MASTER IN COMPUTER SCIENCE

# Improving live debugging of concurrent threads

## Master Thesis

Max Leske

University of Bern

11th August 2016

Prof. Dr. Oscar Nierstrasz
MSc. Andrei Chiş

Software Composition Group, Institute of Computer Science
University of Bern, Switzerland

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

# Abstract

Concurrency issues are inherently harder to identify and fix than issues in sequential programs. Debuggers for concurrent programs therefore need to supply not only information pertaining to concurrency specific issues but also all of the information that sequential debuggers offer. Specialised debuggers for concurrent programs can provide this information but they usually operate on traces and are not live. Live debuggers enable a more immediate way of analysing problems and are used by many programming environments for many different languages, such that most of the developers today are accustomed to using them.

Contemporary live debuggers for concurrent programs usually are simply sequential debuggers with the ability to display different threads in isolation. To these debuggers every thread is a sequential program. Unfortunately, thread call stacks always begin with a designated start routine and the calls that led to the creation of the thread are not visible, as they are part of a different thread. Single threaded, sequential programs have a call stack that reaches back to the start of the application, threads do not. Thus, despite threads being treated as sequential programs, live debuggers do not display the same amount of information for threads as for single threaded, sequential programs.

In this work we propose to augment the call stacks of threads in the debugger, such that live debuggers can display the complete call stacks. The information shown for threads will then be the same as for single threaded, sequential programs. We give an overview of the concurrency related features in current live debuggers, provide examplary implementations of our idea for different use cases and finally describe the downsides of our approach and evaluate possible solutions for them.

# Acknowledgements

I want to thank my supervisor Andrei Chiş for always having time for me, even when he didn't — Prof. Oscar Nierstrasz for his patience and for assembling so many great people in one place — Eliane Rupp for her love and friendship — my parents for supporting everything I do — Tjade Höhre for our outdoor adventures — Philippe Wechsler and Thomas Siegrist for putting up with my technical gibberish — Tudor Gîrba, Clément Bera, Eliot Miranda and Denis Kudriashov for their inspiration and technical feedback — Claudio Corrodi for his eye for misplaced commas — Christoph Wysseier for being such a great boss — and all of the people in my life simply for being there.

# Contents

# 1 Introduction

According to Pennington [26], developers build a mental model of a program in terms of control flow and data flow. Live debuggers support developers in building that mental model by providing access to concrete values of variables and real-time views of the effects of expressions and statements. For control and sequence bugs [2] in particular, such as erroneous conditional expressions or invalid state transitions, the ability to navigate the call stack helps to identify and fix issues.

A developer who is intimately familiar with a program may possess an understanding of the program sufficient to find and fix a bug without needing the complete call stack. The larger a project, however, the less likely it is for a developer to possess that knowledge. Even developers with a good mental model of the program may need the call stack when they find that their model is wrong or incomplete.

## 1.1 Disrupted call stacks

A single thread is in itself a sequential program that can create new threads, which are executed concurrently. We call the original thread *master* and any thread created by the master *slave*. Slaves can themselves be masters of other threads, thus forming a hierarchy rooted in the first thread of the operating system's launch process, which has no master. We use the term *history* to describe the complete call stack of a thread including the stack frames from all its master threads.

The POSIX standard [16] defines that a thread is created in such a way that the first activation record executes a start routine[1], which the caller must specify, and that the thread is terminated immediately after returning from the start routine (a thread may also terminate earlier explicitly). Logically, the stack frames of the master thread preceding the start routine are part of the slave's call stack as well. However, as they are older than the start routine they can never be reached by the slave thread. Hence, thread implementations do not make these older stack frames available to the slave thread in an attempt to reduce the amount of memory consumed by stack frames.

[1] `http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_create.html`

The consequence is that a live debugger operating on such an implementation can only show the frames of a slave thread up to the point of the thread's start routine. One might argue that a developer could simply switch to the master to look at the older stack frames but that is only true when:

- the master has not yet exited and
- the master is waiting for the completion of the slave in the same frame in which the slave was created.

The second point is important because, while some of the stack frames that are part of the slave's history may still be present in the master, stack frames from which the master has returned are no longer accessible. For illustration, consider the following pseudo code example:

```
1  createThread
2    thread := Thread new.
3    thread
4      start: #runThread
5      in: self
```

Listing 1-1: The method `#createThread` creates and starts a new thread.

```
6  run
7    self createThread.
8    thread join.
9    ↑ self readThreadResult
```

Listing 1-2: The method `#run` asks for a new thread to be created, waits for it to exit and returns the result from the computation performed in the thread.

The method `#run` sends `#createThread`, waits for the slave thread to exit and returns the result that the thread has written to a shared variable. The slave thread is initiated in the `#start:in:` method and executes the `#runThread` method. The frame for the method `#createThread` will no longer be on the stack at the point where the master waits for the slave to exit (line 8), since the master will have returned from the method, as shown in Figure 1.1.

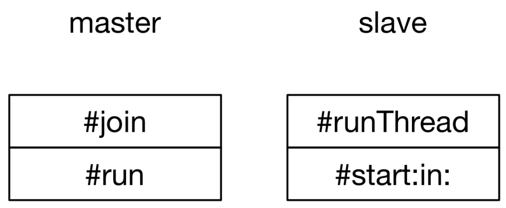


Figure 1.1: Stacks of master and slave threads at the moment where the master is waiting for the slave to exit (line 8) and the slave is executing the method `#runThread`.

The call stack of a thread's master is in general not available to a live debugger. Considering that a thread is also a sequential program we must conclude that live debuggers display less information for concurrent than for sequential programs. Live debuggers should display at least the same amount of information for concurrent programs, especially considering that concurrent programs are inherently harder to debug than sequential ones.

## 1.2 Augmented call stacks

The history missing from a slave thread can be retained by creating a copy of the master's call stack at the point where the slave is being created. A live debugger can later use that copy to augment the call stack of a slave thread with the call stack of its master. Figure 1.2 shows how the stack would look like in a debugger in case of the example from Section 1.1. The augmented call stack would contain not only the frames that are still active in the master and the slave, but also those from which the master has already exited, such as the frame of `#createThread`.

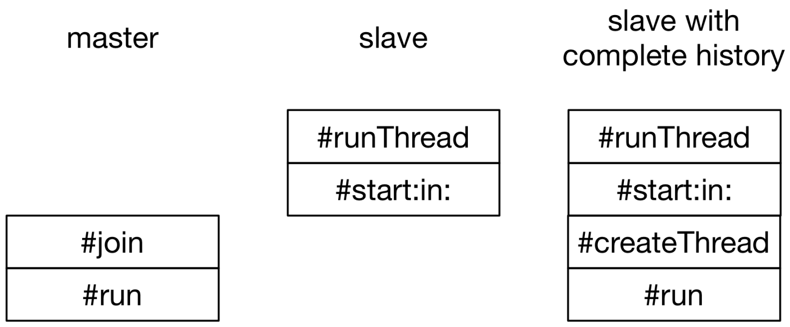


Figure 1.2: The call stack on the right shows the call stack of the slave as it would appear in a live debugger if the slave and its master were executed as a single sequential program. Note that the frame of `#createThread` is accessible in neither the master nor the slave.

The idea of augmenting stack frames can also be applied in other contexts where the call stack is disrupted. Examples include remote execution, promises, asynchronous events and asynchronous messages (*e.g.,* messages sent between actors in the actor model [14]). In a remote execution scenario, master and slave threads exist in separate environments and do not physically share a common call stack. Logically, however, the call stack of the master is part of the history of the remote slave thread. Promises and both asynchronous events and messages share the problem that from the point where they are usually shown in the debugger it is impossible to navigate to the point of their origin. In other terms, the debugger does not display the complete history.

To improve the state of the art in debugging concurrent programs, in this thesis we:

- present a comparison of live debuggers based on how they support the debugging of concurrent programs;
- provide exemplary implementations for creating augmented call stacks for threads, promises and remote execution;
- describe the downsides of our approach, their implications and possible solutions for overcoming them.

# 2 State of the art

In this chapter we explain essential terms used throughout the thesis and examine the current state of debuggers for different languages. Our analysis focuses on the shortcomings of current live debuggers in enabling debugging of concurrent programs.

## 2.1 Terminology

### 2.1.1 Notation

We will use Smalltalk conventions to talk about code. For example, `#copyFrom:to:` designates the name and signature of a method accepting two arguments; colons specify the positions of the method arguments. In case the receiver of a message is not obvious from the context we use `SequenceableCollection>>#copyFrom:to:` to say that the method `#copyFrom:to:` is implemented in the class `SequenceableCollection`. The following is an example of the message `#copyFrom:to:` being sent to an `OrderedCollection`, which is a subclass of `SequenceableCollection`:

```
fruits := OrderedCollection new
  add: 'banana';
  add: 'lemon';
  add: 'orange';
  yourself.
citrusFruits := fruits
  copyFrom: 2
  to: fruits size
```

Listing 2-1: All of the messages in this example are being sent to an instance of `OrderedCollection`. `#yourself` and `#size` are parameterless messages while `#add:` and `#copyFrom:to:` accept one and two arguments, respectively.

### 2.1.2 The call stack

An *activation record* represents the execution of a function or method. It includes all the information necessary to execute the function body and return to the caller. The necessary information is usually comprised of the function arguments, the stack pointer and the frame pointer. The exact layout of an activation record varies across languages and implementations. For example, some implementations push function arguments in *right-to-left* order onto the stack, others in *left-to-right* order. Activation records are linked together, forming a list (usually singly linked) called *call stack*. We use the term *stack frame* in reference to the call stack to mean the physical representation of an activation record.

In Smalltalk-80 the representation of activation records are instances of the `MethodContext` class [13]. Contexts store the program counter (next instruction of the method to be executed), the stack pointer (every context has its own isolated stack), the frame pointer (a reference to the previous context) and the method arguments. The original virtual machines for Smalltalk-80 instantiated an instance of `MethodContext` for every activation record. Newer virtual machines for Smalltalk derivatives use a more efficient scheme and only create context instances when necessary (*e.g.,* when the user wants to inspect a context) [10]. Unless we are talking about instances of `MethodContext` we will use the terms activation record and stack frame to refer to contexts in Smalltalk. In Pharo [5], which is the target language of our implemenations presented in Chapter 4, `MethodContext` has been renamed to `Context`.

### 2.1.3 Processes and threads

The call stack represents the currently active methods or functions within a *thread*, defined by POSIX as *"A single flow of control within a process"*[1]. POSIX further defines *process* as *"An address space with one or more threads executing within that address space, and the required system resources for those threads"*[2]. *Green threads* are virtual threads that are located and scheduled in user space by a virtual machine [29]. They may be executed within a single POSIX thread or mapped to multiple POSIX threads, depending on the capabilities of the virtual machine.

The class `Process` in Smalltalk-80 represents green threads, not processes as its name suggests. We will refer to instances of `Process` as threads to avoid confusion with the POSIX definition of process.

**Process and thread relationships** A new process is created by an existing thread and contains a single thread itself, which is created implicitly. A new thread is also created by an existing thread but will be executed within the same process as its creator. We use the terms *slave* to refer to a newly created thread and *master* to refer to the thread that created it. Slave threads can themselves be masters of other threads, thus forming a thread hierarchy. The processes of these threads consequently also form a hierarchy of which the operating system's launch process is the root. The launch process is the only process not created by another thread.

### 2.1.4 Promises

The concept of *promises* was introduced by Friedman *et al.* [12] for the Lisp programming language. Since then promises have been extended and implemented in a variety of ways so that today the definition depends on the language and the specific use case. Liskov and Shrira, for example, extend promises to have a static type and support for exceptions [20]. The terms

[1] `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_398`
[2] `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_291`

*future* (*e.g.,* in Java) and *task* (*e.g.,* in the .NET Framework) are sometimes used as synonyms of promise, other times both future and task are understood to describe different variants of the same concept. The fundamental idea, however, remains the same and it is that idea we use as the definition in this thesis: a promise is an eventual value, the computation of which may execute in a slave thread. A *remote promise* is a promise that is executed in a process that is not directly accessible by the master, *e.g.,* because it is executed on a different host.

