A promising approach for debugging remote promises

Max Leske
Software Composition Group, University of Bern
maxleske@gmail.com

Andrei Chiş Oscar Nierstrasz
Software Composition Group, University of Bern
{andrei, oscar}@inf.unibe.ch

Abstract

Promises are synchronization constructs that hide the complexity of process synchronisation from the developer by providing a placeholder for the result of a potentially incomplete computation performed in a concurrent process.

Promises evaluated by remote processes pose challenges for debugging when the remote computation raises an exception. Current debuggers are either unaware that there is a problem in the remote computation or give developers access only to the context of the remote process. This does not allow developers to interact at the same time with the process that launched the promise and the remote process that executed the promise's computation.

To improve debugging of remote promises, in this paper we propose a debugger interface that presents a unified view of both the original and the remote process, by merging the call chains of the two processes at the point where the promise was created. We exemplify our approach, discuss challenges for making it practical, and illustrate through an initial prototype that it can improve debugging of exceptions in remote promises.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

Keywords remote debugging, remote promises, debuggers

1. Introduction

Developing concurrent programs is inherently more complex than developing non-concurrent programs, mainly due to the indeterminate order of access to shared resources and the complexity of process synchronisation. Promises provide one concept to simplify development of concurrent programs [1, 9]. A promise hides the complexity of process synchronisation from the developer by providing a placeholder for the value the concurrent process will return. The placeholder performs the necessary synchronisation when access to the value of the promise process is requested. Promises can execute their computation in a local *slave* process within the same address space as the *master* process that launched the promise or in a remote (slave) process running in another address space, on the same machine or another one. We refer to the latter as *remote promises* [12].

Due to the concurrent nature of promises, debugging them is often as complex as debugging conventional concurrent processes [13]. Furthermore, there exists little debugger support for promises in general and for promises involved with remote processes in particular. A developer can proceed to debug promises using generic techniques for debugging concurrent processes. One common approach is to simply open a debugger on the slave process when an exception is raised, as would be the case with a conventional process. Zhang *et al.* [20] proposed an *as-if-serial* exception handling mechanism that delivers exceptions to the same point as they would be delivered if the program were executed sequentially, as opposed to the point where the promise result is accessed.

While there exist approaches to improve exception handling for promises, most conventional debuggers still present only a view on the failed slave process, which is independent of the master process in the system that started the promise. This fails to capture the relationship between the master and the slave.

For processes running in the same address space it is often possible to reconstruct these relationships from contextual information. This approach is used by the developer tools of the Chrome browser to present asynchronous events in a sequential view [6]. In a remote execution setup however, processes are additionally separated by a communication channel which makes it harder to discover the inter-process relationships, and raises challenges related to remote communication.

To improve debugging of remote promises we propose a unified debugger interface that enables live debugging of remote promises within the context of their master process. To achieve this, when an exception is raised in the slave process of the remote promise, instead of showing the call stack of either the master or the slave, the debugger presents to the developer the call stack of *a single virtual process*. This virtual process merges the slave process with its master

at the point where the promise computation was started, to give a serial view of events. It further removes from the call stack frames related to communication with the slave.

To investigate the practical applicability of this idea we developed an initial prototype in the Pharo programming environment [4]. The prototype shows that our idea improves debugging of remote promises, however, it requires support from the virtual machine to reduce the memory footprint of the approach. Furthermore, to avoid transferring the entire stack of the slave process to the master, the current prototype relies on proxies that require permanent communication with the slave process.

The contributions of this paper are as follows:

- A model for debugging remote promises that provides a unified debugging view on the master process starting a remote promise, and the slave process executing the remote promise;
- A report on the technical challenges for making the approach practically feasible;
- A prototype implementation of the approach.

2. Motivation

In this section we illustrate the problem of debugging exceptions in remote promises with a motivating example in which a developer needs access to information stored in both the master and the slave processes. We start by clarifying terminology, introducing the example and discussing limitations of current debuggers.

2.1 Terminology

Processes / threads Throughout this paper we use the term *process* to mean *green thread* [18], unless stated otherwise. Green threads are virtual processes that share their memory and are located and scheduled in user space by a virtual machine.

A remote process is a green thread running in a different virtual machine, possibly executing on a different host. A process consists of a linked list of activation records, called a *stack*. We refer to an activation record also as a *context* or a *stack frame*. Newly activated methods are put on the top of the stack, so that the bottom context represents the starting point of the process.

Promises The concept of *promises* was introduced by Friedman *et al.* [9] 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 [12].

