Crossing Abstraction Barriers When Debugging in Dynamic Languages

Bastian Kruck*, Tobias Pape†, Tim Felgentreff†, Robert Hirschfeld†
Hasso Plattner Institute
Universtiy of Potsdam
*{firstname.lastname}@student.hpi.uni-potsdam.de
†{firstname.lastname}@hpi.uni-potsdam.de

ABSTRACT

Programmers use abstractions to reduce implementation effort and focus on domain-specifics. The resulting application often runs in a convenient guest runtime that is provided by an increasingly complex ecosystem of libraries, VMs, JIT-compilers, operating systems, and native machine architectures.

While abstractions are designed to hide complexity, experience tells us that "All non-trivial abstractions, to some degree, are leaky."[1]. Leaky abstractions are problematic, for example, when the use of under-documented or unspecified behavior of a library or virtual machine causes a failure in domain-specific code. Users may need to understand whether the virtual machine is just under-documented but working as intended or faulty. At that point, the artificially created barrier that protects language users from domain-independent complexity becomes an obstacle. We call this *crossing the abstraction barrier*.

Prior research has investigated how symbolic debuggers can work across language barriers. However, this resulted in dedicated workflows and UIs that differ substantially from traditional symbolic debugging. Users need to remember these rather elaborate workflows, and the learning effort is often larger than the perceived benefit of answering the given debugging questions. As a result, the value of these tools may not be immediately recognized and developers will only consider learning them after having spent much time with conventional debugging methods.

We propose an *interaction model* that generalizes the conventional symbolic debugger so that the known workflow can be kept and users can opt-in to cross-abstraction debugging when necessary. By replacing the traditional list view on the active call chain with a tree model and adding perspective selection, we obtain an unobtrusive, minimal user interface that still offers powerful cross-language debugging features.

[1] http://joelonsoftware.com/articles/LeakyAbstractions.html, accessed September 29, 2016

SAC'17, April 3-7, 2017, Marrakesh, Morocco

 ISBN 978-1-4503-4486-9/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3019612.3019734

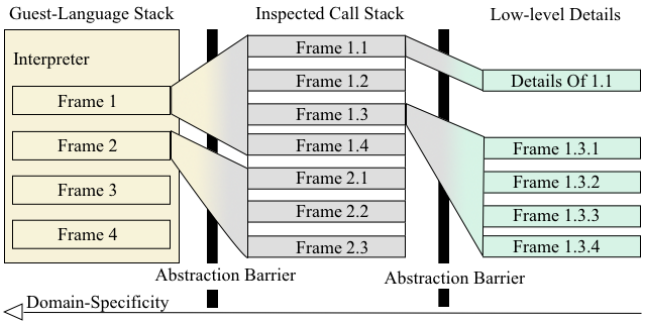

Figure 1: The 2D Call Stack Model: We added a Domain-Specificity Dimension to the conventional Level-of-Detail Dimension; Transforming the stack-frame List into a Tree

CCS Concepts

•**Software and its engineering → Software testing and debugging;** *Interpreters; Domain specific languages;*

Keywords

Inter-Language Debugging; abstraction barriers; Multi-level debugging; Language Workbench; Squeak; IDE; DSL

1. INTRODUCTION

In the past decades, many *kinds of abstractions* have been employed in computer science to cope with the challenges that software developers face. One such abstraction is a virtual machine: Virtual machines and interpreters abstract from the underlying native machine, enabling the programmer to formulate his program in a run-time environment that is closer to the application domain and ignore many application-independent challenges. Other common kinds of abstractions include third party libraries which are abstractions towards a particular domain, engineering patterns, language idioms, or simply method calls. All of these abstractions naturally draw a barrier between the *guest* system with reduced complexity that they establish and the outer *host* system that the abstraction is built with — we call this the *abstraction barrier*.

When searching for a bug in a system, programmers pose questions and use symbolic debuggers to observe the system in action and answer these questions until they find the defect. In most of the cases, observing the execution of a single system with the dedicated tools suffices to isolate the root cause. In rare cases, it might be important to peek outside the system boundaries to investigate what precisely is happening on the other side of the abstraction bar-

rier. In conventional symbolic debuggers, programmers commonly use the *call stack* to zoom into and out of behavioral details of the system. In more general words, they cross the abstraction barriers between the currently active method invocations — we call this *cross abstraction debugging*.