### 2.1.5 Messages and events

Messages, such as defined by the actor model [14] and events, user interface events in the Document Object Model (DOM)[3] for example, are usually implemented to be delivered asynchronously (we will use the term *event* to describe both events and messages). Asynchrony can be achieved in a single threaded implementation, such as JavaScript, by using queues to store events. The event queues effectively decouple triggering and delivery.

Debugging of events most often occurs at the location where an event is received because it is easy for developers to anticipate that a given event will have a specific effect, regardless of the time at which the event will be received. Because of their asynchronous nature, however, the origin of events cannot be determined from that context, *i.e.,* the stack frame in which an event was triggered is not part of the current call stack. Events have this property in common with promises and although we will not discuss events specifically any further, the implementation of our idea for promises is easily transferrable to events. This will become clear when we look at the Scala asynchronous debugger and the Chrome development tools later in this work.

### 2.1.6 Debuggers for concurrent programs

The need for debuggers that show the relationships between processes has been recognised as early as 1986 [7, 31]. One possibility of giving developers access to the inter-process relationships are traces, which have been used for sequential debugging since 1969 [1]. Traces provide serialised records of the events that occurred during the execution of a program. They may be recorded as human readable text so that they can be browsed and processed by command line tools. Because of the high density of information in traces, specialised tools are often used. Trace debuggers simplify search and filtering of traces. Visual trace debuggers use the information from traces to present visualisations to the user that attempt to highlight specific properties of a trace. Specialised (visual) trace debuggers can also display dependencies between processes and threads (Utter and Pancake [31] give an excellent, albeit outdated, overview of parallel visual debuggers). Trace debugging is a powerful way of analysing issues in a program, especially related to concurrency. However, traces are usually costly to process, due to the large amount of data, and are therefore better suited for postmortem debugging.

Live debuggers offer a faster turnaround and more direct interaction with the program than trace debuggers. Some live debuggers even support manipulation of the program on the fly (also known as “fix-and-continue debugging”) so that the program does not need to be restarted or recompiled, such as in Lisp [19], and Smalltalk [13], where this functionality is built into the language enviroment. Other language environments have also begun adopting fix-and-continue debugging, such as Java as of JDK 1.4[4]. Live debuggers usually have some capabilities for dealing with concurrency (see Section 2.2).

[3]`http://www.w3.org/TR/uievents/#sync-async`

[4]`https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`

### 2.1.7 Remote debuggers

Remote debuggers exist for many languages and platforms. They enable debugging of a system which itself has no or only limited debugging facilities. Embedded systems for example have neither the power nor the output facilities, like display drivers, to make local debugging possible. Generally, two approaches exist to implementing remote debuggers. Query based remote debuggers issue queries and commands to their target to manipulate state and retrieve information. The target in this case is often a counterpart of the debugger installed on the remote system. One example of a query based remote debugger is the GNU debugger (GDB), which can be configured to communicate with remote targets via serial or network interface [28]. The other approach to implementing remote debuggers is to use some form of remote proxies that connect remote objects with local ones transparently. Remote debuggers in Smalltalk and its derivatives follow this idea [3, 9, 17, 24].

Remote debuggers can be used to debug remote execution, *i.e.,* execution of a task in a different environment (often a different machine). The call stack of the master thread is part of the history of the remotely executing slave thread, just as for threads that are executed in the same environment. The problem of accessing the complete history of a slave thread also applies to remote slave threads, especially in the case of asynchronous remote execution, of which remote promises are an example.

## 2.2 Current state of live debuggers

The space of programming languages today is too large[5] to produce a survey of all the debuggers of these languages in the context of this thesis. Hence, we had to generate a representative sample of the space of all debuggers. Since debuggers are tied to a single language or only few different ones, we looked at different programming languages to obtain different debuggers.

### 2.2.1 Popular languages

We started by examining debuggers of the most popular programming languages, as these are likely to be supported by financially strong companies that invest in tool innovation. The TIOBE Index[6] provides a continuously updated list of the most popular programming languages based on search engine results. The top ten of the index have been composed of the same languages since 2011 (with the exception of Visual Basic .NET), which indicates that these languages are not only popular but important to the industry. In June 2016 the top ten languages, in descending order, were: Java, C, C++, Python, C$^{\#}$ , PHP, JavaScript, Perl, Visual Basic .NET and Ruby. To round out the picture we add Objective-C and Swift for Apple's iOS platform since mobile platforms belong to the most important and fastest evolving platforms (the languages used for Google's Android and Microsoft's Windows Phone platforms are already included in the top ten).

### 2.2.2 Research languages

Innovation in programming languages and tools for those languages is also driven by research at universities. The languages used in software research are often not popular enough to appear in the top 10 of the TIOBE index. To account for languages used in research, we added Haskell

[5]Wikipedia (`https://en.wikipedia.org/wiki/List_of_programming_languages`) for example listed 694 programming languages on July 8th 2016

[6]`http://www.tiobe.com/tiobe_index`

(University of Glasgow [15]), Scheme (MIT [30]), Scala (EPFL[7]), Self (Stanford University[8]), OCaml (INRIA[9]), Prolog (Universitiy of Edinburgh, Université d'Aix Marseilles [18]) and Pharo (University of Bern[10], INRIA[11]) to the sample.

### 2.2.3 Other languages

Many other languages have their own debuggers. We argue, however, that the debuggers for the languages in our sample represent the majority of innovations. It is therefore unlikely that debuggers of other languages include features that are not already present in our sample but are of interest in the context of this work.

### 2.2.4 Selected debuggers

From the selected languages we derived a list of debuggers by including the standard debugger for each language (if one exists) and possibly others that are used by a significant number of developers (*e.g.,* debuggers used by popular development environments). The debuggers covered by our survey are: Java Debugger Interface[12] (Java), Visual Studio debugger[13] (C# , C++, Visual Basic .NET, JavaScript), PyDev[14] (Python), pdb[15] (Python), perldebug[16] (Perl), GDB[17] (C), Chrome development tools[18] (JavaScript), XDebug[19] (PHP), Zend Debugger[20] (PHP), debug.rb[21] (Ruby), LLDB[22] (Objective-C, Swift), GHCi debugger (Haskell)[23], Concurrent Haskell debugger [6] (Haskell), SISC debugger[24] (Scheme), Scala asynchronous debugger[25] (Scala), Self debugger[26] (Self), ocamldebug[27] (OCaml), XPCE debugger[28] (Prolog), Pharo debugger[29].

### 2.2.5 Debugger features

We identified the features each of the debuggers provides with respect to threads, events, messages (actor model) and promises. For every feature we derived a question that we could ask for a given debugger to find out whether it supports that feature. The following list shows the questions we asked, the answers to them are depicted in Table 2.1:

[7] `http://www.artima.com/weblogs/viewpost.jsp?thread=163733`
[8] `http://www.selflanguage.org`
[9] `http://ocaml.org/learn/history.html`
[10] `http://scg.unibe.ch/research`
[11] `http://rmod.inria.fr/web/research`
[12] `http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/`
[13] `https://msdn.microsoft.com/en-us/library/sc65sadd.aspx`
[14] `https://github.com/fabioz/PyDev.Debugger`
[15] `https://docs.python.org/2/library/pdb.html`
[16] `http://perldoc.perl.org/perldebug.html`
[17] `https://www.gnu.org/software/gdb/`
[18] `https://developer.chrome.com/devtools`
[19] `https://xdebug.org`
[20] `http://files.zend.com/help/Zend-Studio/content/remotely_debugging_a_php_script.htm`
[21] `http://ruby-doc.org/stdlib-2.0.0/libdoc/debug/rdoc/DEBUGGER__.html`
[22] `http://lldb.llvm.org`
[23] `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html#the-ghci-debugger`
[24] `http://www.sisc-scheme.org/manual/html/ch04.html`
[25] `http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html`
[26] `http://handbook.selflanguage.org`
[27] `http://caml.inria.fr/pub/docs/manual-ocaml/debugger.html`
[28] `http://www.swi-prolog.org/pldoc/man?section=guitracer`
[29] `http://pharo.org/documentation`

- **thread hierarchy information:** does the debugger provide some kind of information (visual or textual) about the master-slave relationship of threads?
- **thread switching:** does the debugger allow developers to change the thread currently in focus?
- **thread isolated breakpoints:** can breakpoints be isolated to specific threads, *e.g.,* through conditional breakpoints?
- **event history:** can the debugger show stack frames from the point of creation of events or messages?
- **promise history:** can the debugger show the history of promises across different threads?
- **thread history:** can the debugger display the original call stack of the master thread for a given slave?

### 2.2.6 Notes on debugger features

**Thread hierarchy information** The only debugger that provides any kind of information about the hierarchichal relationship between master and slave threads is the Visual Studio debugger (for so called "managed code"). Even though presenting information about the thread hierarchy intuitively seems like a good idea, it does not appear to be something that developers demand or that many companies think can be sold as a feature of their development environment. The fact that only the Visual Studio debugger provides this feature becomes less surprising when taking into account that Visual Studio until recently targeted the Microsoft Windows operating system exclusively and that Windows *"relies more heavily on threads than processes"*[30]. Thus, the incentive to build a thread hierarchy view may have been greater for the Visual Studio debugger than for debuggers that target other platforms (or are not exclusive to Windows).

**Thread switching** In this case it is interesting to note that Perl and Python implement threading but do not provide adequate debugger support. It is even more interesting that PyDev allows developers to switch threads even though it is not the standard debugger for Python. It would be interesting to find out why debugger support for such a fundamental feature is not being provided by the standard tools.

The debuggers for Haskell, Scheme and OCaml provide only implicit thread switching because they are functional languages in which threads are also implemented with functions. In these implementations only the active thread can be displayed in the debugger and switching to a different thread requires a context switch. Scala, which is also partly functional, by contrast uses the thread implementation of the HotSpot virtual machine and therefore its debugger can provide explicit thread switching. Of the debuggers that we looked at for Haskell, Scheme and OCaml, only the GHCi debugger for Haskell actually supports breaking on different threads in the same session.

**Thread isolated breakpoints** In all of the languages we looked at, threads are exposed as first class objects and allow access to their internal identification ("thread-ID"). Many of the debuggers satisfy the property of isolated thread breakpoints simply because they offer conditional breakpoints where the condition can include a comparison with the current thread's thread-ID.

[30] `https://technet.microsoft.com/en-us/library/bb496993.aspx`

| | thread hierarchy information | thread switching | thread isolated breakpoints | event history | promise history | thread history |
|---|---|---|---|---|---|---|
| Java Debugger Interface (JNI) | · | ✓ | ✓ | · | · | · |
| Visual Studio debugger | ✓ | ✓ | ✓ | · | · | · |
| PyDev | · | ✓ | ✓ | · | · | · |
| pdb | · | · | ✓ | · | · | · |
| perldebug | · | · | ✓ | · | · | · |
| GDB | · | ✓ | ✓ | · | · | · |
| Chrome development tools | n/a | n/a | n/a | ✓ | ✓ | n/a |
| XDebug | · | ✓ | ✓ | · | · | · |
| Zend Debugger | · | ✓ | ✓ | · | · | · |
| debug.rb | · | ✓ | ✓ | · | · | · |
| LLDB | · | ✓ | ✓ | · | · | · |
| Concurrent Haskell debugger | · | · | ✓ | · | · | · |
| GHCi debugger | · | ✓ | ✓ | · | · | · |
| SISC debugger | · | · | ✓ | · | · | · |
| Scala asynchronous debugger | · | ✓ | ✓ | ✓ | ✓ | · |
| ocamldebug | · | · | ✓ | · | · | · |
| Self debugger | · | · | ✓ | · | · | · |
| XPCE debugger | · | ✓ | ✓ | · | · | · |
| Pharo debugger | · | ✓ | ✓ | · | · | · |
| Pharo thread debugger | · | ✓ | ✓ | · | ✓ | ✓ |

Table 2.1: Thread related features of live debuggers

Not all debuggers provide conditional breakpoints out of the box however. The SISC debugger for Scheme for example does not include a facility for conditional breakpoints but can be extended with a custom function that provides it.

