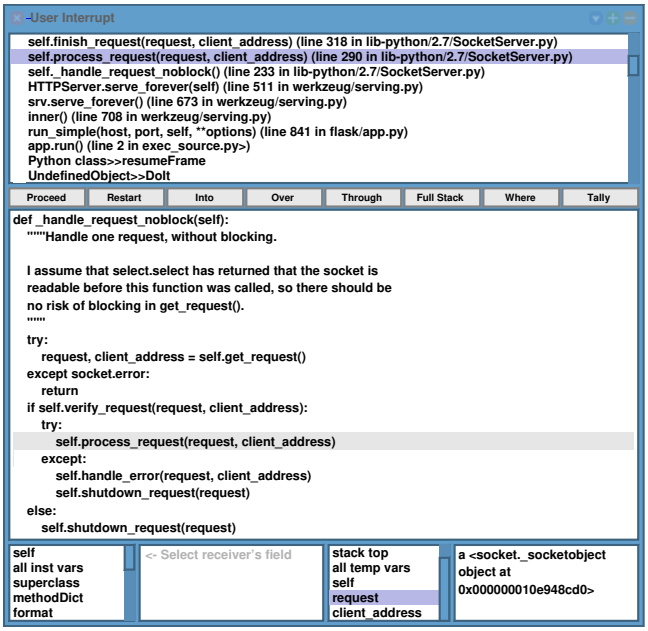


Figure 5: Interrupting a Python application.

with the Python expression that started the server, Flask's main entry point, request handling in Werkzeug (one of Flask's few dependencies), as well as the "SocketServer" module which is part of the Python standard library. This immediately gives developers a lot of information to understand the running application. For example, Flask is considered to be a lightweight web framework. Developers can now interactively experience what that means, and discover that Flask uses a library called "Werkzeug" which in turn uses "SocketServer" in order to serve requests and in which files and lines the corresponding functions can be found.

When Python code is being executed which throws a Python-level exception, the virtual machine informs the image accordingly, so that the image can arrange to open the `PythonDebugger`. A simple example in which a naively-implemented average function is called with an empty list is depicted in Figure 6. This implementation cannot handle the case of an empty list, therefore a `ZeroDivision` exception is thrown. The `PythonDebugger`'s title contains the error description and the topmost frame is presented to the developer with the corresponding error line. It is possible to inspect the `iterable` parameter in order to realize that its size is zero. The user can now insert a check to ensure a division by zero cannot happen again and then save the code. The method that usually hot-swaps Smalltalk methods after saving is overridden, so that the new version of the Python code is written to disk into its source file first. Then the VM compiles the new code, replaces the Python code object of the selected Python frame with the newly produced one, and resets the frame. The next time the Python interpreter continues, it will restart the frame and then the misbehavior is eliminated. This can be done by clicking the "Proceed" button which will let the Smalltalk-level Python process continue.

3.4 Instrumenting more Squeak tools

Similar to the `PythonDebugger`, we have adopted other tools that come with Squeak/Smalltalk. It only took little effort to adopt Squeak's interactive `Inspector` and `Explorer` tools. Only one method needed to be overridden to provide a `PythonWorkspace` which supports Smalltalk-style `doIts`, `printIts`, `inspectIts`, and `exploreIts` for Python. Moreover, we have basic Python support in the `SystemBrowser` which allows us to create Python classes, and add Python methods. Finally, it now is possible to rapidly build custom tools for Python development, such as a tool that allows to observe the execution of Python bytecodes which would allow debugging on Python bytecode level. With for example the Vivide framework [25], one could also build data-flow-based applications which can consist of Python and Smalltalk code.

4 DISCUSSION

This work is aimed at providing the best of two previously separate worlds: the framework approach to cross-language IDE and debugger development provided by environments such as Eclipse or NetBeans, and the live and immediate debugging nature of systems such as Squeak/Smalltalk. On top of that, to be considered useful, we seek for an approach that impedes performance of the integrated languages as little as possible.

