Multi-Level Debugging for Cython

Mark Florisson
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
markflorisson88@gmail.com

ABSTRACT

Programs may be written in multiple languages and may be observed at multiple levels of abstraction. When debugging it is often preferable to use a single debugger that can provide a single debug session that allows the user to observe the program at all the levels of abstraction that are deemed important. This paper will cover such debuggers, design patterns which may be useful when implementing them and will then focus on a reification in the form of a Python and Cython debugger written as an extension to GDB, the GNU Debugger.

Keywords

Multi-Level Debugging, Mixed-Mode Debugging, Debugging Cython Code, Debugging Python with GDB

1. INTRODUCTION

Debugging is an important part of software development and much developer time is spent figuring out why code behaves the way it does. A debugger is an advanced and powerful tool used to debug programs. Features include execution control, code execution, data inspection and data modification. Multi-level abstraction debuggers can provide one single coherent debug session for any program that may need to be inspected and controlled at several levels of abstraction. Such a debugger may be useful to debug a multi-language program, i.e. one that consists of components written in different languages that call each other. In this case it is useful to use the debugger at the source level. Another case where a multi-level debugger may appear especially useful is when debugging virtual machines or code running in a virtual machine. Usually debuggers that work at that source level of the interpreted code are used, but when the need arises to debug the interpreter itself (or misbehaving extensions to such an interpreter), it is useful to look at the different levels simultaneously.

As part of the paper a debugger for Python and Cython was written as an extension to GDB. GDB is a debugger that can already debug at the C/C++, assembly or machine code level. This is particularly useful as Cython compiles Cython code to C in terms of the Python API, which means Cython code cannot be debugged with a Python debugger as it does not allocate Python frames and execute bytecode therein. Using a C debugger to debug both Python and Cython code is bothersome as symbols are mangled and tons of lines of C are generated for a modest amount of Cython lines.

The incentive for this research is to establish the usefulness and implementation possibilities of multi-level debuggers, with a focus on both interpreted and "compiled-down" higher-level languages in GDB.
The rest of the paper will consist of an abstract explanation of the Cython language and the debugger that was created. This will be followed by a couple of research questions, a terminology section, related work and design patterns, the requirements and implementation of the Cython debugger, a conclusion, future work, acknowledgements and finally references.

1.1 The Cython Debugger

Cython [2] is a compiler that compiles a C/Python hybrid language to a CPython extension module, which can then be used from Python code. This means Cython code can deal with both C data structures and Python objects, and can call and be called into by both C and Python code. The Cython Debugger can debug Python, Cython and C code simultaneously. For instance, if you have a Cython extension used by Python code that makes calls into a C library, the debugger will automatically follow control flow while disregarding irrelevant abstractions, such as calls the Python interpreter makes on behalf of the Python code, or calls generated C code make on behalf of the Cython code. What the user will see when stepping through the code is Python source calling Cython source calling C source. Furthermore, data from Python, Cython and C frames can be inspected uniformly and Python code can be executed in both Python and Cython frames.

1.2 Research Questions

The main research question is:

How can multi-level debuggers be implemented?

This question can be decomposed into a set of categorized subquestions:

1.2.1 Related Work:

- Which multi-level abstraction debuggers are currently available?
- How are these debuggers implemented and which design patterns do they use?

14th Twente Student Conference on IT January 21st, 2011, Enschede, The Netherlands.

1.2.2 *Cython debugger:*

- How can we implement the Cython debugger as a GDB extension written in Python?
- Can we use the aforementioned design patterns to aid our implementation?

1.3 Methodology

The research questions for related work were answered by reading research papers describing the implementation and functionality of multi-level debuggers. To figure out whether the design patterns could be used to write the Cython Debugger, the design patterns and the possibilities of GDB were studied. Finally, the Cython Debugger was implemented.