Previous solutions for debugging arbitrary kinds of abstractions don't integrate with the existing conventional symbolic debugger. Past research has built debuggers for particular kinds of abstractions, such as the method call abstraction, distributed systems, domain specific languages, or virtual machines. Most of them provide a user interface dedicated to the mental model they establish [8, 3, 11]. Some describe an API to allow adding abstractions to make the debugger aware of them [11, 3]. Since these tools provide specific workflows and UIs that need learning, the initial effort that needs to be invested is often larger than the perceived benefit of answering debugging questions. As a result, the value of these tools may not be immediately recognized and developers will only consider learning them after having spent some amount of time with conventional debugging methods.

When language users encounter under-documented or unspecified behavior, they may want to cross the VM abstraction barriers without having to learn a new tool. Language designers also benefit from a unified tool that allows them to debug their implementation code right from their guest language test application. This holds not only for VM abstractions, but also applies to designers and users of other kinds of abstractions.

* * *

Given a number of different kinds of abstractions with each having an outer host and inner guest system, developers need tools for debugging and understanding how their programs are represented in each of them.

Our work is based on the assumption that one can conveniently represent many kinds of abstractions by structuring the call stack-frames in a tree through adding virtual stack-frames. We tell apart the *actual* stack-frames that are the original stack-frames of the inspected stack and the *virtual* stack-frames that were added to represent sides of an abstraction that are not part of the inspected execution. If a virtual stack-frame is guest of an abstraction that the inspected stack hosts, we call it *abstract*; if a virtual stack-frame is host of an abstraction that the inspected actual stack-frame is guest of, we call it *concrete*.

Thus, we make the following contributions in this paper:

- We describe an interaction model that extends the conventional symbolic debugger to allow cross-abstraction debugging without dropping the known workflow.
- We show how our model can be used as uniform user interface to various perspectives by applying our model to the library, VM and the interpreter abstractions.
- We demonstrate the feasibility of our design with a tool implementation for VM and interpreter abstractions in Squeak/Smalltalk.

2. AN INTERACTION MODEL FOR CROSS-ABSTRACTION DEBUGGING

We first introduce the workflow of our interaction model (subsection 2.1), continue to describe the implications for the debugger's user interface (subsection 2.2) and then illustrate them with examples in sections 2.4 and 2.5.

2.1 Workflow

Our interaction model is a generalization from the conventional symbolic debugger that consists of a *list of active call stack-frames* and a *source view* that shows the source description of the currently selected frame. The relevant tasks that users usually perform are

1. Select a stack-frame *(conventional)*
2. Read method source *(conventional)*
3. Control execution *(conventional)*
4. Inspect objects *(conventional)*

The usual workflow is to select stack-frames and read method source interchanging to explore the stack of methods that are currently executed. For this work, we leave controlling the program execution as well as inspecting the program state untouched.[6, 3] Nevertheless, a perspective dependent view for object inspection sounds straightforward to us from an interactive point of view and our prototype actually implements level-aware stepping so that filtered stack-frames will be skipped.

For our new workflow, we perform two changes in the user interface: We generalize the call stack from a list into a collapsed tree view that looks almost the same and add a button that will show a context menu on click. This enables two new tasks additionally to the conventional workflow:

5. Expand or collapse a stack-frame *(new)*
6. Change perspective *(new)*

When users encounter a peculiar behavior of an abstraction that the inspected stack-frame is a guest of (for example an operating system call), they might cross the abstraction barrier by expanding the corresponding stack list item and find the lower level details for this abstraction as its children (Figure 1, right column). Since *lower level detail* can have many concrete meanings depending on the users current perspective, they can click the perspective-button and then select one of the provided perspectives. Based on that selected perspective, the tree view will contain additional *virtual stack-frames* that haven't been in the original stack-frame list. Furthermore, the title of tree nodes as well as the content of the source view is specialized on the current perspective.

2.2 User interface implications

Our new workflow requires the following unobtrusive changes in the debugger's user interface:

Concrete virtual stack-frames

We generalize the call stack from a list view to a tree view element and use the newly gained depth dimension to a add detailed information in the form of tree nodes that are children of actual stack-frames (in our prototype: *b* in Figure 2). We use Concrete Virtual stack-frames to add information about abstractions where their parent is guest of the abstraction. This conceptually adds domain-unspecific details, i. e. a column at the right-hand side of Figure 1. Our example in subsection 2.5 inspects such details for the CogVM virtual machine for Smalltalk. It shows how generated JIT-Code, the manual Slang-implementation of primitives[2] and its generated C code are used to display detailed information for the running VM.