4.1 Tool Frameworks and Live Development

As shown, our Python debugger is implemented as a refinement of the default Squeak debugger. Through the inter-language interface (cf. Section 3.2), this debugger can adapt the Python runtime state to the expectations of the Squeak tools. The implementation of the debugger was straightforward as described in Section 3.3. The amount of adaptation necessary on both, the VM and the tools side, was comparatively low. Hence, we think that other languages can easily provide their own adapters to interface with the Squeak debugger, allowing for a variety similar to that of Eclipse, NetBeans, or other IDEs.

Due to this easiness, we also wrote initial adapters for Squeak's code browser, the "workspace" (a tool to evaluate short code snippets), and provided an adapter method for the `PythonObject` class to work with Squeak's object inspector. We are thus able — in a rudimentary fashion — to write and evaluate Python code and to interactively explore Python objects in the live system.

Our experience in writing those tools was every bit as enjoyable for us as writing Smalltalk tools is — the interactive environment allowed us to adapt the tools as we went along, implementing features in the moment we wanted them. These advantages, inherent to live development environments in the style of Squeak/Smalltalk [25], are thus made available to Python developers, allowing them to adapt tools at runtime, without having to restart the IDE or re-execute a running program to iterate over it. Since the tools work on the Python objects directly, there was no need for proxy objects besides `PythonContext`, and these were only necessary because Python frames are not exposed in the Python language itself. Notwithstanding, the interaction and library reuse goes beyond tools and also works in the other direction, so Python code can be used for Smalltalk projects.

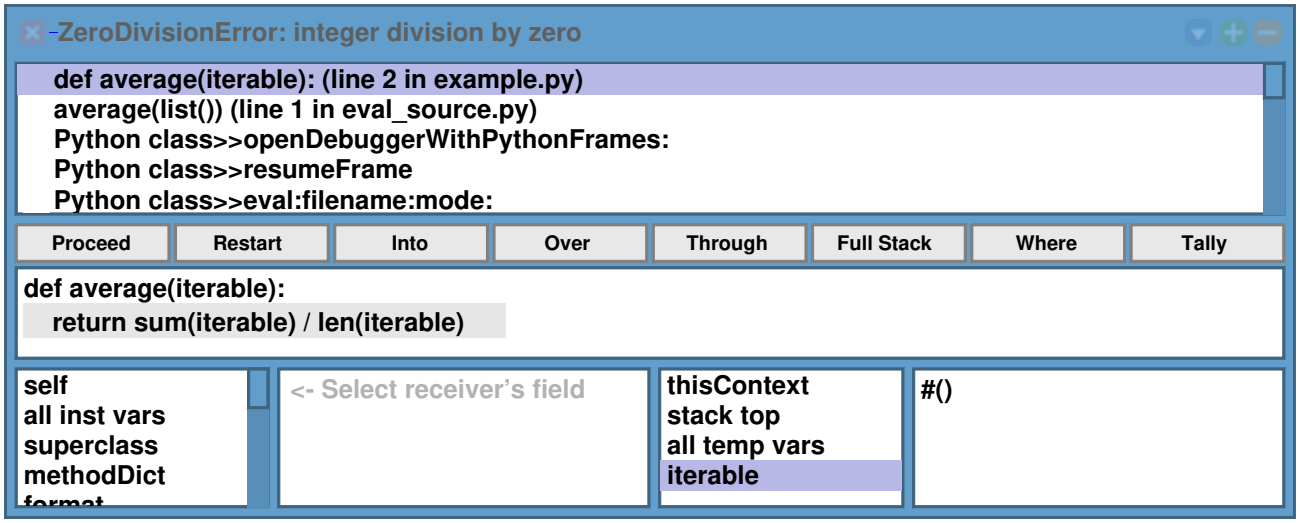


Figure 6: Debugging a Python exception.