2. TERMINOLOGY

This section aims to explain esoteric terminology that may be used in the rest of the paper. Firstly, when we talk about multi-language and multi-level debuggers we are talking about the same thing. Multi-language does not preclude a view on the machine level code nor does it preclude manual abstraction-level specification. To explicitly indicate that abstraction levels may be specified manually instead of being determined programmatically by debugger "views", we shall use the term "multi-level" when talking about the Cython debugger.

Another piece of terminology is GDB's **Machine Interface** [7]. This component is part of the GNU Debugger and allows external programs to programmatically control the debugger. Tools such as the Eclipse CDT (C/C++ Development Tooling) [10, 4], Apple Xcode [4] and EMACS [6] make use of the Machine Interface to control GDB.

When we talk about an **inferior** we are talking about the process that constitutes the debuggee.

With **object file** we mean a loadable file that contains executable code, i.e. an executable or a shared library.

When we talk about Python, we may refer either to the de facto CPython implementation or to Python the programming language, depending on the context. There are multiple implementations of the Python language besides CPython, for instance Jython, IronPython and PyPy.

Lastly, A **Python extension module** is an extension to the CPython implementation of the Python language, usually written in C or C++. An extension module needs to implement a certain interface so Python can open the resulting object file and expose functionality and data contained therein. Cython generates the code for such modules, ergo these modules can only be used with CPython-compatible Python implementations.

3. RELATED WORK

There are existing multi-language debuggers available, two of which documented in a research paper. One is the aforementioned RTEEM debugger [3], which can be used to debug programs written for embedded systems in multiple languages, the other paper covers a multi-language debugger that can debug Smalltalk, XQuery and Javascript. Both papers describe an extensible architecture that allows a user to easily add support for new languages. This is accomplished by providing a set of interfaces that can be implemented, where one important interface constitutes an abstract representation of a Virtual Machine. This way a debugger can talk to any implementation of this Virtual Machine as long as it implements the interface properly. This is illustrated by Figure 1 on the next page.

"The Reflective VM pattern uses reflection and a specification of an abstract VM, implemented as a collection of Java interfaces, to ease the burden of incorporating a new VM view into the RTEEM debugger. The Chain of Responsibility pattern gives users access to multi-level VMs via multi-level loaders and command interpreters. The Composite design pattern assists in multiprocessor and multithreaded synchronization of execution" [3].

The abstract VM is an idealized Virtual Machine which can be suspended, resumed and queried for it's successor. It's also queryable for threads (which can in turn also be suspended and resumed). A subsequent chain of DebugTarget objects can be requested to set (conditional) breakpoints, can be queried for variables along with their values in the current scope, can be asked for the name of the function or method, can be asked to evaluate an expression, can perform stepping and stepping over and so on. Each DebugTarget implementation in the chain may decide whether it wants to handle the event or request, where the debug target of the lowest level of abstraction is queried last. A DebugTarget may determine whether it wants to handle a request based on the request itself, the current state of execution of the inferior and possibly command line options passed to it. For instance, when setting a breakpoint for Virtual Machines with dynamic symbol information (i.e. symbols that can only be deduced from the current state of execution) may need an additional command line option to be told that the breakpoint applies to their level of abstraction, which means they should handle the request. This architecture does not only allow easy addition of new target languages, but it also allows uniform command syntax when using a command line interface to the debugger. This pattern of a delegation chain is called the Chain of Responsibility and is illustrated by Figure 2 on the next page.

Language specific implementations of higher-level languages that could not be loaded by inference of debug information contained in executable files can still be loaded manually by manual commands.

The composite pattern is used to have multiple objects appear as a single object, which can be used to compose multiple chains of responsibility into one so it can be treated uniformly across multiple threads or processes.

Another paper covering the architecture of a multilanguage debugger is [11]. It uses a different set of interfaces but is in essence similar to architecture described by the RTEEM paper.

4. THE CYTHON DEBUGGER

This section covers the Cython language, requirements for the Cython debugger, the possibilities of GDB extensions written in Python and the implementation of the Cython debugger as such an extension.