For this paper we define a promise as an eventual value, the computation of which may execute in a slave process. Our implementation of promises supports exception handling. A remote promise is a promise that is executed by a different virtual machine than its master process; the virtual machine may run on a different host.

2.2 A Motivating Example

Static analysis of source code is an expensive task which, to improve performance, can be run in a remote process. Our example program is a service that performs static source code analysis according to user-defined options. For simplicity we assume that the source code has already been placed in a local directory and that the options have already been processed and can be accessed by the remote promise.

```
1  runAnalysisOn: inputPath
2    | absolutePath promise |
3    absolutePath := self absolutePathFrom: inputPath.
4    promise := self createPromiseFor: absolutePath.
5    self informUserToWait.
6    self checkResult: promise value
```

```
7  createPromiseFor: absolutePath
8    | closure promise |
9    closure := [ self privateRunAnalysisOn: absolutePath ].
10   promise := closure remotePromiseOn: self connection.
11   ↑ promise
12       openDebuggerOnError;
13       run;
14       yourself
```

```
15   self runAnalysisInPath: '/sourcecode'.
16   self runAnalysisInPath: 'sourcecode'.
```

Code for launching a static analysis task using a remote promise is shown above. The method #runAnalysisOn: must receive as parameter a relative or absolute path to an existing directory, *i.e.*, either *"sourcecode"* or *"/private_repositories/sourcecode"*. The program performs the following steps:

1. Create an absolute path based on the inputPath parameter (line 3). When the input path is already absolute, absolutePath will have the same value as inputPath.
2. Initialize the analysis as a remote promise with the absolute path to the directory as input (lines 9 and 10).
3. Configure the promise to open a debugger when an exception occurs in the slave process (line 12).
4. Launch the remote promise which will attempt to access the directory at the given path (line 13).
5. Inform the user that the analysis is executing (line 5).
6. Attempt to access the result of the analysis (line 6 – #value) and block if the analysis is not finished.

When the remote promise 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 15. #absolutePathFrom: will not modify inputPath since '/sourcecode' is already an absolute path. The value of absolutePath, which is passed to the slave process, is '/sourcecode'.
2. The original input was correct (line 16) however, the call to #absolutePathFrom: returned an erroneous path *i.e.*, '/repositories/sourcecode' instead of '/private_repositories/sourcecode'. The value of absolutePath is then '/repositories/sourcecode'.

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

2.3 Limitations of current debuggers

Debuggers are a well-established tool in software development. Unfortunately, many debuggers used by imperative and object-oriented languages target debugging of non-concurrent processes.

These debuggers ignore that there exist explicit and implicit relationships between processes, such as the point where a new process is created. These relationships are important when investigating problems related to multiple processes. While it is possible in trivial cases to correctly guess the control flow across process boundaries more complex programs will leave developers at a loss as to what events occurred before the creation of a failing process. Remote processes connected to other environments through a communication channel such as a TCP network or a serial line further increase the complexity of debugging concurrent processes and of determining the control flow of the program across process boundaries.

Remote debuggers are specialised debuggers that can connect to other environments, possibly located on a different machine. They aim to improve remote debugging by enabling a developer to interact live with a remote system from their local environment. Nevertheless, like traditional debuggers, remote debuggers do not provide developers with enough information concerning the relationships between processes. This is especially true for promises where we are not aware of any model or implementation that specifically supports the debugging of promises, with the exception of Zhang *et al.* [20], who only deal with local and not remote promises.

2.4 Summary

Developers must be able to determine the relationships between the involved processes when debugging promises. Debugging remote promises adds an additional layer of complexity due to the communication channel between the master and slave processes. To the best of our knowledge no implementation of remote debuggers exists that explicitly supports the debugging of promises.

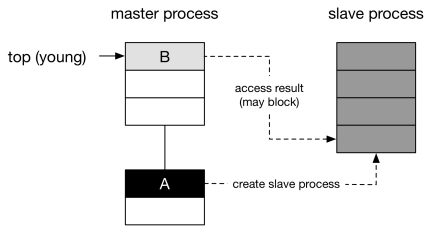

Figure 1. The master and slave processes before the slave terminates or fails. *(A)* marks the context in which the slave process was created, *(B)* the context which will access the value of the promise.