[2]also known as *builtin*s or *native*s.

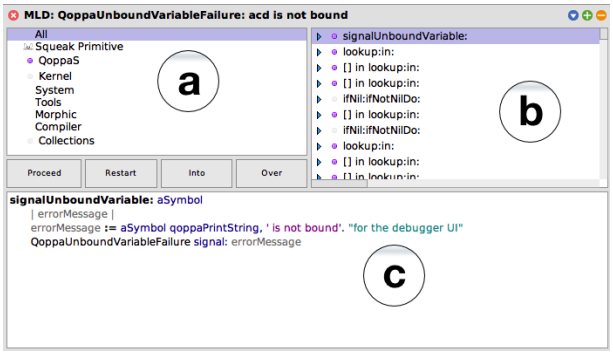

Figure 2: Our debugger with a new perspective-selection view (a), the stack list (b) that was turned into a tree view, and the source view (c)

Perspective selector

Developers can select a *perspective* that they are approaching the system from (in our prototype: *a* in Figure 2). When selecting a perspective, the tree view will adapt its contents, the node's titles and icons will change and the source view will show the content that the perspective determines for the selected node.

Abstract virtual stack-frames

We group neighbor stack-frames when a perspective focuses on abstractions that are hosted by the inspected environment. This opposes the low-level concrete virtual frames that we use when the inspected environment is guest of the abstraction. The added tree nodes form a new domain-specific top level of the tree, i. e. resulting in a new column at the left-hand side in Figure 1. At first sight, a particular member of one group can become the parent node and serve as a representative (for example, the event trigger call may represent all the handler nodes). Up next, we add the event name to the title of the parent node and change the source content to a description of the state of the applied state/event machine model. This shows up that the representative may be the technical data source but that the parent node is semantically a new virtual node that comes from another model. Our walk-through in subsection 2.4 shows how First-level virtual nodes are used to build a guest-level debugger for an interpreter that is hosted by the inspected call stack. In that example, we debug a program written in the guest-level language Qoppa which's virtual machine QoppaS is implemented in Squeak/Smalltalk.

* * *

The generality of this model allows new perspectives to introduce new kinds of abstractions to the debugger. Such perspectives can be build by application developers or library and framework builders when realizing that the conventional debugger can be augmented to ease working with it. As a result, libraries that introduce new paradigms like aspect oriented programming or context oriented programming can ship together with a generated perspective for each layer or aspect. This extendability also enables implementation of existing (multi-level) debugging protocols to use them as data back-end for our user interface.

2.3 Abstraction barriers between stack-frames

Previous work described a debugger that allowed users to pick an abstraction-level of interest and alter the stack list, lexical view and data view to represent contents on that selected level of abstraction. [6]

The *abstraction barrier* for a method call abstraction is the semantic step from one stack-frame to the next one [1]. The barrier may be small, for example, when application level code calls other application level code; it may be larger when the application calls into the standard library, or possibly largest when high-level language code calls low-level VM primitives. The varying barrier sizes between the stack-frames of a trace make it hard to navigate the stack to find frames of interest.

The call stack is often consulted in a debugging session where the underlying computing model is a stack machine. Developers are simulating the machine in their head and are looking at its implementation details. To account for abstraction barriers, we split the stack list into two dimensions: the first representing a homogeneous view on the conceptual behavior at the same abstraction levels so that moving between frames does not require moving over large abstractions barriers. The second one allows moving over large barriers and thus into or out of implementation details.

2.4 Example: browsing high-level virtual stack-frames, debugger-native stack-frames and low-level virtual stack-frames

Throughout the development and lifecycle of a language feature, the logic of an algorithm may be duplicated a number of times when transforming it into different representations. Some transformations are manual, others are automatic. For example, compilation into intermediate code that is portable across architectures or operating systems (LLVM, JVM, .NET) is such a transformation.

Especially in languages that have multiple VM implementations, libraries cannot always assume that a particular primitive is implemented on all VMs. In cases where such primitives can be simulated from within the language, there is fallback (sometimes called polyfill) code to deal with the absence of VM support. (Examples for such languages include Squeak and JavaScript.)

As example for a highly run-time flexible application, we implemented a recursive-evaluating interpreter QoppaS in Squeak/Smalltalk. Its interpreted language is a Scheme dialect, called Qoppa[3]. In the next subsections, we go through a couple of exploratory tasks with it to illustrate the user experience of working with a debugger that implements such an interaction model.