**Promise history** Only the Scala asynchronous debugger and the Chrome development tools can show stacks for promises across multiple threads. What this shows is that the concept is an important idiom in the respective languages, more important at least than in other languages.

**Event history** The same two debuggers that can show the promise history across different threads, namely the Scala asynchronous debugger and the Chrome development tools, can also show an augmented stack for events. This suggests that the implementation of promises and events could be similar, so that the implementation for either promises or events led to the implementation for the other structure naturally. For JavaScript this conclusion is in fact rather obvious since it is single threaded.

**Thread history** None of the debuggers we looked at implements our idea of augmenting the call stack of a thread with the call stack of its master.

### 2.2.7 Notes on debuggers

Most of these debuggers use the same approach to debugging concurrent processes and threads:

1. when a breakpoint is hit, suspend all running processes / threads;
2. present the user with lists of all current processes / threads;
3. let the the user choose the process / thread she is interested in;
4. present details of the selected process / thread to the user, isolated from other processes / threads.

**perldebug** perldebug includes an experimental thread debugging option that identifies the current thread and can list all running threads. It is not possible to switch between different threads in perldebug.

**Visual Studio debugger** The Microsoft Visual Studio debugger (C$^{\#}$ , C++, Visual Basic .NET) can visualise thread hierarchies (*i.e.,* the relationship between master and slave threads within a single process) and show stack frames shared among different active threads (*e.g.,* two threads created at the same point will share at least some frames). An additional “task view” provides the same information for tasks, a concept related to promises[31].

**Chrome development tools** JavaScript is single threaded but the event mechanism uses threads or a mechanism amounting to the same. For that reason only “promise history” is an applicable question for JavaScript debuggers.

The “Async” option in the Chrome development tools makes the call stack list display a contiguous stack for events, promises and asynchronous network requests (`XMLHttpRequest`)[32]. Traditionally, these constructs are displayed only with the stack of their current context *i.e.,* the

[31]`https://msdn.microsoft.com/en-us/library/hh873175(v=vs.110).aspx`
[32]`http://www.html5rocks.com/en/tutorials/developertools/async-call-stack/`

place where an event, promise or `XMLHttpRequest` was created cannot be determined from the visible stack. The idea is very similar to the one used in the Scala asynchronous debugger.

**Concurrent Haskell Debugger** An experimental debugger for Haskell that employs the same scheme as other debuggers but also visualises relationships between threads through lines between threads and shared resources.

**Debugging of futures in Java** Zhang *et al.*, for their *Directive-based lazy futures* implementation in Java, proposed a serialised view of the events of both the slave thread and its master [32]. In contrast to our approach, the serialised view is not created on demand but is a side-effect of their stack-splitting strategy: the virtual machine (a customised Jikes VM) creates a future by copying the current thread and marking the current activation record as the bottom of the stack. The slave process thus contains all the activation records from before its creation (the mark is reversible). Their implementation of futures is based on the idea that the virtual machine can decide to create threads for long running computations, hence a future is implicit and not guaranteed to run in a separate thread at all. It also means that their implementation does not support remote futures because the location of execution cannot be configured.

**Scala asynchronous debugger** The Scala asynchronous debugger is an extension to the Scala debugger that adds support for debugging of actors and futures. To that end the debugger creates a continuation at the point where a future is being created or a message is being sent to an actor. The continuation is later used in the debugger to present the context of the future or message to the developer as a call stack, separate from the thread's call stacks.

**Pharo debugger** With regard to threads, events, messages and promises the Pharo debugger does not distinguish itself from other debuggers as it only supports thread switching (although only implicitly) and isolated thread breakpoints (via named threads and conditional breakpoints). We mention this explicitly because we use the Pharo debugger as the basis for the implementations introduced in Chapter 4.

## 2.3 Summary

Live Debuggers across different languages are remarkably similar with respect to the features we looked at. This suggests that there is little innovation in this particular area or that at least very few innovations of this kind are being built into debuggers. The Chrome development tools, the Scala asynchronous debugger and the Visual Studio debugger are clearly the most innovative debuggers when it comes to debugging of asynchronous programs but none of them solves the general problem of lost thread history.

The thread hierarchy view in the Visual Studio debugger is a great idea that we would like to see in other debuggers. Unfortunately, the view can only show the hierarchy between threads that are live *i.e.,* the relationship between a slave and its exited master is not visible. By applying our idea of augmenting thread call stacks to the thread hierarchy view, a thread would always be visible with its complete hierarchy, regardless of the state of the master and thread execution order (the state of the master may differ depending on the order of execution). Users of the view could then also easily navigate to the point where a thread was created.

Events and promises have been hard to debug in the past partly because both have implicit relationships to different parts of the call stack, *e.g.,* point of creation and point of activation, that could only be inferred from other information. There was no way to tell the debugger to navigate to any such point directly. Both the Scala asynchronous debugger and the Chrome developer tools have solved this problem, for Scala and JavaScript respectively, by using the same idea we are proposing. The key difference however, is that we propose to apply the idea to threads in general, which solves the problem for promises and events as special cases.

# 3
# Augmenting thread call stacks

Threads, as defined by POSIX and implemented in the languages listed in Chapter 2, do not hold on to the stack frames that are part of their masters. This is largely due to threads not being created as copies of the current thread, in contrast to POSIX processes, but with an explicit start routine. Since a thread is meant to exit at the latest when it reaches the end of its start routine, no frame of the master that is conceptually part of the history of the slave would ever be executed. From the point of view of implementation and resource management it would be a waste to copy the frames of the master. Even the use of a copy-on-write [23] scheme would entail wasted space because the memory in which those stack frames lie from which the thread has returned cannot be reclaimed until the slave exits.

Despite these disadvantages the master thread needs to be copied in order to provide the information that allows us to build better debuggers for concurrent programs.

## 3.1 Constructing a virtual call stack

We propose to save the histories of threads so that they can be supplied to the debugger. For a given thread the debugger can then display the call stack as if the slave and its master were a single sequential program.

Figures 3.1 and 3.2 show the call stacks of a master thread and its slave. “A” marks the frame in the master in which the slave has been created. Figure 3.1 depicts the situation in which both master and slave have performed some method calls concurrently. While creating the slave, the master also created a copy of itself, which contains copies of all the stack frames that are part of the history of the slave, *i.e.,* all frames up to and including “A”.

When the slave thread is interrupted, *e.g.,* due to a breakpoint or unhandled exception, we can construct a virtual stack from the slave and the copy of its master as shown in Figure 3.2. The bottom part of the virtual stack references the stack frame copies from the master thread, while the top part is comprised of the references to the stack frames from the slave. The bottom stack frame of the slave thread immediately follows the frame “A”, in which the slave was created. A debugger operating on such a virtual stack now enables users to navigate the stack frames from two threads sequentially.

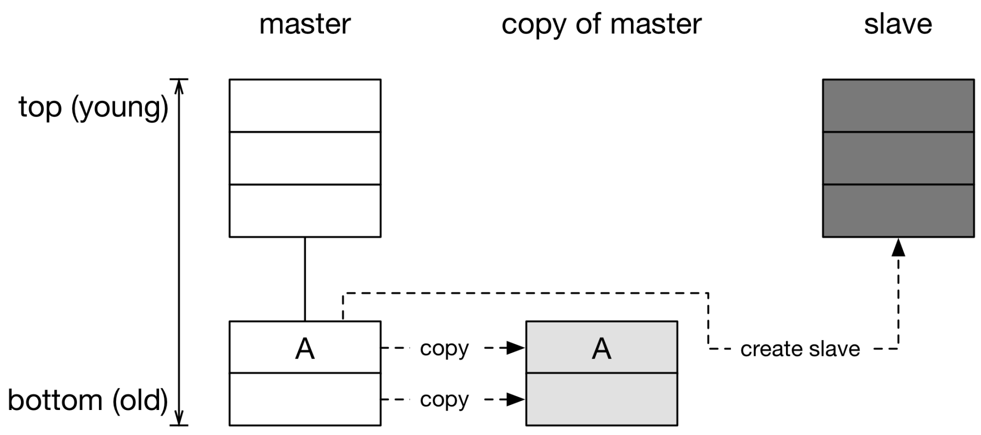


Figure 3.1: Stacks of two concurrent threads with a master-slave relationship.

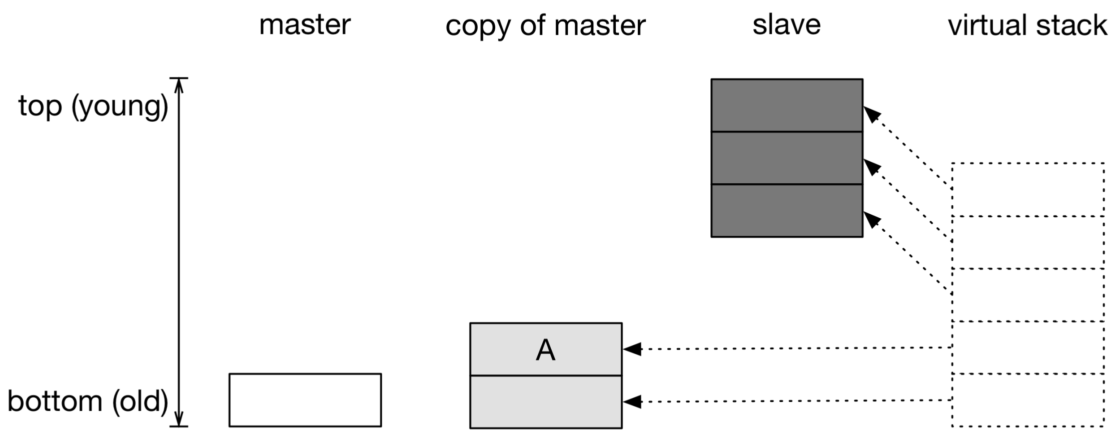


Figure 3.2: The virtual stack references a copy of a frame instead of the original if the master has already returned from it.

The virtual stack is a list of stack frames that does not depend on the links between frames, an array for example. This of course depends on the debugger being able to operate on a collection of stack frames instead of accessing the stack directly through the stack frame links. For debuggers that cannot operate on a virtual stack the concept remains the same but has to be achieved by manipulating links between stack frames, *i.e.,* by linking the bottom frame of the slave to the top frame of the master's copy.

### 3.1.1 Virtual call stack of multiple threads

Master and slave threads form a hierarchy in which every thread is a slave thread (with the exception of the launch process thread) and can itself be the master of other threads. If we indeed want to preserve the complete history of a thread we must therefore construct the virtual stack recursively such that the call stacks of all masters are represented in the virtual stack. For debuggers working with augmented threads this means they have to support dynamic thread switching and activation record selection, opposed to hard coding a master and a slave thread.

## 3.2 Thread states

When the debugger displays the virtual stack, the state of the master is undetermined. The following states are possible:

- running,
- suspended,
- exited,
- blocked (waiting on another thread, or for an interrupt),
- blocked while waiting on the slave that is currently being debugged.

For simplicity, we assume in our model that the copy of the master's frames can be inspected but cannot be acted upon. If we knew, however, that the master was blocked and waiting for the slave that is open in the debugger, we could use the active frames of the master in the virtual stack and the debugger would be able to operate on those stacks. In all other cases, letting the debugger perform actions on active frames of the master (or copies) could be harmful and in general lead to undefined behavior.

## 3.3 Interaction with the virtual stack

For debuggers, the interaction with a virtual stack is more complex than the interaction with a single thread. The actions developers can perform on stack frames or threads must all be implemented to take into account that more than one thread is part of the call stack.