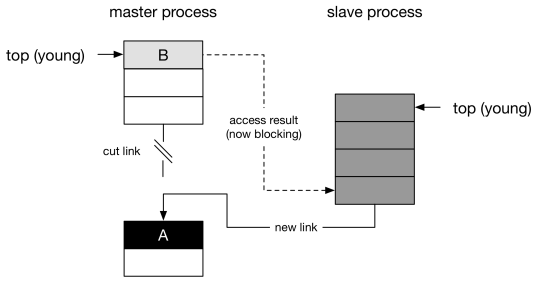

Figure 2. A new virtual process is created by linking one process to the other at the point where the slave process was created *(A)*.

3. A unified approach

To improve debugging of remote promises we propose that when an interrupt occurs in the slave process, *e.g.*, a breakpoint of unhandled exception, the developer is presented with a debugger that gives her access to the stacks of both the master and the slave processes.

It is paramount for developers to know the exact order of events when investigating a concurrency bug. Ergo, it is important to know the location where the remote promise was created. We follow the idea of Zhang *et al.* of presenting to the user the remote exception as triggered from the code location where the remote promise was created, instead of the location where the value of the promise was accessed.

3.1 Virtual process

To give the developer a serialised and unified view on both the master and the slave process we propose the use of a *virtual process*. A virtual process presents the user with a single call stack that merges the call stacks of the two actual processes at the place where the promise was created. This process is exemplified in Figures 1 and 2. Figure 1 shows the state of the master and slave processes before the slave process terminates or fails. When an exception is raised in the slave a virtual process is created by placing the slave process at the top of the stack, and the master process at the bottom (Figure 2). Hence, the serialized view of events is preserved.

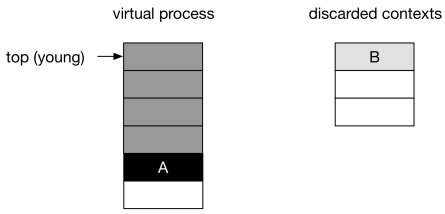

Figure 3. The new virtual process consists of both the master and the slave processes, joined at *(A)*. Contexts of the master process after the context where the slave was created are discarded from the virtual process.

Stack frames of the master process after the frame where the slave was created are discarded from the virtual process as is shown in Figure 3. When the virtual process is opened in a debugger the developer is presented with only a single, unified call stack.

In this paper we focus on the basic case of one master and one slave process. In practice, however, processes often form a hierarchy and each master is the slave of another process. To incorporate this into the presented approach, the virtual process can be extended to include master processes recursively; our current implementation of virtual processes, detailed in a related work [11], handles this scenario.

3.2 Remote Communication Through Proxies

Communication with the remote process should be as transparent as possible. Proxies provide a convenient way of communicating with remote objects that does not require any knowledge of the communication mechanism. In addition, proxies enable live debugging of the remote process in certain languages. In Pharo for example, the values of variables can be modified and the process can be resumed or even restarted at an earlier point.

A different strategy would be to marshal the object graph of the remote process and present this object graph with local objects. Variables and objects could still be inspected but restarting or resuming the process would not be possible.

To allow the slave process to be resumed, we base our approach on remote communication through proxies.

3.3 Addressing the motivating example

In the motivating example from Section 2 the slave process can fail as a result of line 16 if the absolute directory path computed in line 3 and passed to the remote process does not exist. To determine the root cause of the error a developer needs to examine both the master and the slave processes and determine if there was an error in computing the remote path or if the path does not actually exist on the server. With our approach this can be done in a single debugger.

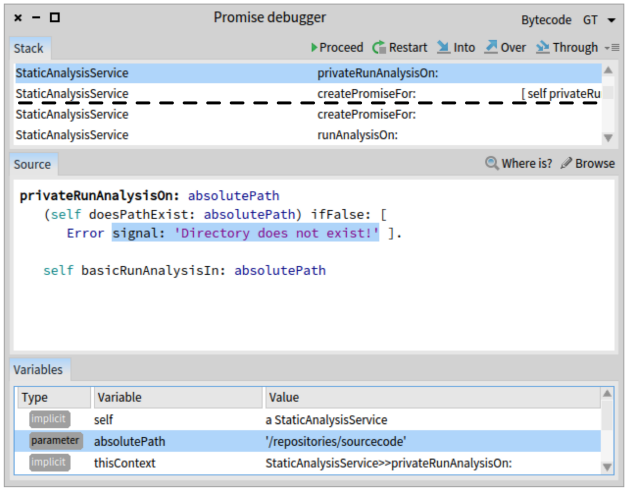

Figure 4. The source of the exception in the slave process. The dashed line indicates the point where the slave process has been joined to its master. Note that the parameter absolutePath contains an absolute path.