Select a debugging perspective to specialize the debugger on the current use-case

It is use-case-specific what to show in the frametitle, displayed content, and how to create virtual stack-frames. These properties vary with not only the debugged application and unit, but also with the question developers currently try to answer. When users select the QoppaS perspective (Figure 3), the debugger shows the QoppaS stack-frames that were created by introspecting the Smalltalk frames and adding abstract virtual frames as tree parents of the existing Smalltalk frames.

Expand a stack-frame to inspect its implementation details

When we expand the tree node to get further details on the evaluation of a Qoppa Instruction, we see the native Smalltalk stack-frames as we know them from the conventional debugger. Selecting it here reveals the code that raised the exception (Figure 4). Looking through the recursive calls of `lookup:in:` reveals that the variable is indeed not declared in any of the scopes.

[3]described in http://mainisusuallyafunction.blogspot.de/2012/04/scheme-without-special-forms.html, accessed on December 2, 2016

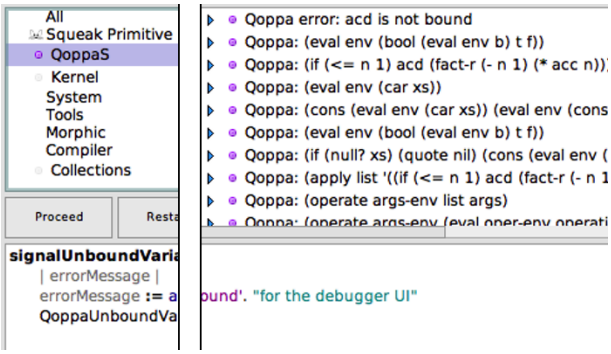

Figure 3: Switching to the QoppaS perspective adds high-level nodes that group evaluations of a qoppa instruction together. We see the Qoppa stack.

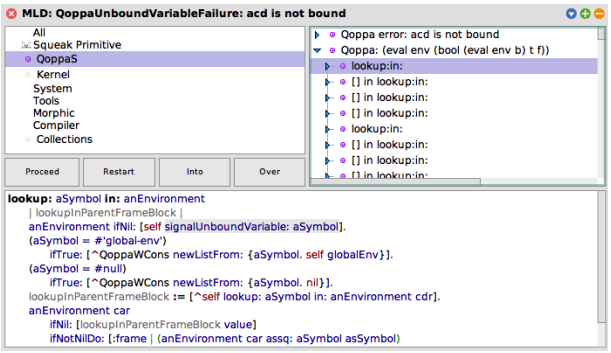

Figure 4: Implementation details of the QoppaS VM show where the exeption was thrown.

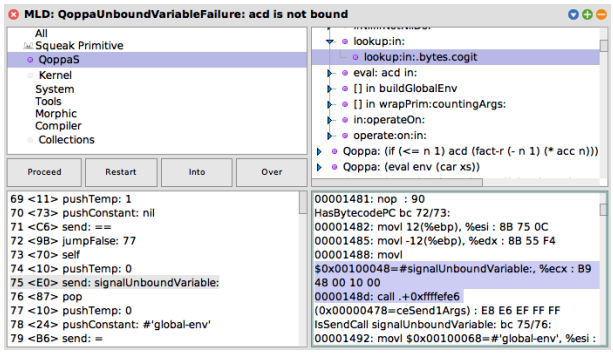

Figure 5: Left: the bytecode of `lookup:in:`. Right: Matching the generated machine code and confirm that `operate:on:in:` is being called.

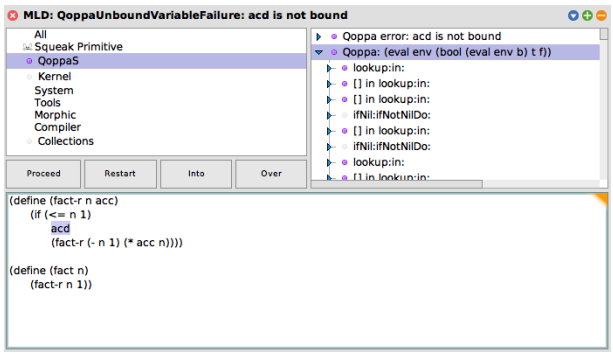

Figure 6: Browsing High-level code, we find the mistake on the user application level: `acd` should be named `acc`

Expand a Smalltalk stack-frame and select the Cogit-child to show the output of the Cogit JIT compiler

Since the `lookup:in:` method is called recursively and often, the method was JIT compiled so that the bug may be caused by erroneous JIT-Compiler behavior. When selecting the *.cogit* child, the source view at the bottom is split to show the Smalltalk-bytecode at the left an the compiled machine instructions on the right so that users can understand the machine code better by mapping the two onto each other (Figure 5). The JIT-Compiler did not inline the `signalUnboundVariable` so that the hypothesis on a broken internal state through faulty inlining was invalidated.