4.2 Performance of Combined Languages

The RPython toolchain makes it simple to combine multiple interpreters, but there is a performance impact in doing so. There are two reasons, one inherent to RPython, the other due to our approach.

The first reason is a current limitation of the RPython framework: a very potent call stack optimization (called *virtualizables*) that avoids allocation of stack frame objects can only be applied to at most one interpreter. In our case, this means that while RSqueak/VM runs at full speed, there is an overhead to each PyPy method call. Although we have not yet put a focus on performance, a ballpark measurement of the Richards Benchmark[5] shows that our implementation performs comparable to CPython.

The second reason is that our approach—for the time being—relies on the cooperative scheduling mechanism of Squeak/Smalltalk processes to concurrently execute code in different interpreter loops. Currently, each interpreter maintains counters to decide when to give the other processes a chance to run. At minimum, the user interface (UI) process of Squeak will get a chance to handle user input, but there may also be other processes running concurrently at the same or higher priority that get scheduled instead. This means that even when there is no user input and no other concurrent computation, there is the overhead of maintaining the counters—at the moment at each bytecode dispatch in PyPy. This overhead can be reduced if we switch to counters only for loops and method calls. As another mitigation strategy, we are considering to allow convenient deactivation of the UI process and only start it when a Python error needs to be handled or when the Python process is interrupted by the user with a signal. This might be desirable for a deployment scenario in any case.

5 RELATED WORK

Related work can be found both, in the domain of debuggers as well as interpreter composition.

[5] https://www.cl.cam.ac.uk/~mr10/Bench.html

5.1 Debugging and Debuggers

The Smalltalk debugger [13] and debuggers like it have capabilities such as object- and stack-inspection, edit-and-continue, restart, resume, etc. (see above), and can be extended with new concepts, for example with test-driven fault navigation [19]. However, more prevalent debuggers present different sets of capabilities. GDB [24] and descendant or similar debuggers (LLVM, Xcode, etc.) typically present a very low-level debugging experience. While providing rich intercession (for example breaking, hardware breakpoints, interrupt handling) or inspection, these are very close to the machine/processor and not necessarily to the program. Especially debugging languages that are not very C- or assembler-like requires a lot of effort for effective inspection. Edit-and-continue in GDB is only supported in a severely limited fashion under the idea of *altering execution* [24, chapter 17]. Similarly geared towards the machine, the Microsoft Visual Studio Debugger [17], supports a comparable set of capabilities, paired with a more user-friendly edit-and-continue mechanism.

On the other hand, debuggers for higher-lever or dynamic languages like Python or Ruby provide more direct inspection—often used in a read-only post-mortem fashion—but also limited stop-and-resume and seldom edit-and-continue. For JavaScript, certain tools, such as Chrome DevTools, provide more capable stop-and-resume and a limited edit-and-continue. However, they are hardly extendable or scriptable.

Ongoing research based on the Truffle framework investigates how to leverage a language implementation framework to provide fast, unified debugging facilities to different language implementations [22, 26].

5.2 Interpreter Composition and Embedded Languages

The composition of programming languages and—in extension—their implementations has been investigated since the late 1960s [8]. More recent research targets advanced just-in-time (JIT) compiler

frameworks, such as RPython with Unipycation [2] (composition of Python/PyPy and Prolog/Pyrolog) or Truffle with Sulong (able to compose Ruby/JRuby and similar with C/LLVM) [15], among others [3]. A composition of Ruby with Smalltalk that executes mixed stacks as Smalltalk processes is also implemented in MagLev, the Ruby implementation on the GemStone/S Smalltalk VM [23]. Our composition approach bears most resemblance with Unipycation.