4.1 Cython

As mentioned, Cython [2] is a hybrid language between C and Python. Cython allows the declaration and definition of new C and Python data structures, and Cython code can call and be called from both C and Python code (to allow C access, things have to be declared with the `public` and `api` keywords). The Cython compiler compiles Cython code to C code that uses the Python C API

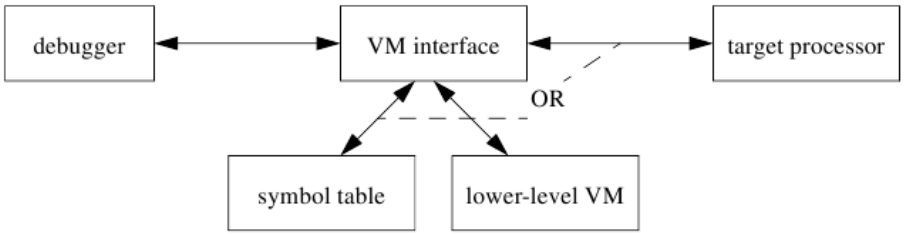

Figure 1. Shows the abstract design of the RTEEM debugger.

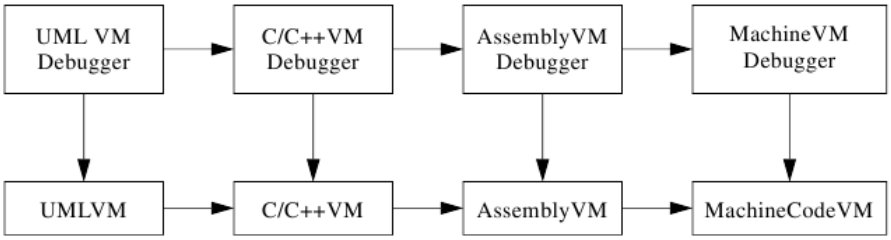

Figure 2. Shows the Chain of Responsibility pattern in the RTEEM debugger.

when needing to deal with anything related to Python. This C code is then compiled to a Python extension module using the same build flags that were used to build Python and unless Python was statically built, this module is then linked to libpython. This extension module can be imported and used by Python code as a regular Python module.
An obvious use of Cython is to provide Pythonic bindings to C libraries or to speed up (computational) Python code.

4.2 Cython Debugger

4.2.1 Requirements

In short, the requirement for the Cython debugger is that it should be possible to utilize common debugger functionality on the Python and Cython source code levels. More specifically, it should support Python and Cython symbols for breakpoints, it should have stepping and stepping over functionality, (stack) data inspection, stack trace formatting and preferably code execution. This functionality is chosen as it is likely the functionality that is most useful and most used with any debugger. Lower-level (e.g. C) functionality may still be used by invoking commands provided by GDB, possibly in combination with functions provided by the Cython debugger.

Convenient methods should be provided to simplify the export of debug information for the user. This includes support for `distutils` (a library for building Python extension modules, part of the Python Standard Library), and an executable that starts the debugger and automatically imports debugging information and debug functionality.

It would be preferable, though not required, to export debugging functionality through GDB's Machine interface thus allowing various graphical frontends to make use of this functionality.

The following table provides a list of commands:

Command	Functionality
cy import	Explicitly import Cython debug information
cy break	Set a breakpoint for a Cython function or method
cy step	Step into
cy next	Step over
cy run	Run a Python/Cython program
cy cont	Continue a Python/Cython program
cy finish	Finish execution of the current stack frame
cy up	Move up a stack frame
cy down	Move down a stack frame
cy print	Print a Cython or Python variable in the current context (local or global)
cy list	List the Cython source code
cy globals	List the globals in the current Cython module
cy locals	List the locals in the current Cython stack frame
cy select	Select a stack frame as numbered by `cy backtrace`
cy backtrace	Print a stack trace of all frames considered relevant
cy exec	Execute Python code in the most recent Python or Cython stack frame relative to the currently selected frame

An important requirement is that these commands (apart from `cy exec`) do not change state in the inferior or call functions therein. This is important because the inferior may not be in a safe state to call arbitrary functions from the debugger (e.g. it may be in a non-reentrant function). So the debugger may only read memory from the inferior (other than injecting software breakpoints).