Looking at the high-level code again, we find that the variable was just a typo on the QoppaS language level: `acd` should have been spelled `acc` (Figure 6). Integrating the language levels helped us to go back and forth on the language levels of abstraction. So the bug finally was found to not be a VM bug, but a mistake from the language consumer instead.

Multi-level debugging is not strictly necessary in this scenario. The actual error can be spotted *without* crossing abstraction barriers. However, in case the typo is harder to spot or the bug is more complex, it can be desirable to *verify the absence* of faulty behavior on lower-level abstractions to rule out too many possible causes. Multi-level debugging is not intended as a replacement for rigorous development.

2.5 Example: browsing a Smalltalk primitive

In another scenario, we wonder how the <= comparison primitive is implemented in the VM. When running the Qoppa code `(if (<= 'n 1)`, we see an error that the QoppaS wrapper class should implement `adaptToNumber:andSend:` to enable such comparison with a string. We see that the Smalltalk method <= tried to call it, but wonder how the Smalltalk fallback method <= was called instead of the primitive implementation.

Looking at the frame titles, we see the tiny Squeak symbol that we can lookup in the perspective selector as the Squeak Primitive level (symbol in Figure 7, selector in Figure 6). The number already hints us that this method is annotated to be the primitive 5 (<=). In Smalltalk, the method body of such a method describes the fallback code that gets executed when the primitive fails or is not implemented. So let's expand the node to see which one is the case.

We find two new virtual nodes, both are calling the primitive `primitiveLessOrEqual`, so the 5 maps to the correct primitive (Figure 7, top center). Selecting the C stack-frame shows us the generated C code that is used to compile the primitives. Using

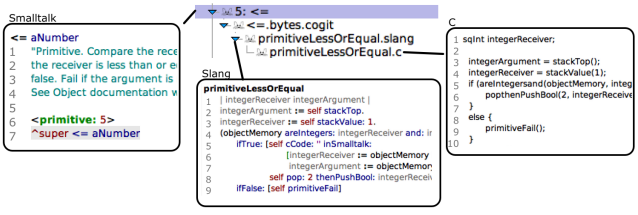

Figure 7: We select the Low-level Nodes to inspect the fallback code, Slang and its generated C code (from left to right).

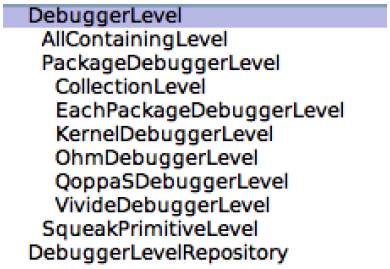

Figure 8: Extensions to our system are done by subclassing from the `DebuggerLevel` class.

the step buttons in here will not function for now. The code calls `primitiveFail` method if an argument is not an integer (Figure 7, C, lines 5-10). So lessOrEqual comparison to strings is implemented to jump to the fallback code. To see if that behavior is intended by the implementors, we go to the high-level description of that primitive (Figure 7, Slang, line 10). It looks clearly intended. So we can rely on this behavior as a language feature and just implement this method without having to be afraid of relying on a bug.

3. IMPLEMENTATION

We evolved our prototype from earlier work on building and debugging interpreters. Our extended design applies the Mediator-Wrapper-Pattern to allow the debugger to selectively enrich stack-frames with additional information.

Our debugger works on an enriched stack of a halted Squeak process (also called green thread in other systems). The basic Squeak debugger reflectively accesses all frames in a halted process directly. Our extension inserts *virtual context nodes* into the stack. The first kind of these nodes combines groups of Smalltalk stack-frames into high-level behavior. (This can be used in a similar fashion to stack filters in other environments). A second kind of virtual context nodes are shown as children of Smalltalk stack-frames – these represent behavior of the execution models that the inspected stack is running on such as bytecodes, virtual machine C sources, or just-in-time compiled assembler.

The enriched stack thus turns a simple linked list of frames into a tree, with high-level virtual nodes to combine multiple actual stack-frames and low-level virtual nodes to offer a closer look at the underlying virtual machine execution for Smalltalk stack-frames.