From a different point of view, interpreter composition is similar to language embedding, however, research that targets the latter is typically concerned with the *language* as being written than with the *runtime environment* as being executed. Examples include Eco [9], a syntax-directed-style editor for language composition, as well as domain-specific language (DSL) tool environments such as Helvetia [20] or language workbenches [10]. Our approach, however, is focused on debugging, both with respect to composition and tooling; more extensive tooling and editing capabilities are still under investigation.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we presented an approach to using a Smalltalk environment for debugging other programming languages. We applied this approach to the RSqueak/VM and PyPy, so that Squeak/Smalltalk can be used for Python debugging and direct interaction between the virtual execution environments. Squeak/Smalltalk can therefore control the Python interpreter, which enables edit-and-continue style debugging of Python programs. We demonstrated that only little effort is required to adapt Squeak's debugger to alter and control the execution of Python code. Such features are not available in most other Python implementations, including the reference CPython implementation. Moreover, we adapted the inspection tools in Squeak/Smalltalk to be able to inspect and explore Python objects. We can now build new tools for Python in Squeak/Smalltalk more rapidly.

However, a debugger alone does not make an IDE. Our initial results with the debugger are promising and demand expansion to other tools as well as languages. We plan to adapt Squeak tools, such as the system browser, to Python in such a way that Squeak can act as an IDE for Python applications, but benefiting from the dynamicity of Squeak's environment. Moreover, it will be interesting to see how well much larger Python applications can be debugged and developed using these tools. It is currently possible to affect Python objects from Squeak/Smalltalk but not vice versa, which would be also useful. For example, a Squeak/Smalltalk image could act as a persistable object space or even database similar to GemStone [7]. The changes required on the Python side are however still uncertain. To extend our approach to other languages, we have already started integrating Topaz [11]/Ruby and plan to integrate other interpreters based on RPython, such as Pyrolog [2]/Prolog or Pycket [4]/Racket.

Lastly, we plan to evaluate our approach more thoroughly, on the one hand — regarding the programming experience — with user studies, on the other hand — regarding performance — with meaningful benchmarks.

ACKNOWLEDGMENTS

We would like to thank Carl Friedrich Bolz and the PyPy team for their help integrating the PyPy interpreter into RSqueak/VM. We gratefully acknowledge the financial support of HPI's Research School[6] and the Hasso Plattner Design Thinking Research Program.[7]

REFERENCES

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. ACM, New York, NY, USA, 53–64. DOI: http://dx.doi.org/10.1145/1297081.1297091

[2] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. 2013. Unipycation: A Case Study in Cross-language Tracing. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. ACM, New York, NY, USA, 31–40. DOI: http://dx.doi.org/10.1145/2542142.2542146

[3] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. 2015. Approaches to interpreter composition. *Computer Languages, Systems & Structures* 44, Part C (2015), 199–217. DOI: http://dx.doi.org/10.1016/j.cl.2015.03.001 arXiv:1409.0757

[4] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G Siek, and Sam Tobin-Hochstadt. 2015. Pycket: A Tracing JIT for a Functional Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*, Vol. 50. ACM, New York, NY, USA, 22–34. DOI: http://dx.doi.org/10.1145/2784731.2784740

[5] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. 2008. Back to the Future in One Week — Implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems*. Lecture Notes in Computer Science, Vol. 5146. Springer Berlin Heidelberg, Berlin, Heidelberg, 123–139. DOI: http://dx.doi.org/10.1007/978-3-540-89275-5_7

[6] Gilad Bracha. 2012. Debug mode is the only mode. https://gbracha.blogspot.com/2012/11/debug-mode-is-only-mode.html. (2012). Talk at the 2012 meeting of the IFPI TC2 Working Group on Language Design.