Commands for running and continuing (`cy run` and `cy cont`) programs are not strictly necessary, but useful when execution is halted in a Python or Cython frame so the command can display the line of source code written by the user instead of a line of interpreter code or generated C code.

The stepping, up, down and backtrace commands should work with the notion of a "relevant frame". A frame is considered relevant when it is:

- A Cython frame
- A Python frame
- A C frame belonging to a C function called from the user's Cython source code

The `cy exec` command should be able to execute Python code in the most recent Python or Cython frame, relative to the currently selected frame. Seeing that Cython frames are actually C frames, a dict of locals will have to be constructed, mapping variable names to the Python objects they are referencing. Changes do not have to be merged back (which is in fact not even possible to do in Python frames for predefined local variables [13]).

It should be easy to invoke the debugger and to automatically import debug information if needed. The debugger should set the Python interpreter that was used to build the Cython extension modules.

4.2.2 GDB extensions written in Python

In GDB 7, the debugger has become scriptable with Python. The Python interpreter is embedded in GDB and can be invoked from the GDB command line. Python code can introduce new commands (that can receive arguments), override existing commands and call other commands (and optionally catch their output). The API also allows you to introduce new GDB parameters, which can be used to configure the behavior of the debugger. There is also an API to deal with the stack, symbol tables, GDB functions and expressions, types, etcetera.

Functionality written in Python can be automatically loaded by the debugger by installing such functionality in a gdb-specific directory in the user's system and giving the script the name `objfile-gdb.py`. Another way to load scripts automatically is to give an executable a section named `.debug_gdb_scripts` that holds a list of script names to load. Both these mechanisms are not entirely flexible as they both require an executable to be loaded before commands or parameters can be used, and `objfile-gdb.py` is not an valid module name in Python which makes importing them harder.

The GDB Python API allows you to use something called a "pretty printer". A pretty printer handler can be installed globally, program wide or it can be associated with a specific object file. Whenever a GDB value is printed or formatted GDB will look at all the pretty printer handlers sequentially (from specific to general) until it finds a handler that returns a pretty printer object. This object is then requested to format the value. This allows unrelated pretty printers associated with different libraries to be loaded and used automatically.

4.2.3 Implementation

Compiler modifications.

To support debugging functionality for Cython code, the Cython compiler was modified to export information about the Cython source code. It currently exports information in the XML format and does not use an existing debug format like STABS [15] or DWARF [5], because Cython debug information is not stored in platform-specific binaries. It does not store debug information in the executables for several reasons:

- The user should be able to debug on multiple levels, which means debug information of multiple levels will have to be available, i.e., it should be possible to "C-step" through Cython code
- Extension modules are loaded at import-time by the Python interpreter, i.e. Python is not linked to the modules. This means debug information (especially for breakpoints) would not be available until after the import

The Cython compiler exports the following information about Cython code:

- Symbol information with regard to context
- A C to Cython line number mapping
- Function and method signature information
- Lists of "step-into functions"

Symbol information includes information about Cython variables, functions and methods. Because Cython compiles down to C it uses name mangling to avoid name clashes. This means the Cython name and the C name become disassociated which means it becomes hard to set (pending) breakpoints or inspect Cython variables.

The surjective C to Cython line number mapping is needed to provide functionality such as stepping, stepping over, breakpoints on specific lines and source code listing.

Signature information is included so backtraces could have the option of listing the parameter names and their arguments. However, to provide clean backtraces without a lot of noise, parameter names and their argument values are omitted.