**Set breakpoint** Breakpoints can be set in any stack frame. However, for frames that will never be reached by the execution, *i.e.,* frames that are not part of the slave and are no longer live, it may be better to ignore the action and inform the user that the breakpoint would have no effect.

**Step to next instruction** Stepping to the next instruction is always possible, except when crossing thread boundaries. It may be necessary in this case to perform additional actions to properly terminate the slave thread. This also applies to the related action “step to next instruction in current method”.

**Resume** Resumption is only possible when the top context of the virtual stack is part of the slave thread.

**Restart** For debuggers that support a restart action the action may be associated with a frame other than the top frame. For the frame that is associated with the action the debugger must check if the frame is part of the slave process before performing the action.

**Return value** Some debuggers support explicitly returning from a method with a value, which is only possible when the frame selected for this action is part of the slave process. Additional care may be necessary when crossing thread boundaries.

## 3.4 Summary

We have presented a simple model for live debuggers to display the history of a thread as a single, sequential virtual call stack. The key point of the model is that the master thread is copied at the point where a slave thread is being created, thus ensuring that no stack frames of the slave thread’s history are lost. Although it should be straightforward for most debuggers to display the virtual call stack, special care must be taken when a user requests to perform actions on the stack frame in focus because these actions depend on the state of the thread they are being performed on.

# 4 Implementation

We created prototype implementations of our approach in the Pharo Smalltalk environment. The main reason for choosing Pharo is that green threads have a first-class representation and it is thus possible to manipulate threads and their chains of activation records without support from the virtual machine. In addition, the debugger model in Pharo provides the means for extending existing debuggers and adding new ones with little effort.

See Appendix A for obtaining the reference implementations discussed in this chapter.

## 4.1 Augmented threads

The UML diagram in Figure 4.1 shows a simplified schematic of how the class `Process` in Pharo, whose instances represent green threads, is connected to its activation records. Activation records are represented by instances of class `Context`, which is the equivalent of the class `MethodContext` in Smalltalk-80, and are linked to each other through the *sender* field of `Context`. Every activation record is bound to the *method* whose activation it represents (`CompiledMethod`). Contexts also map their method to a *receiver*, which is the object bound to the *self* pseudo-variable in the activated method [13].

Also shown in Figure 4.1 are the new classes used for augmenting a thread with the history from its master. We introduce a subclass of `Process` called `UserProcess` that stores a copy of the active thread's call stack in an instance of `ProcessHolder`. This could have been implemented on `Process` but we wanted to guarantee a clean separation from the existing environment and prevent the creation of call stack copies for system threads, hence the name. `ProcessHolder` and `ContextHolder` are data structures that reduce the complexity of thread management, *e.g.,* by providing explicit mappings between an activation record, its copy and the executing thread.

The chain formed by the self reference on `ProcessHolder` is the manifestation of the thread hierarchy mentioned in Chapter 3. Through this chain it is possible to construct a virtual stack containing the complete history of a thread, *i.e.,* the virtual stack includes the activation records from all the master processes of the slave thread. The exception to this rule are instances of `Process` for which we do not want to create call stack copies, meaning that the chain of `ProcessHolder` instances terminates when the master is an instance of `Process`.

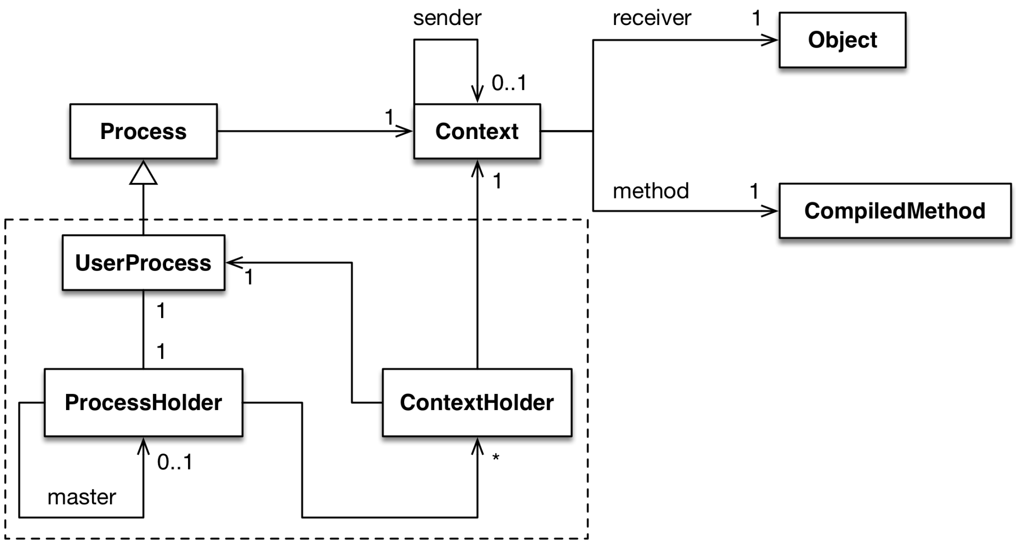


Figure 4.1: UML diagram of `Process` and its `Context` chain in Pharo.

## 4.2 The debugger model

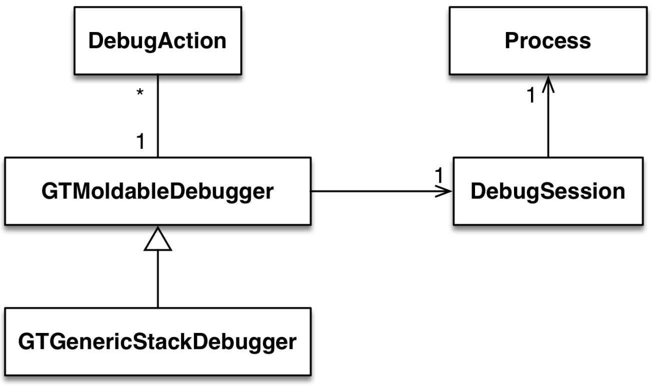


Figure 4.2: UML diagram showing the relevant classes of the moldable debugger infrastructure in Pharo.

Our thread call stack augmenting debugger uses the moldable debugger infrastructure [8] in Pharo, which allows us to implement a specialised debugger without the need to modify the default debugger of the system. Figure 4.2 provides an overview of the important classes provided by the moldable debugger infrastructure. The abstract class `GTMoldableDebugger` defines the visual model of a debugger while `DebugSession` represents the context of a debugger instance and handles interaction with the thread, an instance of `Process`, that is being debugged. Instances of `DebugAction` represent actions that can be performed by the debugger, such as stepping or evaluation of selected code. `GTGenericStackDebugger` is the the default Pharo debugger, which we use as the basis for our debugger implementation.

## 4.3 A live debugger for threads

In this section we introduce important details of the debugger implementation before looking at an example in which we use the thread debugger to analyse a bug.

### 4.3.1 Implementation details

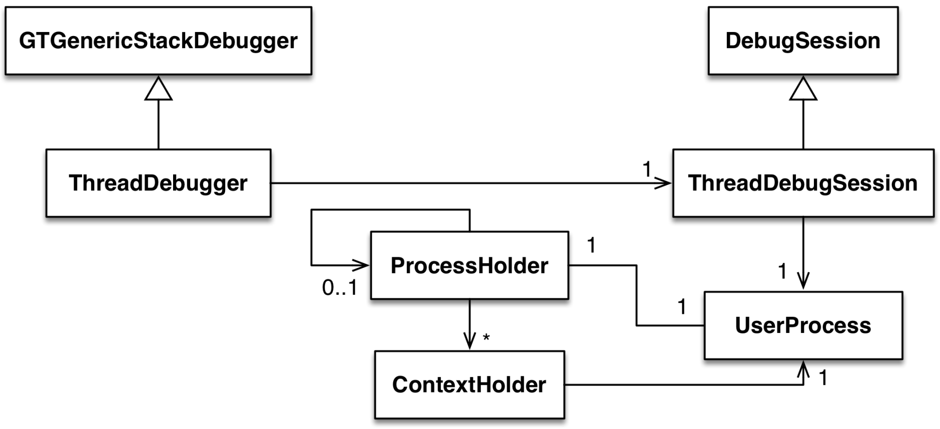


Figure 4.3: Our thread call stack augmenting debugger is implemented in the two classes `ThreadDebugger` and `ThreadDebugSession`.

For the implementation of our thread call stack augmenting debugger we introduced two new classes as shown in Figure 4.3. `ThreadDebugger` is a subclass of `GTGenericStackDebugger` that uses `ThreadDebugSession`, a subclass of `DebugSession`. Our custom debug session needs to control every operation on the stack to guarantee that activation records are correctly being mapped to the thread they belong to. `GTGenericStackDebugger` and `DebugSession` both assume that the debugger is operating on a single thread to which all of the activation records belong. In our case, however, we will be operating on two or more threads, and since a thread is responsible for executing instructions of a given activation record it is important to select the correct thread for execution. The classes `ProcessHolder` and `ContextHolder` make it simpler to check conditions and select the correct activation records and threads.

In the example shown in Listings 4-1 through 4-3, we create a new instance of `UserProcess`. We have configured our debugger to be the default debugger so that any unhandled exception, *e.g.,* a division by zero, will be opened in a `ThreadDebugger` instance.

```
process := [
  self
      divide: aNumerator
      by: aDivisor ] newUserProcess.
process
    priority: 30;
    resume
```

Listing 4-1: Example code showing how a new instance of `UserProcess` is being created by sending `#newUserProcess` to a closure (delimited by brackets). The new thread, an instance of `UserProcess`, is started by sending the message `#resume`.

```
25 newUserProcess
26     ↑ UserProcess
27       forContext:
28           [self value.
29           Processor terminateActive] asContext
30       priority: Processor activePriority
```

Listing 4-2: The method `BlockClosure>>#newUserProcess` creates instances of `UserProcess`.

```
31 initialize
32     super initialize.
33
34     masterProcessHolder := ProcessHolder for: Processor activeProcess
```

Listing 4-3: `UserProcess>>#initialize` is executed during creation of `UserProcess` instances and stores a reference to the active thread, which will become the master of the new thread.

The method `BlockClosure>>#newUserProcess` (Listing 4-2) creates a new unscheduled user process. The method `UserProcess>>#initialize` (which is sent upon creation of the instance) stores the copy of the current thread's call stack. The call stack copy is being created in `ProcessHolder>>#initializeWithProcess:` by the method `Context>>#copyStack` on line 38 (Listing 4-4). The pseudo variable `thisContext` refers to the activation record of the current method, which is also the topmost stack frame of the current process (the method `#stack` simply creates a collection from the linked list of activation records). In addition, `#initializeWithProcess:` stores a reference to the process and creates an instance of `ContextHolder` for every activation record (lines 39 to 48).

```
35 initializeWithProcess: aProcess
36     | stackCopy previousHolder |
37     process := aProcess.
38     stackCopy := thisContext copyStack stack.
39     contextHolders := thisContext stack withIndexCollect: [ :context :index |
40         ContextHolder
41             forProcess: aProcess
42             context: context
43             andCopy: (stackCopy at: index)
44             withIndex: index ].
45     previousHolder := nil.
46     contextHolders reverse do: [ :contextHolder |
47         contextHolder next: previousHolder.
48         previousHolder := contextHolder ]
```

Listing 4-4: `ProcessHolder>>#initializeWithProcess:` is executed during instance creation of `ProcessHolder` and creates a copy of the call stack of the thread passed as argument.