[7] Paul Butterworth, Allen Otis, and Jacob Stein. 1991. The GemStone Object Database Management System. *Commun. ACM* 34, 10 (Oct. 1991), 64–77. DOI: http://dx.doi.org/10.1145/125223.125254

[8] Thomas E. Cheatham, Jr. 1969. Motivation for Extensible Languages. *SIGPLAN Not.* 4, 8 (Aug. 1969), 45–49. DOI: http://dx.doi.org/10.1145/1115858.1115869

[9] Lukas Diekmann and Laurence Tratt. 2014. Eco: A Language Composition Editor. In *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 82–101. DOI: http://dx.doi.org/10.1007/978-3-319-11245-9_5

[10] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Number 8225 in Lecture Notes in Computer Science. Springer International Publishing, Cham, 197–217. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_11

[11] Tim Felgentreff. 2013. Topaz Ruby. http://lanyrd.com/2013/wrocloverb/sccygw/, https://github.com/topazproject/topaz. (March 2013). Invited Talk at the 2013 edition of Wroclove.rb.

[12] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. 2016. How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies (IWST'16)*. ACM, New York, NY, USA, Article 21, 10 pages. DOI: http://dx.doi.org/10.1145/2991041.2991062

[13] Adele Goldberg. 1984. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Longman, Boston, MA, USA. The Red Book.

[14] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman, Boston, MA, USA. The Blue Book.

[6] https://hpi.de/en/research/research-school.html

[7] https://hpi.de/en/dtrp/

[15] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity*. ACM, New York, NY, USA, 1–13. DOI: http://dx.doi.org/10.1145/2724525.2728790

[16] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *SIGPLAN Not.* 32, 10 (Oct. 1997), 318–326. DOI: http://dx.doi.org/10.1145/263700.263754

[17] Microsoft. 2017. Debugging in Visual Studio. (Jan. 2017). https://msdn.microsoft.com/en-us/library/sc65sadd(d=default,l=en-us,v=vs.140).aspx

[18] Eliot Miranda. 2011. The Cog Smalltalk Virtual Machine: Writing a JIT in a High-level Dynamic Language. In *5th Workshop on Virtual Machines and Intermediate Languages (VMIL)*.

[19] Michael Perscheid, Michael Haupt, Robert Hirschfeld, and Hidehiko Masuhara. 2012. Test-driven fault navigation for debugging reproducible failures. *Information and Media Technologies* 7, 4 (2012), 1377–1400. DOI: http://dx.doi.org/10.11185/imt.7.1377

[20] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. 2010. Embedding Languages without Breaking Tools. In *ECOOP 2010 – Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 380–404. DOI: http://dx.doi.org/10.1007/978-3-642-14107-2_19

[21] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, New York, NY, USA, 944–953. DOI: http://dx.doi.org/10.1145/1176617.1176753

[22] Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. 2014. Debugging at Full Speed. In *Proceedings of the Workshop on Dynamic Languages and Applications (Dyla'14)*. ACM, New York, NY, USA, Article 2, 13 pages. DOI: http://dx.doi.org/10.1145/2617548.2617550

[23] Matthias Springer. 2016. Inter-language Collaboration in an Object-oriented Virtual Machine. *arXiv preprint* (2016). arXiv:1606.03644

[24] Richard M. Stallman, Roland Pesch, and Stan Shebs. 2011. *Debugging with GDB: The GNU Source-Level Debugger, V 7.3.1* (10th ed.). GNU Press, Boston, MA, USA.

[25] Marcel Taeumel, Bastian Steinert, and Robert Hirschfeld. 2012. The VIVIDE programming environment: connecting run-time information with programmers' system knowledge. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)*. ACM, New York, NY, USA, 117–126. DOI: http://dx.doi.org/10.1145/2384592.2384604

[26] Michael L. Van De Vanter. 2015. Building Debuggers and Other Tools: We Can "Have It All". In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '15)*. ACM, New York, NY, USA, Article 2, 3 pages. DOI: http://dx.doi.org/10.1145/2843915.2843917

[27] Bret Victor. 2012. Stop drawing dead fish. (May 2012). http://san-francisco.siggraph.org/stop-drawing-dead-fish/ Talk to the San Fransicso ACM SIGGRAPH.