A list of step-into functions is recorded so the debugger knows which C functions are called by the user and which are generated by the compiler. When stepping, the debugger will not step into generated C function calls, but only into C functions that the user called.

Distutils Integration.

Distutils is a library that can build Python extension modules. Cython already had support for distutils, which was extended to include support for debug flags that tell the Cython compiler to export debug information. When the debug flag is active it will skip C compiler optimization as this can optimize things as data initialization. For instance, if the user initializes a Cython C variable of an integer type, then the compiler may optimize this step out which means the debugger won't be able to step to that line.

The Python Debugger.

There was already a project called EasierPythonDebugging [12] which provided some functionality for debugging Python code in GDB. The Cyton debugger has incorporated this project and patches were provided for additions made. The Easier Python debugging project provides support for pretty printing most built-in Python types like integers, unicode- and byte-strings, dictionaries, lists, etcetera. Types are correctly inferred by looking at certain flags in the type of an upcast `PyObject` (any object in Python is a `PyObject`), after which the object can be safely downcast to a specific type and its data safely read (at least, as safe as one can ever be). If any pointer

should point outside the address space the debugger prevents reading from (or writing to) that location without affecting the debuggee.

The Python debugger also has commands to list source code, print backtraces and local variables and commands to go up and down the Python stack. The Cython debugger augments Python functionality with breakpoints, stepping, stepping over, finishing of frames, running and continuing, listing of globals and python code execution.

Implementation of the debugger.

Use of design patterns

The implementation of the debugger does not quite follow the aforementioned design patterns. This is because the debugger was not written from the ground up, instead it uses GDB's Python API. The API provides a command-driven interface instead of an event-based interface. This means functionality can only get invoked by the user typing in commands, so it is not possible to subscribe to events such as breakpoints that are hit or memory addresses that change. To be able to use those design patterns we would have to transfer control to our Python code which invokes GDB commands and infers events based on returned output. For instance, if the user sets a watchpoint for a certain memory address, the user would then subsequently have to use the Cython debugger commands to control execution, i.e., they would have to use `cy cont` instead of `cont`. This command could then infer possible events and delegate these into a chain of responsibility. However, it would be much more ideal if commands could get invoked by function calls which return information to the caller, instead of parsing output of which a standard and consistent format is not guaranteed.

So, instead the Cython debugger execution control commands try to detect if **some** event occurred, which can be anything from the delivery of a signal to the firing of a watchpoint. This output will be displayed to the user along with the source code depending on the current layer of abstraction.

Command implementation

Every Cython command subclasses the CythonCommand class, which has methods like `is_relevant_frame(gdbframe)` and `get_source_descriptor(gdbframe)`, which can be used to determine whether a frame is considered relevant and to get a descriptor of a file which can be used to read and colorize lines of source code. There is also a Python decorator that is used to dispatch a command based on the abstraction level of the current frame. This is kind of like a delegation chain without the flexibility and it requires the Cython abstraction to know the abstraction level of all frames. Fortunately this is easy to establish using the `is_relevant_frame(gdbframe)` method, but a design flaw is that this mechanism is not pluggable with new or other abstraction layers other than Python, Cython and C.

The decorator that can do the dispatching was used in for instance `cy locals`:

```
class CyLocals(CythonCommand):
  ...
  @dispatch_on_frame(c_command='info␣locals',
                     python_command='py-locals')
  def invoke(...):
        ...
```

Now, when the user invokes the `cy locals` command, the `invoke` method will only be called if the currently selected frame is a Cython frame. If the currently selected frame was a C frame or Python frame the wrapper function returned by the decorator returned by the `dispatch_on_frame` function would have invoked the respective `info locals` and `py-locals` commands instead.