In the debugger we can make use of the stored call stack copy by appending it to the stack of the current process (lines 50 to 52, Listing 4-5). Note that `#filteredMasterStack` collects the activation records recursively for all master threads. The thread that is being debugged has been suspended. Its master, however, will in general continue to run. As mentioned in Chapter 3, we use a simple model and prevent all interactions with activation records from other threads so that it is unnecessary to implement logic for detecting thread state, corresponding actions and safety checks. The virtual stack we constructed is comprised of, from youngest to oldest, the live contexts of the slave thread followed by the copied activation records from its master threads, recursively. The debugger has to decide whether commands are allowed to be executed (live or copied context) and which thread to dispatch them to, depending on the selected stack frame and the action to be performed.

```
filteredCombinedStack
↑ self filteredSlaveStack
    addAll: self filteredMasterStack;
    yourself
```

Listing 4-5: The method `ThreadDebugSession>>#filteredCombinedStack` creates the virtual call stack for the debugger.

### 4.3.2 User interface

We demonstrate the use of the debugger user interface with an example, in which an attempt is made to perform static analysis of some code. Listings 4-6 through 4-8 show the example code for launching a static analysis task using a concurrent thread. The method `#runAnalysisOn:` must receive as parameter a relative or absolute path to an existing directory, such as either *"sourcecode"* or *"/private_repositories/sourcecode"*. The program performs the following steps:

1. Create an absolute path based on the *inputPath* parameter (line 55). When the input path is already absolute, *absolutePath* will have the same value as *inputPath*.
2. Initialize the analysis as a concurrently executed thread with the absolute path to the directory as input (line 59). The new thread will attempt to access the directory at the given path.
3. Inform the user that the analysis is executing (line 57).
4. The main thread returns from the method to wait for the next call.

```
53 runAnalysisOn: inputPath
54     | absolutePath |
55    absolutePath := self absolutePathFrom: inputPath.
56     self runAnalysisConcurrentlyOn: absolutePath.
57     self informUserToWait.
```

Listing 4-6: The method `StaticAnalysisService>>#runAnalysisOn:` is the entry point to the static analysis service.

```
58 runAnalysisConcurrentlyOn: absolutePath
59     [ self privateRunAnalysisOn: absolutePath ] newUserProcess resume
```

Listing 4-7: `StaticAnalysisService>>#runAnalysisConcurrentlyOn:` delegates the actual computation to a concurrent thread.

```
60     self runAnalysisOn: '/sourcecode'.
61     self runAnalysisOn: 'sourcecode'.
```

Listing 4-8: Possible ways to invoke `StaticAnalysisService>>#runAnalysisOn:`

When the concurrent thread attempts to access a directory that does not exist it will fail. The question that a developer debugging this problem must answer is, why did the directory not exist? Assuming that a directory exists at *"/private_repositories/sourcecode"* there are two locations for possible failure in the example:

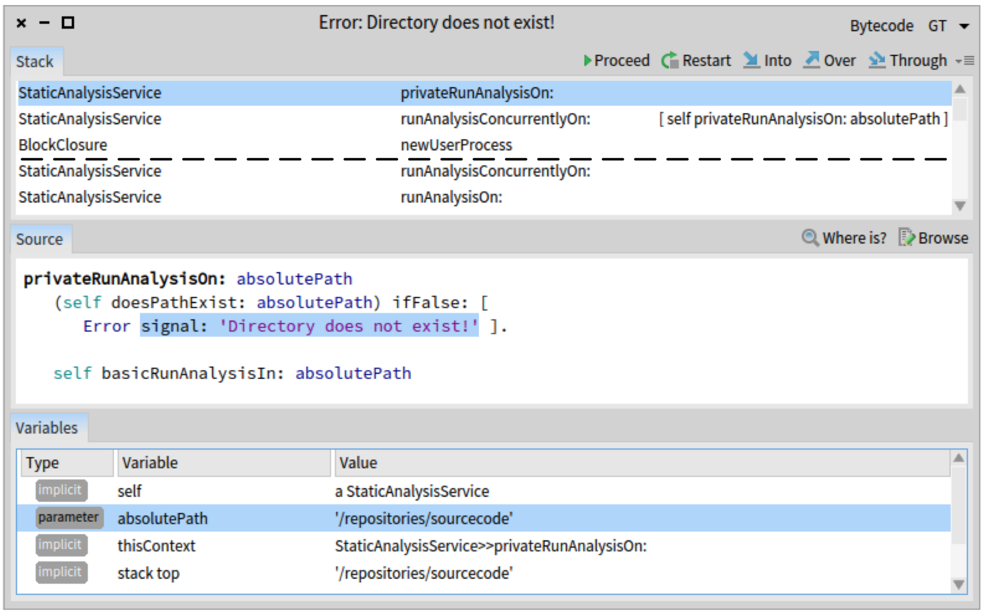
1. The original input was erroneous, as in line 60. `#absolutePathFrom:` will not modify *inputPath* since `'/sourcecode'` is already an absolute path. The value of *absolutePath*, which is being passed to the slave thread, is `'/sourcecode'`.

2. The original input was correct (line 61), however, the invocation of `#absolutePathFrom:` returned an erroneous path, *i.e.,* `'/repositories/sourcecode'` instead of `'/private_repositories/sourcecode'`. The value of *absolutePath*, which is being passed to the slave thread, is `'/repositories/sourcecode'`.

Without access to the master thread a developer cannot determine whether the user supplied an invalid input, or whether `#absolutePathFrom:` performed an erroneous modification, since the original input is not available in the debugger.

Figures 4.4, 4.5 and 4.6 show the thread debugger after an exception has been signalled in the slave thread. The point where the slave thread has been joined to its master in the virtual call stack is highlighted by a dashed line (slave thread above, master below the line). In Figure 4.4 the activation record in which the exception was signalled is shown. The stack frame selected in Figure 4.5 is the last activation record of the slave thread. The variable *inputPath* is not part of this activation record and cannot be compared with the method argument *absolutePath*.

Figure 4.6 shows the activation record that receives the method argument *inputPath* that the user supplied. This activation record is part of the master thread and has access to both the *inputPath* and *absolutePath* variables.



| | |
|---|---|
| StaticAnalysisService | privateRunAnalysisOn: |
| StaticAnalysisService | runAnalysisConcurrentlyOn: [ self privateRunAnalysisOn: absolutePath ] |
| BlockClosure | newUserProcess |
| StaticAnalysisService | runAnalysisConcurrentlyOn: |
| StaticAnalysisService | runAnalysisOn: |

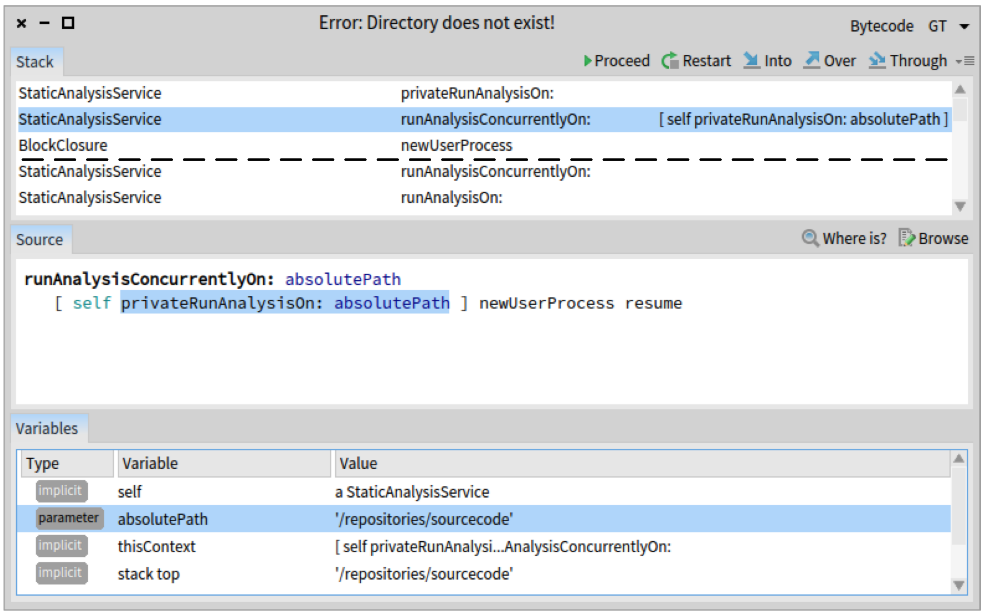
```
privateRunAnalysisOn: absolutePath
   (self doesPathExist: absolutePath) ifFalse: [
      Error signal: 'Directory does not exist!' ].

   self basicRunAnalysisIn: absolutePath
```

| Type | Variable | Value |
|---|---|---|
| implicit | self | a StaticAnalysisService |
| parameter | absolutePath | '/repositories/sourcecode' |
| implicit | thisContext | StaticAnalysisService>>privateRunAnalysisOn: |
| implicit | stack top | '/repositories/sourcecode' |

Figure 4.4: The top stack frame is part of the slave thread and shows where the error was signalled.

## 4.4 Application to special cases

Promises and events, like threads, have a history that is usually discarded. Debuggers should in both cases be able to show the place of activation but as shown in Chapter 2 only the Scala asynchronous debugger and the Chrome development tools support that. With a mechanism in place to copy the call stack of a thread, however, this problem becomes a special version of the problem of augmenting slave thread call stacks. The same is true for remote execution where we simply need a mechanism to link a local and a remote thread to each other.



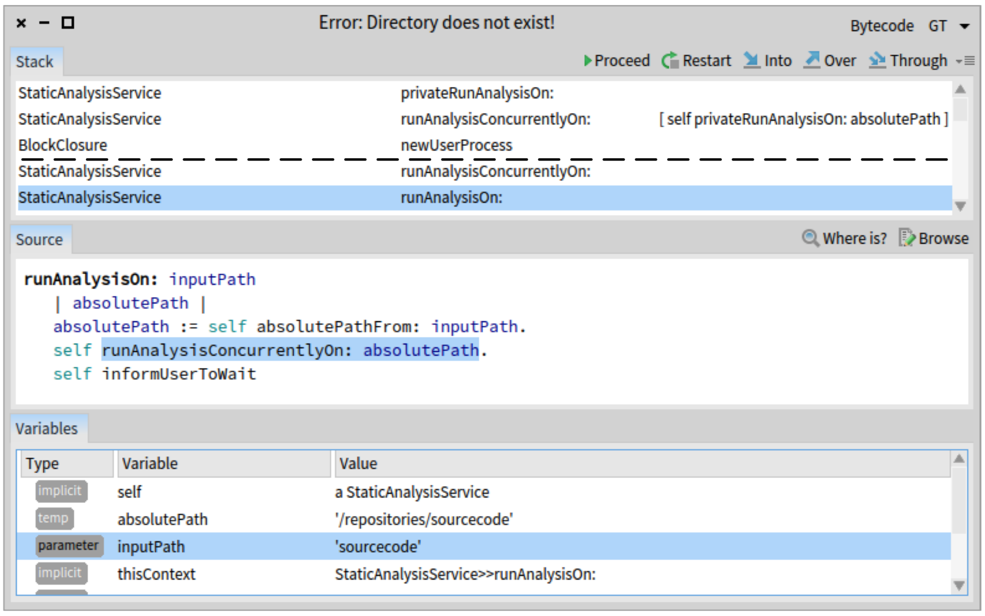
Error: Directory does not exist!

| | | |
|---|---|---|
| StaticAnalysisService | privateRunAnalysisOn: | |
| StaticAnalysisService | runAnalysisConcurrentlyOn: | [ self privateRunAnalysisOn: absolutePath ] |
| BlockClosure | newUserProcess | |
| StaticAnalysisService | runAnalysisConcurrentlyOn: | |
| StaticAnalysisService | runAnalysisOn: | |

```
runAnalysisConcurrentlyOn: absolutePath
    [ self privateRunAnalysisOn: absolutePath ] newUserProcess resume
```

| Type | Variable | Value |
|---|---|---|
| implicit | self | a StaticAnalysisService |
| parameter | absolutePath | '/repositories/sourcecode' |
| implicit | thisContext | [ self privateRunAnalysi...AnalysisConcurrentlyOn: |
| implicit | stack top | '/repositories/sourcecode' |

Figure 4.5: The selected stack frame shows the method `#runAnalysisConcurrentlyOn:`, the activation record is part of the slave thread. The variable *inputPath* is not visible.