The system is extensible by subclassing from `DebuggerLevel` class. All subclasses are used to determine the concrete stack for the debugger. Subclasses represent the mediator in our architecture, and, for each actual Smalltalk frame, they may return additional `VirtualContextNode` instances as children, a different frame title or icons to provide visual guidance. Figure 8 shows the currently provided mediators. It includes classes to filter based on various packages, whether or not a method includes a primitive call, or which source code repository a method was loaded from.

As the examples have shown, a path from any first-level node to a leaf of the tree always contains exactly one call stack node and all the others are virtual nodes. During our implementations we observed the trees to only form by grouping neighbor method activations with a new virtual parent (*High-level virtual nodes*) and by adding VM-external details to a particular activation (*Low-level virtual nodes*).

High-level virtual nodes

As we've argued in prior work, the language developer's current perspective on the system they are debugging varies with respect to the abstractions they want to regard as black boxes and which they want to debug into. Our current implementations of high-level virtual nodes help in this regard by allowing developers to hide frames based on a variety of factors:

- Per-package hiding helps focus on those frames that are inside the currently developed projects, ignoring frames from, for example, the Squeak standard library.
- When a guest language like Qoppa is interpreted on top of Squeak, frames that implement the Guest language primitives can be hidden to focus on the execution model like a guest-level debugger would.
- Frames can be grouped by source revision, which can be useful when bisecting a bug that is present in some revisions but not others.
- Event-nodes can be used to help debug event-based systems by grouping frames triggered from the same events.

Low-level virtual nodes

Our new focus in this work was the addition of virtual nodes below the actual Smalltalk frames. We use two Squeak virtual machines, the Cog Spur VM written in Slang and C [9] that uses domain general abstractions and a method JIT, and the RSqueak/VM written in object-oriented RPython that uses domain specific abstractions and a tracing JIT.

One use-case for virtual machine developers is browsing the generated JIT code. Squeak already allows switching between showing a method's source and showing the generated bytecode. We have added the bytecode virtual nodes and added a virtual node with the generated assembler from the JIT (if available for the method). For the Cog VM, the VM sources provide a simulation package which we use to generate annotated assembler specifically for a given method. Figure 9 shows the entire code that was required to implement this. For the RSqueak/VM, we use RPython reflective API to ask for annotated RPython intermediate representation if it is available. Experienced developers can compare the generated code and look for inefficiencies or errors. However, as part of future work we are planning to provide a visual mapping from bytecodes to assembler and to integrate a simulation of the assembler execution so that intermediate results can be stepped in parallel.

An additional use-case that is more specific to Squeak is debugging into primitives. Most primitive behavior in Squeak is written in Slang (a subset of Smalltalk) that can be converted to C and compiled to create the primitive functions in the VM. Many primitives, however, also have so-called fallback code in the Squeak image. This is ordinary Smalltalk code without limitations. The intention of this code is to run when a given primitive was not compiled in the executing VM, or when some cases were too difficult to express in the Slang subset. This means that there may be two completely separate implementations of the same primitive behavior (in Slang and in Smalltalk). Furthermore, the C code derived from the Slang code can be prone to bugs from the automatic type inference and compilation. And as stated above, the JIT may have bugs or inefficiencies when compiling the Smalltalk fallback code[4].

[4]For example, a bug was discovered in the JIT for integer division that does not occur when running without JIT. https://github.com/OpenSmalltalk/opensmalltalk-vm/issues/6#issue-161101277, accessed June 29, 2016

```
children: activation
| children |
activation basicHasChildren ifTrue: [^activation basicChildren ].
children := OrderedCollection new.

activation isSmalltalkContext ifFalse: [^children].

children add: (VirtualContextNode new
    definitionContent: ([:methodOrDoitString :optionsDictionaryOrArray|| tuple stream |
        tuple := StackToRegisterMappingCogit
            cog: methodOrDoitString
            selectorOrNumCopied: methodOrDoitString selector
            options: optionsDictionaryOrArray.
        stream := ReadWriteStream on: ''.
        tuple second disassembleMethod: tuple last on: stream.
        stream contents asString withBlanksTrimmed ]
            value: activation contextPart method value: #(ObjectMemory Spur32BitCoMemoryManager))
    group: self;
    root: activation;
    at: #viewer put: #cogit;
    at: #leftDefinitionContent put: (activation contextPart method symbolic asText);
    summary: (activation contextPart selector),'.bytes.cogit';
    yourself).
```

Figure 9: Adding low-level virtual nodes for JIT compiled code. For each Smalltalk activation frame, we return one `VirtualContextNode` child with its content set to the annotated JIT code we generated using the JIT simulation class `StackToRegisterMappingCogit`.