The `dispatch_on_frame` function and its returned decorator are written as one closure, where the `dispatch_on_frame` function takes the `c_command` and `python_command` arguments and returns the decorator. The decorator then gets the decorated function (the `invoke` method) as the first and only argument, and its return value is assigned as the invoke method (methods are actually functions that get bound on access through the descriptor protocol [9]). Here we return a wrapper function that performs the dispatching:

```
def dispatch_on_frame(c_command,
    python_command):
    def decorator(function):
        def wrapper(self, *args, **kwargs):
            check selected frame and
            perform dispatch
        return wrapper
    return decorator
```

This kind of dispatching is only possible for a couple of commands, specifically commands that depend on context. Commands that are not dependent on context, e.g. commands that set breakpoints, need some kind of indication of the intended abstraction level. The Cython debugger handles breakpoints with command line arguments or the use of different commands. For instance, there is the `break` command for the C layer, the `py-break` command for the Python layer and the `cy break` command for the Cython layer. However, Python and C breakpoints may also be set using the `cy break` command. This command differentiates between layers by first looking at the Cython layer (of which symbol information is statically available) and, when no such symbol is found, defaulting to the C layer. Python breakpoints may be set by providing a command line option, i.e.:

```
cy break -p python_function
```

Of course, this cannot work for C symbols that have the same name as a Cython symbol.

Execution control

Execution control is provided through an interface that mandates the exposure of:

- A boolean function that indicates whether a frame is considered relevant
- The source file name associated with a relevant frame
- The current line in the source code
- A list of static breakpoint locations
- A list of runtime (i.e. dynamic) breakpoint locations

The interface is a class that should be subclassed. The interface will then automatically provide code execution control functionality, such as stepping, stepping over, finishing frames, and running and continuing a program. This means that whenever execution of the debugee halts, a reason (if applicable) is printed, along with the current line of source code. For the Cython debugger we use the aforementioned concept of relevance.
Unlike the lists of breakpoint locations, the source file name and line number should be self-explanatory. The list of runtime breakpoints is optional, but the list of static breakpoints is mandatory. These lists are needed to provide support for efficient source code stepping. First, stepping was implemented by a loop around GDB's `step` command, which meant that for every C function call and every C source line the debugger knows about, the inferior is halted, transfer controller to the debugger where Python code running in the debugger repeats this process until a new Python or Cython line number is reached:

```
while true
    execute C step
    break if:
        new line,
        new relevant frame, or
        return to older frame
```

This appeared to be a real big bottleneck and made stepping extremely slow. So instead, an approach with breakpoints was chosen: before starting stepping, breakpoints are set for all the C functions which are considered relevant after which execution can be resumed as usual. Then the "step-loop" can suddenly become a "step-over loop" that has to check:

- Whether a breakpoint was hit
- Whether a new line of source code was reached

If any of these conditions are met the breakpoints are unset and control is transferred back to the user.

The step-over loop is still needed, as it can be very hard to set a breakpoint for the next line number in the user's source code. For instance for Cython the line number mapping is known in advance, but in Python source code is compiled to bytecode which is then interpreted. It is hard to tell the debugger to "run until the next line of Python code is reached". For instance, if a software watchpoint was used with a GDB-defined function that determines whether a new line of Python code was reached, then this would be just as slow as the step-over loop as it would also require the context switch for every new line of C code:

```
watch current frame is original frame and
      line number has changed
continue
```

The fastest possible manner would be to set a conditional breakpoint at a certain point in the code evaluation loop which can check for a change of line number in the condition. This could save a lot of context switches for Python code and really speed up Python code stepping. For instance, the point of opcode dispatch is likely to be a good place to set the breakpoint and have the condition evaluate there.
This does, however, require intimate knowledge of the interpreter and it is hard to reliably specify the breakpoint location of opcode dispatch in the code evaluation function. It might be reliable to use goto labels used in the Python evaluation function (`PyEval_EvalFrameEx`) [16]) as the breakpoint location, but this functionality will be new in the upcoming release of GDB (7.3). There are also no probe points [8] [1] which can help in this regard. Maybe a watchpoint could be used on the instruction pointer. For now, however, the relatively slow step-over approach is chosen as it is certainly reliable.