Error: Directory does not exist!

| | | |
|---|---|---|
| StaticAnalysisService | privateRunAnalysisOn: | |
| StaticAnalysisService | runAnalysisConcurrentlyOn: | [ self privateRunAnalysisOn: absolutePath ] |
| BlockClosure | newUserProcess | |
| StaticAnalysisService | runAnalysisConcurrentlyOn: | |
| StaticAnalysisService | runAnalysisOn: | |

```
runAnalysisOn: inputPath
    | absolutePath |
    absolutePath := self absolutePathFrom: inputPath.
    self runAnalysisConcurrentlyOn: absolutePath.
    self informUserToWait
```

| Type | Variable | Value |
|---|---|---|
| implicit | self | a StaticAnalysisService |
| temp | absolutePath | '/repositories/sourcecode' |
| parameter | inputPath | 'sourcecode' |
| implicit | thisContext | StaticAnalysisService>>runAnalysisOn: |

Figure 4.6: Only in the activation record of `#runAnalysisOn:`, which is part of the master thread, is the variable *inputPath* visible and can be compared to the value of *absolutePath*.

### 4.4.1 Promises

Implementing promises with threads is a natural choice when threads are an option, as the result of the promise can be written to a shared variable and the threads can be synchronised through a semaphore. As long as the promise is created with a `UserProcess`, the debugger will not need to know whether it is debugging a promise or a regular thread.

```
62 promise := [ self runAnalysisConcurrentlyOn: absolutePath ] promise.
63 promise run.
64 self doOtherWorkConcurrently.
65 promise hasException ifTrue: [ promise debug ]
```

Listing 4-9: Example of executing the static analysis concurrently as a promise.

The sequence diagram in Figure 4.7 shows the important aspects of the promise execution. An object obtains a promise by sending `#promise` to a closure describing the statements to execute, as shown in Listing 4-9 on line 62. The message `#run` (line 63) starts the execution of the promise and returns immediately, such that the promise and the invoking thread are now being executed concurrently.

The promise creates a new `UserProcess`, which will create a copy of its master's call stack. The new `UserProcess` will execute the promise after receiving the message `#resume`. In our implementation we leave the choice of debugging a promise in case of an exception to the user, which means we have to store the failed `UserProcess` in the promise (by sending `#exceptionProcess:`) so that the user can send `#debug` at a later time (line 65). We must also suspend the thread since we don't want it to execute further and because threads being debugged must not be runnable by the scheduler (the thread in the debugger would be in an undefined state).

The message `#value` can be sent to a promise to obtain its result. In case of an exception in the promise thread, `#value` will answer a configurable value, wich is `nil` by default. To debug an exception, the user can send `#debug` to the promise or configure it to open a debugger immediately when an exception occurs, by sending `#openDebuggerOnError:` with the argument `true`. The only change for immediate debugging in the sequence of events depicted in Figure 4.7 is that the promise sends `#debug` to itself automatically after the thread has been suspended. Exceptions in the master are handled by the master thread itself, independently of the promise and the copy stored therein.

### 4.4.2 Remote communication

The following two subsections are concerned with threads and promises in remote environments. We use Seamless [25] in our implementation for the communication facilities, which is a framework for distributed computing that provides high adaptability by separating the different aspects of distributed communication, such as connection and authentication strategies, into modules. To provide transparent communication with remote threads, Seamless can use proxies, which is what our implementation relies upon.

Proxies allow us to interact with objects uniformly, whether they are local or remote objects. Thus, our debugger becomes a debugger for remote threads automatically and even supports virtual stacks with both local and remote `Context` instances.

### 4.4.3 Remote promises

Our implementation of promises can easily be extended to execute promises remotely by using a Seamless connection, as shown in Listing 4-10. The functionality for interacting with Seamless is implemented in the class `RemotePromise`. The challenge in the case of a remote execution is to

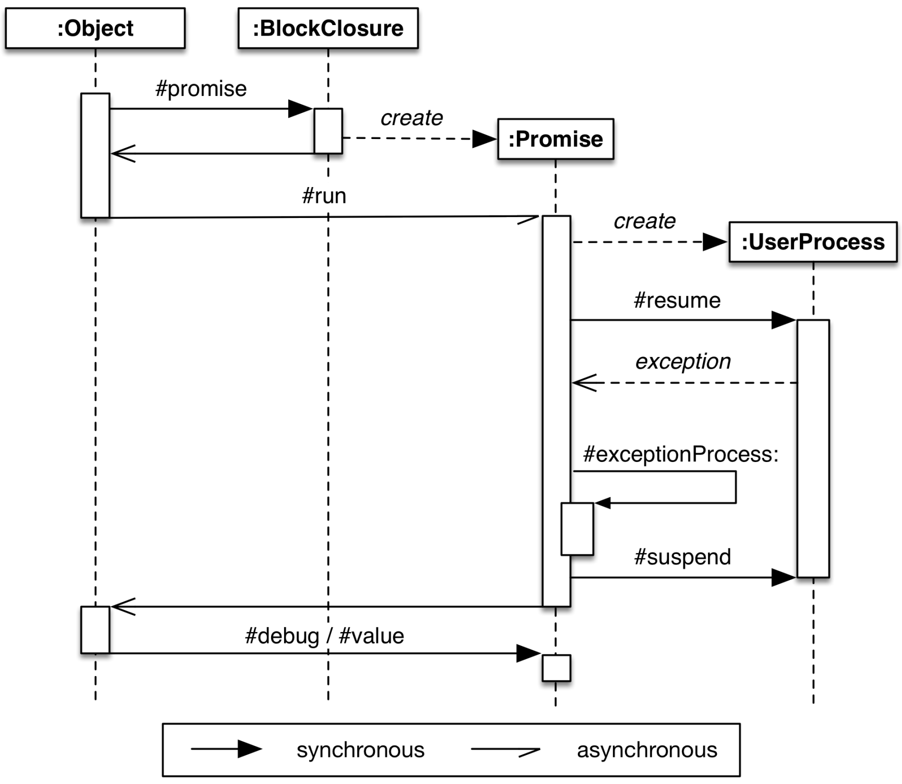


Figure 4.7: Sequence diagram of the execution of a promise.

bind the local master to the remote slave thread; Figure 4.8 depicts the sequence of operations that are performed in the remote environment to obtain a reference to the remote thread. The information we need to open a debugger must be present within the promise, so we pass a reference to the promise, an instance of `RemotePromise`, to the remote environment. We then execute the promised computations in a separate `UserProcess` thread, which allows us to intercept exceptions. When an exception occurs we store the promise computation thread in the promise that we passed by reference and finally suspend the promise computation thread, as we would for a regular promise. The garbage collector does not know about references between different environments. Fortunately, suspended threads are not considered garbage, so we do not need to create a strong reference to the thread.

In the local environment we need to perform one additional step so that our remote promise is equivalent to a local promise (not shown in Figure 4.8): we need to set the thread that activated the promise as the master of the suspended remote thread. The remote promise can now be opened in the debugger like a local promise and information about the promise thread will be fetched from the remote image through proxy objects.

As the remote thread is suspended and will not be garbage collected we must ensure that it will terminate eventually. Fortunately, the same is true for local threads in the debugger, which means the default debugger implementation already takes care of resuming or terminating the thread that is being debugged.

Since promises are a special case of threads in our implementation, the example for remote threads also demonstrates the feasibility of augmenting the call stack of remote threads in general. Hence, we will not address remote threads specifically.

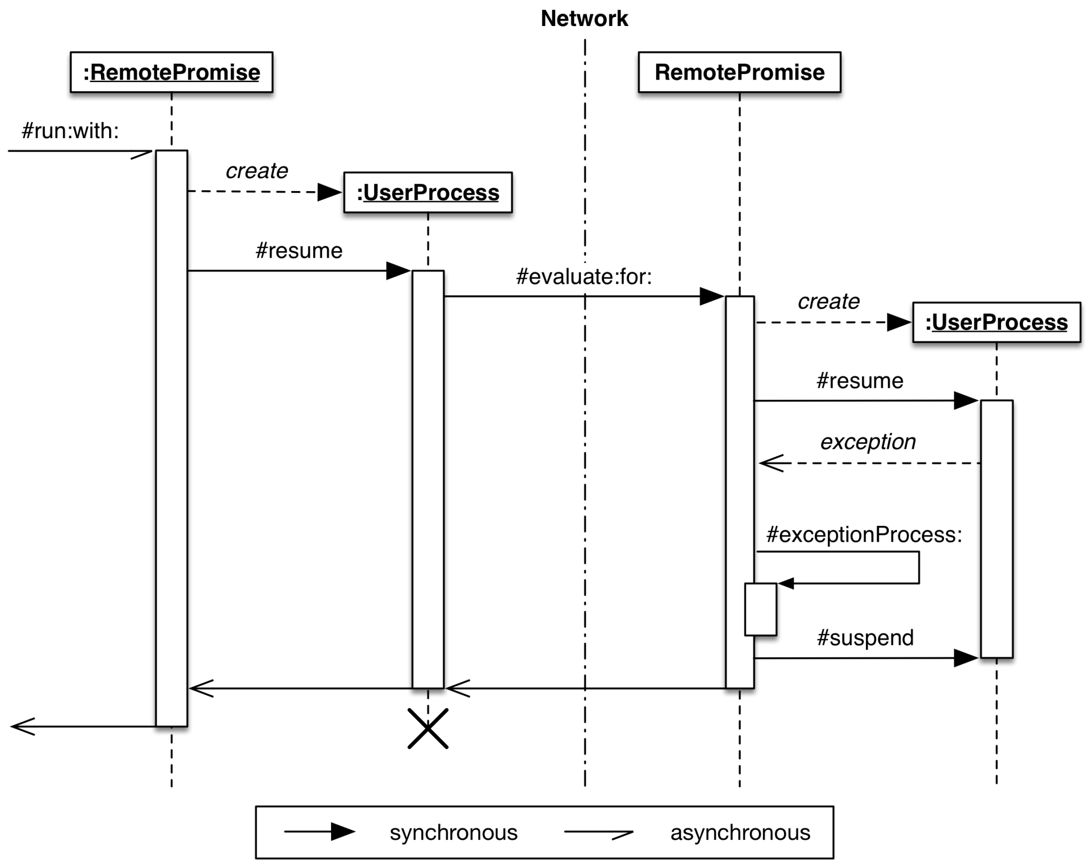


Figure 4.8: Sequence diagram of the execution of a remote promise.

```
connection := TCPAddress ip: #[127 0 0 1] port: 1111.
remotePromise := [ self runAnalysisConcurrentlyOn: absolutePath ] remotePromiseOn: connection.
remotePromise run.
self doOtherWorkConcurrently.
remotePromise hasException ifTrue: [ remotePromise debug ]
```

Listing 4-10: Example of executing the static analysis concurrently as a remote promise. The only difference to a regular promise is that the promise creation message takes a connection to the remote enviroment as argument.

## 4.5 Summary

We have shown how the model described in Chapter 3 can be implemented to retain the complete thread history and construct a virtual call stack comprised of multiple threads. Our implementation of a live thread debugger displays the activation records from the virtual call stack and ensures that actions performed on a given activation record are valid. The implementations given for promises and remote promises prove that the model of call stack augmentation can also be applied in other contexts to improve debugging.

# 5 Memory and performance considerations

Creating a copy of a call stack means creating a copy of each of its activation records. We consider the impact of this operation with respect to performance and memory, exemplified by our implementation in Pharo. Our analysis focuses on the 32-bit version of the virtual machine; a 64-bit virtual machine is currently in development.