Our prototype allows developers to decide which code to use and compare the results. The fallback and Slang code can be executed through the debugger, the C code corresponds to just running the primitive in the VM. The JIT code can be simulated at the moment by running through the fallback code a few times to make sure it is jitted in the VM. This is brittle, however, and in future work we are planning to run the C code by compiling a shared library and running it through FFI and to run the JIT code by sending it to a CPU simulator like Bochs. This will also allow live development of these representations to find correct forms and then work backwards from the desired C or assembler to change the Slang-to-C compiler or the JIT. Similarly, the execution control of external call stacks is desirable for further understanding. For stepping through primitives, we would need to be able to step through the VM the debugger itself is running in or use a VM-level implementation of that primitive to simulate it.

4. DISCUSSION

The presented interaction model extends the existing interaction with symbolic debuggers, is uniformly applicable to different perspectives, and is feasible to implement.

The interaction model itself extends the idea of symbolic debugging by introducing optional, extended context. Large parts of the interaction remain the same as with standard symbolic debugging. The conventional model of a plain stack list is preserved as the default debugging perspective. From the point of view of our interaction models, this corresponds to the case where no stack-frames have been filtered and no virtual stack tree nodes are visible. As the new perspective selector view can be provided for example as context menu, our extended capabilities can disappear completely if necessary. This fits our requirement for an unobtrusive extension to the conventional symbolic debugger. When advanced perspectives are desired or stack-frames should be hidden, our interaction model can provide this functionality in a uniform way, providing the multi-level debugging capability described earlier. Since the interaction model only depends on the idea of debugging execution using a list of stack-frames as its data source, we consider it generally applicable to other symbolic debuggers that also use stack-frames.

We have provided two walk-throughs with our prototype that illustrate the work-flow and the uniform user interface for different kinds of abstractions. In particular, the walk-throughs highlight the possibility of debugging abstraction client code while *selectively* showing or hiding abstraction host code and vice versa. Furthermore, they illustrate how to inspect different transformation artifacts of the same source. However, the walk-throughs also revealed aspects of the interaction model that are dependent to the specific debugging use case. For example, nodes of a Squeak/Smalltalk primitive (its simulation, JIT assembly, C-generating code and the generated C code) did not exhibit inherent levels of abstraction. Establishing a tree order for them required tuning on a per-use-case basis. Nonetheless, the debugger can be extended to change this behavior and adapt it to a particular use-case. For example when debugging the JIT compiler, the simulation code may be displayed first to remember a primitive's intended semantics, followed by the JIT implementation node to reason about it, and finally the Slang implementation to compare with another implementation. While the application to primitive implementation may not be applicable to a lot of other stack-based languages, similar abstractions can be found in other stack-based interpreters and virtual machines and this concept can be applied likewise. The application of our interaction model to non-stack-based computation models remains future work, however. Similarly, its use for programming concepts such as AOP or COP requires further investigation.

The interaction model is feasible to be implemented as demonstrated with our prototype in Squeak/Smalltalk. It implements the described interaction model using standard UI elements including the level-implementations for the described walk-throughs. While the prototype does not share the actual user interface of the Squeak debugger, its standard interactions remain nevertheless. The perspective selector is visible by default, but implementations that only show it on demand are certainly possible. Our prototype can be hence described as an extended Squeak debugger. The implementation is open to extensions itself. A new *level* may be added to the system by sub-classing from *DebuggerLevel* so that libraries and

frameworks could potentially bring their own levels. Library developers could hence build their knowledge about their library's kind of abstractions into the debugger and ship library specific extensions to the debugger together with their library itself. More data points and experience with this extension API are however necessary to report about its feasibility and limitations. Nevertheless, our implementation suggests that a similar architecture can be chosen in other languages. We see no reasons that would inhibit the extension of other stack-based symbolic debuggers.

5. RELATED WORK

TIDE [11] helps building domain specific languages and a debugger for them. It describes a language-generic API for extending the debugger that could be used in our system. While TIDE is built for building and debugging DSLs, our system additionally allows the inspection of arbitrary kinds of abstractions, such as libraries or transformation artifacts (C, JIT-Code).