Code execution
The `cy exec` command allows Python code execution in the most recent (relative to the selected stack frame) Python or Cython stack frame. This is implemented by copying the code from the debugger into a newly allocated buffer in the debuggee. This buffer is passed to a function that compiles it into a code object which is then evaluated in a similar fashion as the Python command line interpreter does. This means statements are executed as usual, but if an expression evaluates to a non-None value, this value's representation will be printed.

When Python code is executed, the debugger ensures that the original exception value is saved before the code execution starts and restored after the code execution has finished. If there was an uncaught exception in the code, a traceback will be printed. This is implemented conveniently using context managers.

In Python, code can be executed in two dictionaries: a dictionary of global variables and a dictionary of local variables. When the nearest frame is a Python frame, its globals and locals dictionaries are used to execute the code in. When the nearest frame is a Cython frame, the debugger will build a dictionary of all initialized local Python variables on the C stack and will grab the globals dictionary from the Cython module object and then execute the Python code in these dictionaries.

Tests
Unit tests for the Cython debugger are executed in GDB running as a subprocess and the results are collected in the parent process. This seems to be a robust way to test functionality as it allows the writer to write unit tests in a normal manner, instead of telling the debugger to invoke certain commands and then try to match the output. It is not possible to use the GDB Python API from outside GDB as the `gdb` module that provides all the functionality is a built-in module only available from the embedded interpreter and not as a standalone module.

5. FUTURE WORK
There are several things that can be implemented in future work. One is the aforementioned approach to faster stepping and stepping-over.

5.1 New Language API
It would be nice if new languages could be added more easily. For this a design similar to the design of the RTEEM debugger could be used. Some preliminary discussion has taken place in which it was established that a lot of functionality could work by simply having the user implement filters that, if accepting, return a wrapper around a GDB frame that can tell about the source file name, the line number, the name of the frame, the symbol table, etcetera. A filter could return an "ignore", a "can't handle" and a "handle" indicator (in which case it would be queried for a frame wrapper object). The debugger would provide a "raw" mode (i.e. normal behavior) and "cooked" mode, which automatically chooses the highest layer of abstrac-

tion (as determined by the list of filters). With only this information symbol listing, stack inspection and navigation, source code listing, execution control and backtrace formatting could be handled. More delicate issue are breakpoints and code execution. For breakpoints the symbol information might not be statically available and code execution is entirely dependent on the target language or interpreter.

5.2 Exposure of debugger functionality to the Machine Interface

When the user implements the aforementioned interface it would be preferable if the exposed information could be automatically exported in the Machine Interface, which would allow for instance Graphical frontends to utilize new language support. In this case it might be preferable if the frontends were oblivious to the newfound functionality. Alternatively, the Machine Interface (with according adjustments made to its specification) could expose the "raw" and "cooked" information at the same time which would allow the debugger to choose what it wants to display (or to display both abstraction views simultaneously).

6. CONCLUSION

Although it is possible to write a nice multi-language debugger as an extension to GDB written in Python, it would require a bit of work to make the debugger flexibly adaptable to new layers of abstraction. Currently, however, a lot of code like reading source code from files and colorizing it needs to be reimplemented in every extension because there is as of yet no generic way to export the required information to code that implements this kind of functionality.

Related work such as the RTEEM Debugger uses three well-known design patterns to provide an architecture that can easily be extended to include support for new views on any executable program. Although we could have written such an interface, it did not really seem worth the trouble and it would have remained questionable how many projects would have been able to make use of it. Instead this functionality should be part of the GDB Python API written in a language that can interface with both Python and C (Cython comes to mind) where it is both easiest to implement and most useful as it would be fully standardized.
However, it must be noted that this interface will not greatly simplify supporting new virtual machines or languages, as the common functionality is not by any means the biggest part of the code. The biggest part of the code is dealing with language specific features: symbols, breakpoints, it's type system and associated pretty printers, code execution and optionally a performing step and step-over implementation.