## 5.1 Context reification

Activation records in Pharo are represented by instances of `Context`, equivalent to the `MethodContext` in Smalltalk-80. The virtual machine for Pharo, however, only creates contexts on demand [10], except for cases where a `Context` instance is needed anyway (Miranda [21] provides a list of such situations). Hence, creating a copy of an activation record in general entails the creation of two `Context` instances. To calculate how much memory is needed per instance we need to know how many bytes are needed to represent the object structure of contexts and what information is being stored.

Every regular object requires a header of 8 bytes, which is also true for contexts [4, 22]. `Context` defines the fields `stackp`, `method`, `closureOrNil` and `receiver`, and inherits `sender` and `pc`, each field being four bytes long. `Context` is also a so called "variable class", meaning it has a fixed number of unnamed, indexed slots. In contexts, these slots represent a stack with fixed capacity that is used to implement a stack machine for the activated method. Elements consumed by the bytcode instructions are pushed onto the stack, values produced by the instructions are popped from it. The number of slots, *i.e.,* the stack capacity, depends on the "large context flag" bit [13] of the `CompiledMethod` object header that is being activated, the value of which depends on the number of method arguments, literals, local variables and temporary values of the method. The flag is set to "large" by the compiler when the stack needs to hold more than a certain amount of elements at a time.

In the current virtual machine for Pharo the two states of the "large context flag" represent contexts of 16 (small context) or 56 (large context) variable slots. In a fresh Pharo 6 image (build 60143) the number of `CompiledMethod` instances is 94824, 344 (0.36 %) of which have the large context flag set. Using a single bit flag instead of the exact number of frames, which would require multiple bits, reduces the memory consumed by instances of `CompiledMethod`.

We can now determine the size of small `Context` instances:

$$\begin{array}{l} \quad 8 \text{ bytes for object header} \\ + 6 \times 4 \text{ bytes for instance variables} \\ + 16 \times 4 \text{ bytes for variable slots} \\ \hline = 96 \text{ bytes} \end{array}$$

Large `Context` instances have 56 instead of 16 indexed slots:

$$\begin{array}{l} \quad 96 \text{ bytes} \\ + (56 - 16) \times 4 \text{ bytes for variable slots} \\ \hline = 256 \text{ bytes} \end{array}$$

The contents of contexts do not require additional space when copied, as references are already accounted for in the field and variable slot sizes, and immediate values, such as integers, do not require additional memory allocation. In general, creating a copy of a context therefore requires $2 \times 96$ bytes $= 192$ bytes of additional storage in the best case, $2 \times 256$ bytes $= 512$ bytes in the worst one. Knowing that typical call stacks have sizes of tens or hundreds of frames [11, 27], we can estimate an upper bound of $1000 \times 512$ bytes $= 512$ kB of additional memory required for large call stacks with large methods. Even 512 kB are not much in comparison with the memory bound by the associated object graph.

## 5.2 Bound memory

Memory bound by call stack copies may be a bigger problem than the additional memory required for new contexts as the amount of memory occupied by the object graph rooted in a given context is potentially many times larger than the memory needed for the context itself. Contexts usually have a short life and objects referenced only from a context, such as those referenced by temporary variables, will be collected by the garbage collector after a short time. This is not the case when we create copies of contexts. The memory occupied by objects that would normally no longer be needed can only be reclaimed when the slave thread terminates. This is a problem for Pharo in particular because the 32-bit virtual machine can only allocate a certain amount of memory (around 2 GB, depending on the operating system)[1]. The memory bound by context copies is a problem even when the debugger is not being used, as we cannot know in advance whether or not a thread will be opened in a debugger and we therefore must create call stack copies proactively.

## 5.3 Computational overhead

The creation of a `UserProcess` instance incurs a performance penalty due to the copy operation on the active thread. This overhead is negligible however, as the `Object>>#copy` is implemented as a virtual machine primitive. With a stack of 100 000 frames the average and median times needed to copy the complete stack are less than 100 milliseconds as shown in Table 5.1. These numbers were obtained using the benchmark shown in Listings B-1 through B-3 in the appendix.

[1]Note that this limitation is inherent to 32-bit memory management

| | small context | | | large context | | |
|---|---|---|---|---|---|---|
| garbage collection time | average | median | max | average | median | max |
| included | 130.211 | 130 | 167 | 130.148 | 130 | 152 |
| excluded | 57.127 | 57 | 71 | 57.112 | 57 | 66 |

Table 5.1: Benchmark for copying a call stack of 100 000 frames, small (16 variable slots) and large (56 variable slots). The times are given in milliseconds and shown for bare computation time and computation including time needed for intermittent garbage collection.

Given that stacks typically contain less than 1000 frames [11, 27], performance clearly is not a problem. Furthermore, thread creation is an operation rarely performed (in relative terms), so that the overhead does not add up either. It is interesting to see that the size of the context does not seem to have any effect on the execution time even though it is clear that a larger context will require more operations to create a copy. It is possible that other operations mask the actual time it takes to create the copies, which would also explain the uniformity of the obtained results. Miranda [21] for example, describes how context reification in the VisualWorks virtual machine, which is related to the Pharo virtual machine, can cause large portions of memory to be rearranged.

## 5.4 Virtual machine support

Operations executed directly by a virtual machine take less time than interpreted instructions. `Context>>#copy`, the most expensive, because most frequent, operation in our implementation, is already a primitive. However, contexts are copied individually and the repeated sends of `#copy` occur in the interpreted space. Moving the complete stack copy procedure into the virtual machine would certainly improve performance. The following ideas could be used by an implementation in the virtual machine:

- the call stack could be duplicated lazily (only copy stack frames that are being returned from);
- operations could be split into chunks to ensure responsiveness;
- context reification could be optimised, for example by delaying reification or reifying only the copies.

As shown in Section 5.3, however, the computational overhead is already neglectable for most applications and as such we do not deem virtual machine support necessary with respect to performance.

A possibility to mitigate the problem of memory bound by the call stack copy may be to compress the memory of objects that are only being referenced from copied call stacks. Since these objects will only be accessed from the debugger, the additional time to decompress the memory should not be of concern. The additional time required for compression on the other hand may have a negative impact on performance, so that moving the stack copy procedure into the virtual machine may indeed become necessary. Another option might be to reclaim memory from copied call stacks when necessary, starting with the oldest copies.

## 5.5 Summary

Neither the amount of memory required by instances of `Context` nor the time required for copying call stacks pose problems for usual applications. The memory bound by the call stack copies, however, may be a problem, especially for Pharo with the current 32-bit virtual machine with its low memory allocation limit. To compensate for this it may be beneficial to add a flag to `UserProcess` that, when set, will prevent call stacks from being copied.

# 6 Conclusion and future work

We have shown in this work that it is feasible to augment thread call stacks in the debugger in order to provide access to the complete thread history. We have provided reference implementations for threads, promises and remote promises that make use of call stack augmentation in a customised debugger.

## 6.1 Conclusion

The goal of this work was to improve debugging of asynchronous threads in live debuggers. We realised that threads do not have access to their complete history and that therefore a debugger displays less information for concurrent programs than for sequential programs. To solve this problem of incomplete thread history we proposed to create a copy of the master thread at the point where a slave thread is being created and to make that copy available to the debugger. The debugger can then construct a virtual call stack, comprised of stack frames from multiple threads, to present the complete call stack history of a given thread. The implementations presented in Chapter 4 demonstrate that our proposal is technically feasible and that debugging of promises, remote execution and asynchronous events and messages can also be improved by using the same approach.

The discussion of downsides to our implementation showed that the creation of stack frame copies has only little impact on performance and that the memory consumed by the additional stack frames is generally negligible. Despite these observations, the issue of memory consumption requires further investigation, as the amount of memory bound by the object graph that is referenced solely by the copied stack frames may strongly increase the total memory consumption.

## 6.2 Future work

In this section we present possibilities for future research and improvements on the implementations discussed in Chapter 4.

### 6.2.1 Memory consumption

We did not perform any measurements of the memory that cannot be reclaimed because it is bound by call stack copies. Such measurements could answer the question whether the amount of memory bound by copied call stacks exclusively is large enough to warrant support by the virtual machine, *e.g.,* through memory compression or reclaiming such memory on demand. It was out of the scope of this work to perform these measurements as it would have meant to modify the garbage collector, which requires a good working knowledge of the virtual machine.

There is likely a limit to how many master call stacks it is useful to show in a debugger. This observation could be used to release memory when creating a new thread by ensuring that the number of master threads that hold a call stack copy is limited to a fixed number.

### 6.2.2 Logging

Our discussion of augmented call stacks has focused on live debuggers but in many cases live debuggers are not available in production environments. In environments where only logging of exceptions is available, being able to include the master thread's call stack in a stack trace could be very helpful to developers. With our proposed solution implementing a logger with the ability to access the master thread's call stack is trivial.

### 6.2.3 Implementation for processes

We have proposed an implementation for augmenting the call stack of threads but not of processes. Processes created using a function of the `fork`[1] family are almost exact duplicates of their masters and as such the thread history will stay intact[2]. Hence, for this type of process augmentation of the stack is unnecessary. Processes created by calls to functions of the `posix_spawn`[3] family, or by `fork` followed by a call to a function of the `exec`[4] family, on the other hand, are completely disjoint from their masters. Usually, such processes are also logically disjoint, *i.e.,* they perform a self contained task. There may be benefits in augmenting the stack of such a process' first thread, however, POSIX processes are executed within different address spaces and may have differing permissions. It may therefore be more complex to implement stack augmentation between processes, albeit possible.

### 6.2.4 Hiding stack frames

The call stacks in Pharo include activation records that are not of particular interest to users because they are part of some setup procedure. The methods `#newProcess` and `#newUserProcess`, for example, are part of the thread setup in Pharo. We chose to not hide any activation records for simplicity but it would be possible to do so by not including them in the virtual stack. Hiding activation records must be done with care because, even though such activation records may not be interesting to users, they are still integral parts of the threads they belong to.

### 6.2.5 Debugger user interface

The simple user interface presented in Chapter 4 lacks visual cues to distinguish different threads. In particular, users need to be able to identify where the slave thread ends. Other visual aids

[1]`http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html`
[2]`fork` only copies the active thread to the new process.
[3]`http://pubs.opengroup.org/onlinepubs/9699919799/functions/posix_spawn.html`
[4]`http://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html`

could designate stack frames that are still live, or cannot be acted upon.

The thread hierarchy view as found in the Visual Studio debugger could also be a valuable addition to the debugger. This view could be implemented as an alternative presentation of the the virtual stack and users could switch between presentations, depending on their needs and preferences.

### 6.2.6 Promises in Pharo

Pharo currently does not provide an implementation of promises as part of the core distribution while it has become a standard construct in languages such as Java, Scala, Ruby, Python, C# and JavaScript. Our prototype could be used as basis for such an implementation.

## 6.3 Summary

We have successfully shown how debugging of concurrent threads in live debuggers can be improved. Though the performance of our implementation is good, the increased memory requirements may present a problem in certain situations. Further research is needed to quantify the amount of memory bound by the call stack copies.

An extension of our idea to processes may be an interesting research topic, such that a debugger could show relationships between threads across different processes.

Finally, there is much work to be done on our debugger implementation before it is ready to be used productively, especially with respect to user experience.

# Bibliography

[1] R. M. Balzer. “EXDAMS: Extendable Debugging and Monitoring System”. In: *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference.* AFIPS ’69 (Spring). Boston, Massachusetts: ACM, 1969, pp. 567–580. DOI: `10.1145/1476793.1476881`. URL: `http://doi.acm.org/10.1145/1476793.1476881`.

[2] Boris Beizer. *Software testing techniques (2nd ed.)* New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN: 0-442-20672-0.

[3] John K Bennett. *The design and implementation of distributed Smalltalk.* Vol. 22. ACM, 1987.

[4] Clément Béra and Eliot Miranda. “A bytecode set for adaptive optimizations”. In: *Proceedings of the International Workshop on Smalltalk Technologies (IWST’14).* 2014.