The Moldable Debugger [3] introduces a new workflow for multi-level debugging and motivates its need with various use cases. It offers an new user interface for debugging that requires developers to learn and get used to it. Our system can unobtrusively be built into modern times' IDEs without learning needed until users need further details. We assume that unfolding the abstraction levels to see node-related sub-nodes is quickly discoverable.

The Maxine VM inspector [8] facilitates the debugging of stack-frames of different nature, notably stack-frames from baseline-JIT-compiled code, optimizing-JIT-compiled code, and (native) C. The focus for the inspector is to foster the understanding of the different implementation levels of the VM. It does not intend to support application developers or library developers. Also, due to the nature of the Maxine VM, the inspector is a standalone tool that does not integrate with any existing debuggers. Moreover, the inspector is not intended as a full-fledged debugger; while it supports stepping through code, changes to data and behavior are not supported.

6. FUTURE WORK AND CONCLUSIONS

When debugging a language runtime, it is often required to have dedicated tools depending on used technology, abstractions and system knowledge. The system we have presented builds on top of a general purpose Smalltalk debugger instead. It introduces extension points to allow runtime and language implementers to add perspectives for the abstraction and translation levels of the system into one shared place.

There are several debugging tools that introduce a new debugging language [2, 4, 7, 10]. Others constrain the development, e. g. by requiring the code to be UML-generated so that the debugger can map runtime and diagram [5]. Some of those systems also describe the need for higher-level abstractions on top of the implemented program. Nonetheless, they still require application developers to formulate such mapping themselves. With our system, the debugging perspective may be automatically generated or shipped with a library – although the cost of possibly being less specialized on the developers actual design model. For debugging complex systems, the effort required to provide specific mappings for such systems may pay back (such as our specific mappings for gathering assembler and C code for Squeak primitives). However, for developers of many smaller code bases we think that even the basic heuristics that our system brings as default can ease their process without learning nor programming the debugger themselves.

While our previous work [6] focused on the concept of building a recursive-evaluating interpreter and how to debug a guest language together with its implementation, we now show that the basic model can be extended to contain details on levels below the inspected VM. Previously described virtual nodes represented higher-level behavior and show the Smalltalk stack-frames as implementation details. Now, virtual stack nodes may also represent behavior on a lower level of abstraction and with that closer to the machine's actual execution model.

There are several directions for future work. We are planning to extend the prototype to allow more fine-grained stepping through and inspecting of low-level frames. In particular, intermediate results may be useful to compare execution behavior of different levels of abstraction for e. g. primitive behavior. Furthermore, we want to allow developers to choose which of the functionally equivalent transformation results should be executed when running a program. When executing all copies for all primitives, the debugger could automatically compare their results and stop when they differ. We believe this could be a powerful tool to develop and experimentally verify transformations.

Our prototype is already useful for understanding the connection between various transformations and to investigate bugs that may occur when these transformations are incorrect.

7. REFERENCES

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., USA, 1985.

[2] P. C. Bates and J. C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software*, 3(4):255–264, 1983.

[3] A. Chiş, T. Gîrba, and O. Nierstrasz. The moldable debugger: A framework for developing domain-specific debuggers. In *International Conference on Software Language Engineering*, pages 102–121. Springer, 2014.

[4] M. Ducassé. Coca: An automated debugger for c. In *Proceedings of the 21st international conference on Software engineering*, pages 504–513. ACM, 1999.

[5] L. Geiger and A. Zündorf. Graph based debugging with fujaba. *Electr. Notes Theor. Comput. Sci.*, 72(2):112, 2002.

[6] B. Kruck, S. Lehmann, C. Kessler, J. Reschke, T. Felgentreff, J. Lincke, and R. Hirschfeld. Multi-level debugging for interpreter developers. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 91–93, New York, NY, USA, 2016. ACM.

[7] D. Liang and K. Xu. Debugging object-oriented programs with behavior views. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 133–142. ACM, 2005.

[8] B. Mathiske. The maxine virtual machine and inspector. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 739–740. ACM, 2008.

[9] E. Miranda. The cog smalltalk virtual machine. In *VMIL'11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, 2011.

[10] R. A. Olsson, R. H. Crawford, and W. W. Ho. A dataflow approach to event-based debugging. *Software: Practice and Experience*, 21(2):209–229, 1991.

[11] M. Van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju. Tide: A generic debugging framework—tool demonstration—. *Electronic Notes in Theoretical Computer Science*, 141(4):161–165, 2005.