The code was however still able to make certain things generic, like reading source code from files and lexing it (it uses the `pygments` library to do this, which implements a wide variety of lexers and can produce output in a variety of formats [14]). Also execution control commands (`cy run`, `cy continue`, `cy finish`, `cy step`, `cy next` and their Python equivalents) were implemented in a generic way where the user could subclass the interface and export information about the relevance of frames, the current line number and source file name.

7. ACKNOWLEDGEMENTS

Special thanks goes to Tom Tromey for providing technical assistance with GDB and its Python API, and for ideas, thoughts and discussion on support for a scripting language API for GDB in Python. Thanks to Michael Weber and Hasan Sozer for their assistance on the paper and thoughts on the design and functionality of the Cython debugger.

8. REFERENCES

[1] *Built-in probe point types (DWARF probes)*. 2010. URL: http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5/html/SystemTap_Language_Reference/ch04s02.html.

[2] *Cython Homepage*. Sept. 2010. URL: http://cython.org/.

[3] David J. Murray Dale Parson and Yu Chen. "Object-Oriented Design Patterns for Debugging Heterogeneous Languages and Virtual Machines". In: (2010). URL: http://www3.interscience.wiley.com/journal/109800987/articletext?DOI=10.1002%2Fspe.634.

[4] *Debugger Machine Interface Frontends*. 2010. URL: http://www.linuxfoundation.org/en/DMI.

[5] *DWARF Debugging Format*. 2010. URL: http://wiki.dwarfstd.org/.

[6] *EMACS' GDB-MI*. 2010. URL: http://www.emacswiki.org/emacs/GDB-MI.

[7] *GDB Machine Interface Official Documentation*. 2010. URL: http://sourceware.org/gdb/current/onlinedocs/gdb/GDB_002fMI.html.

[8] Jon Haslam. *Statically Defined Tracing for User Applications*. 2007. URL: http://wikis.sun.com/display/DTrace/Statically+Defined+Tracing+for+User+Applications.

[9] Raymond Hettinger. *Descriptor HowTo Guide*. 2010. URL: http://docs.python.org/howto/descriptor.html?highlight=descriptor%20protocol.

[10] *Interfacing with the CDT debugger, Part 2: Accessing gdb with the Eclipse CDT and MI*. 2010. URL: http://www.ibm.com/developerworks/opensource/library/os-eclipse-cdt-debug2/index.html.

[11] Michal Pise Jan Vrany. "Multilanguage Debugger Architecture". In: (2010). URL: http://users.fit.cvut.cz/~vranyj1/data/papers/Vrany10a-unified-debugger.pdf.

[12] David Malcolm. *Easier Python Debugging*. 2010. URL: http://fedoraproject.org/wiki/Features/EasierPythonDebugging.

[13] *Python builtins documentation*. 2010. URL: http://docs.python.org/library/functions.html#locals.

[14] *Python Syntax Highlighter*. 2010. URL: http://pygments.org/.

[15] *STABS Debugging Format*. 2010. URL: http://docs.freebsd.org/info/stabs/stabs.pdf.

[16] *The Very High Level Layer*. 2010. URL: http://docs.python.org/c-api/veryhigh.html#PyEval_EvalFrameEx.

APPENDIX

A. GRAPHICAL OVERVIEW

The following illustration provides a graphical overview of all the components involved when debugging a Cython program.

The user can interact with the debugger either through a Graphical frontend or through the command line. The Cython debugger extension runs in the embedded Python interpreter in GDB, and its code is called whenever its commands are invoked. The Cython debugger can import debug information that was stored by the Cython compiler when the user's Cython code was compiled. The debugger can then debug a Python interpreter that uses any combination of Python, Cython and C code. The illustration is not entirely accurate, as Cython generates C code that is compiled by a C compiler to a Python extension module.