[5] Andrew Black et al. *Pharo by Example.* Square Bracket Associates, 2009. ISBN: 978-3-9523341-4-0. URL: `http://pharobyexample.org`.

[6] Thomas Böttcher and Frank Huch. “A debugger for concurrent Haskell”. In: *Draft Proc. 14th Intl. Workshop on Implementation of Functional Languages (IFL’2002).* 2002, pp. 129–141.

[7] T.A. Cargill. “Pi: A Case Study in Object-Oriented Programming”. In: *Proceedings OOPSLA ’86, ACM SIGPLAN Notices.* Vol. 21. Nov. 1986, pp. 350–360.

[8] Andrei Chiş et al. “Practical domain-specific debuggers using the Moldable Debugger framework”. In: *Computer Languages, Systems & Structures* 44, Part A (2015). Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014), pp. 89–113. ISSN: 1477-8424. DOI: `10.1016/j.cl.2015.08.005`. URL: `http://scg.unibe.ch/archive/papers/Chis15c-PracticalDomainSpecificDebuggers.pdf`.

[9] Kim Clohessy, Brian Barry, and Peter Tanner. “New Complexities in the Embedded World - the OTI Approach”. In: *Object-Oriented Technologys.* Springer, 1997, pp. 472–478.

[10] L. Peter Deutsch and Allan M. Schiffman. “Efficient Implementation of the Smalltalk-80 system”. In: *Proceedings POPL ’84.* Salt Lake City, Utah, Jan. 1984. DOI: `10.1145/800017.800542`. URL: `http://webpages.charter.net/allanms/popl84.pdf`.

[11] David R. Ditzel and H. R. McLellan. “Register Allocation for Free: The C Machine Stack Cache”. In: *SIGPLAN Not.* 17.4 (Mar. 1982), pp. 48–56. ISSN: 0362-1340. DOI: `10.1145/960120.801825`. URL: `http://doi.acm.org/10.1145/960120.801825`.

[12] Daniel P. Friedman and David S. Wise. “Aspects of Applicative Programming for File Systems (Preliminary Version)”. In: *SIGSOFT Softw. Eng. Notes* 2.2 (Mar. 1977), pp. 41–55. ISSN: 0163-5948. DOI: `10.1145/390019.808310`. URL: `http://doi.acm.org/10.1145/390019.808310`.

[13] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Reading, Mass.: Addison Wesley, May 1983. ISBN: 0-201-13688-0. URL: `http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf`.

[14] Carl Hewitt, Peter Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

[15] Paul Hudak et al. “A history of Haskell: being lazy with class”. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM. 2007, pp. 12–1.

[16] The Open Group. “International Standard - Information technology Portable Operating System Interface (POSIX)Base Specifications, Issue 7”. In: *ISO/IEC/IEEE 9945:2009(E)* (Sept. 2009), pp. 1–3880. DOI: `10.1109/IEEESTD.2009.5393893`.

[17] Eileen Keremitsis and Ian J Fuller. “HP Distributed Smalltalk: A Tool for Developing Distributed Applications”. In: *HEWLETT PACKARD JOURNAL* 46 (1995), pp. 85–85.

[18] Robert A Kowalski. “The early years of logic programming”. In: *Communications of the ACM* 31.1 (1988), pp. 38–43.

[19] D. Kevin Layer and Chris Richardson. “Lisp Systems in the 1990s”. In: *Commun. ACM* 34.9 (Sept. 1991), pp. 48–57. ISSN: 0001-0782. DOI: `10.1145/114669.114674`. URL: `http://doi.acm.org/10.1145/114669.114674`.

[20] B. Liskov and L. Shrira. “Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: ACM, 1988, pp. 260–267. ISBN: 0-89791-269-1. DOI: `10.1145/53990.54016`. URL: `http://doi.acm.org/10.1145/53990.54016`.

[21] Eliot Miranda. *Context Management in VisualWorks 5i*. Tech. rep. ParcPlace Division, CINCOM, Inc., 1999.

[22] Eliot Miranda and Clément Béra. “A Partial Read Barrier for Efficient Support of Live Object-oriented Programming”. In: *Proceedings of the 2015 International Symposium on Memory Management*. ISMM ’15. Portland, OR, USA: ACM, 2015, pp. 93–104. ISBN: 978-1-4503-3589-8. DOI: `10.1145/2754169.2754186`. URL: `http://doi.acm.org/10.1145/2754169.2754186`.

[23] Daniel L. Murphy. “Storage Organization and Management in TENEX”. In: *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*. AFIPS ’72 (Fall, part I). Anaheim, California: ACM, 1972, pp. 23–32. DOI: `10.1145/1479992.1479996`. URL: `http://doi.acm.org/10.1145/1479992.1479996`.

[24] Nick Papoulias et al. “Mercury: Properties and Design of a Remote Debugging Solution using Reflection”. In: *Journal of Object Technology* (2015), p. 36.

[25] Nikolaos Papoulias. “Remote Debugging and Reflection in Resource Constrained Devices”. PhD thesis. Université des Sciences et Technologie de Lille-Lille I, 2013.

[26] Nancy Pennington. “Stimulus structures and mental representations in expert comprehension of computer programs”. In: *Cognitive Psychology* 19 (1987), pp. 295–341. ISSN: 0010-0285. DOI: `10.1016/0010-0285(87)90007-7`. URL: `http://www.sciencedirect.com/science/article/pii/0010028587900077`.

[27] Kavitha Srinivas and Harini Srinivasan. “Summarizing Application Performance from a Components Perspective”. In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 136–145. ISSN: 0163-5948. DOI: `10.1145/1095430.1081730`. URL: `http://doi.acm.org/10.1145/1095430.1081730`.

[28] Richard Stallman, Roland Pesch, Stan Shebs, et al. "Debugging with GDB". In: *Free Software Foundation* 51 (2002), pp. 02110–1301.

[29] Minyoung Sung et al. "Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread". In: *Information Processing Letters* 84.4 (2002), pp. 221–225. ISSN: 0020-0190. DOI: `10.1016/S0020-0190(02)00286-7`. URL: `http://www.sciencedirect.com/science/article/pii/S0020019002002867`.

[30] Gerald Jay Sussman and Guy L Steele. "The first report on Scheme revisited". In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 399–404.

[31] Paula Sue Utter and Cherri M Pancake. *Advances in Parallel Debuggers: New Approaches to Visualization.* Vol. 18. Cornell Theory Center, Cornell University, 1989.

[32] Lingli Zhang, Chandra Krintz, and Priya Nagpurkar. "Supporting Exception Handling for Futures in Java". In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java.* PPPJ '07. Lisboa, Portugal: ACM, 2007, pp. 175–184. ISBN: 978-1-59593-672-1. DOI: `10.1145/1294325.1294349`. URL: `http://doi.acm.org/10.1145/1294325.1294349`.

# A
# Installation instructions

1. Download a copy of Pharo version 6 (image and changes) from `https://pharo.org/download`.

2. Download the stable virtual machine for your platform from `https://pharo.org/download`.

3. Download the sources file from `http://files.pharo.org/get-files/50/sources.zip`, unpack it and place it next to the virtual machine.

4. Open the Pharo image with the virtual machine.

5. Download and install the code by pasting the following code into a playground and evaluating it. This will also install Seamless for the remote execution examples.

```
Metacello new
    baseline: MLThesis;
    repository: 'github://theseion/master-thesis:master/mc';
    load
```

The package "ML-Threads" contains the extension of the thread model, as well as the generic thread debugger implementation, the static analysis example from Chapter 4 and the code for benchmarking the stack copy operation (see Appendix B). The package "ML-Promise" extends the code from "ML-Threads" to implement simple versions of promises and remote promises.

`ThreadDebugger`, `Promise` and `RemotePromise` include executable examples on the class side demonstrating their functionality. Futher documentation is provided by the class comments and the unit tests.

# B
# Benchmark code

The benchmark from Chapter 5 can be reproduced in Pharo with the code from Listings B-1 and B-2 installed on the class side of a class named `StackCopyBenchmark` by evaluating `StackCopyBenchmark run`. The benchmark can also be found in source code package (see Appendix A). The results reproduced in Chapter 5 were obtained with the following combination of hardware and software:

- machine model: Apple MacBook Pro, late 2011;
- memory: 8 GB, 1333 MHz DDR3;
- CPU: 2.2 GHz Intel Core i7;
- operating system: OS X 10.11.5;
- PharoVM (Spur, 32-bit) build 589;
- Pharo 6 build 60086

```
run
    | timesSmall counter timesLarge withGCSmall withoutGCSmall withGCLarge withoutGCLarge |
    timesSmall := OrderedCollection new.
    timesLarge := OrderedCollection new.

    self assert: (self class >> #runLargeWith:collectingTimesInto:) frameSize = 56.
    self assert: (self class >> #runSmallWith:collectingTimesInto:) frameSize = 16.

    1000 timesRepeat: [
        Smalltalk garbageCollect.
        counter := 0.
        self
            runSmallWith: counter
            collectingTimesInto: timesSmall.

        Smalltalk garbageCollect.
        counter := 0.
        self
            runLargeWith: counter
            collectingTimesInto: timesLarge ].

    withGCSmall := timesSmall collect: #key.
    withoutGCSmall := timesSmall collect: #value.
    withGCLarge := timesLarge collect: #key.
    withoutGCLarge := timesLarge collect: #value.

    Transcript
        clear;
        open;
        show: 'small + GC max: ';
        show: withGCSmall max; cr;
        show: 'small + GC average: ';
        show: withGCSmall average asFloat; cr;
        show: 'small + GC median: ';
        show: withGCSmall median; cr; cr;
        show: 'small - GC max: ';
        show: withoutGCSmall max; cr;
        show: 'small - GC average: ';
        show: withoutGCSmall average asFloat; cr;
        show: 'small - GC median: ';
        show: withoutGCSmall median; cr; cr; cr;

        show: 'large + GC max: ';
        show: withGCLarge max; cr;
        show: 'large + GC average: ';
        show: withGCLarge average asFloat; cr;
        show: 'large + GC median: ';
        show: withGCLarge median; cr; cr;
        show: 'large - GC max: ';
        show: withoutGCLarge max; cr;
        show: 'large - GC average: ';
        show: withoutGCLarge average asFloat; cr;
        show: 'large - GC median: ';
        show: withoutGCLarge median; cr
```

Listing B-1: `StackCopyBenchmark>>#runBenchmark` runs the benchmark for both small and large contexts.

```
runSmallWith: counter collectingTimesInto: aCollection
    | before time after |
    counter < 100000
        ifTrue: [
           self
              runSmallWith: counter + 1
              collectingTimesInto: aCollection ]
        ifFalse: [
              before := Smalltalk vm totalGCTime.
              time := [ thisContext copyStack ] timeToRun asMilliSeconds.
              after := Smalltalk vm totalGCTime.
              aCollection add: time -> (before + time - after) ]
```

Listing B-2: `StackCopyBenchmark>>#runSmallWith:collectingTimesInto:` is the method used for benchmarking small contexts.

```
runLargeWith: counter collectingTimesInto: aCollection
    | temp1 temp2 temp3 temp4 temp5 temp6 temp7 temp8 before time after |
    temp1 := 'temp1'. temp2 := 'temp2'. temp3 := 'temp3'. temp4 := 'temp4'.
    temp5 := 'temp5'. temp6 := 'temp6'. temp7 := 'temp7'. temp8 := 'temp8'.

    counter < 100000
        ifTrue: [
           self
              runLargeWith: counter + 1
              collectingTimesInto: aCollection ]
        ifFalse: [
              before := Smalltalk vm totalGCTime.
              time := [ thisContext copyStack ] timeToRun asMilliSeconds.
              after := Smalltalk vm totalGCTime.
              aCollection add: time -> (before + time - after) ]
```

Listing B-3: `StackCopyBenchmark>>#runLargeWith:collectingTimesInto:` is the method used for benchmarking large contexts. The excessive number of temporary variables causes the large context flag to be set in the `CompiledMethod` object header.