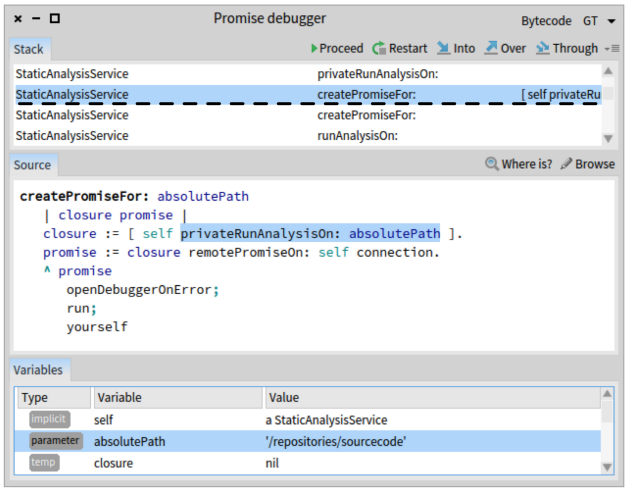

Figure 5. The context in which the promise was created. The context of #runAnalysisOn:, through which the variable inputPath can be accessed, is not part of the slave process.

Figures 4, 5 and 6 show three screenshots of the proposed debugger opened after an exception was raised in the slave process[1].

The debugger shows the stack of the aforementioned virtual process. The slave and the master process are joined at the point indicated by the dashed line.

In Figure 4 the selected stack frame is the one where the exception was raised and is part of the slave process. The method parameter absolutePath holds the erroneous absolute path that was passed to the slave process. Figure 5

[1] A video showing this process is available at:
`https://vimeo.com/maxleske/promisedebugger-iwst2016`

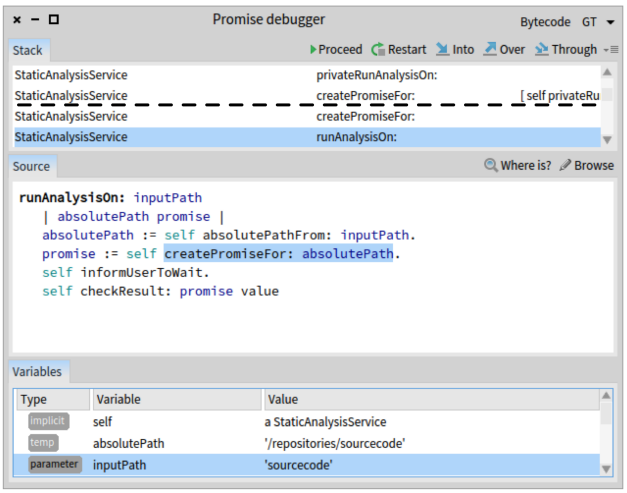

Figure 6. The original input inputPath contains only a directory name. This information is only available to the master process in the selected context.

shows the bottom context of the slave process. The context of #runAnalysisOn: with the method parameter inputPath is not part of the slave process. With only the slave process available it would be impossible to decide whether inputPath contained an erroneous value or if #absolutePathFrom: performed an erroneous modification of inputPath to produce the value of absolutePath.

The stack frame selected in Figure 6 is part of the master process. The method parameter inputPath contains the original input, a directory name without path delimiters (*"sourcode"*). Since we know that a directory with that name exists at *"private_repositories/sourcecode"* the method #absolutePathFrom: is clearly the source of the error and should have supplied the prefix *"private_repositories"* instead of *"repositories"*.

By combining the information from the slave process with that of the master process in a single debugger it is possible determine the control flow of the program accross process boundaries and identify the root cause of the problem presented in the example.

4. Implementation

In this section we present our prototype implementation[2] in Pharo. We chose Pharo because processes can be manipulated without the need to modify the virtual machine. Given enough resources however, we believe that it is feasible to implement our approach in any other programming language with a stack-based implementation.

[2] The prototype can be downloaded from the following URL: `http://scg.unibe.ch/download/promisedebugger/iwst16.zip`

4.1 Promise interface

```
17   | promise |
18   closure := [ self compute ].
19   promise := closure remotePromiseOn:self connection.
20   promise
21     timeout: 5 seconds;
22     onTimeout: [ self error: 'Computation timed out' ];
23     defaultReturnValue: 0;
24     onError: [ :error | self error: 'Error in computation' ];
25     openDebuggerOnError;
26     run.

28   self inform: promise value.
```

A promise in our implementation contains a closure. The message #remotePromiseOn: creates a promise from a closure. Sending #run to the promise evaluates the closure in the slave process. The promise can be *redeemed* by sending the #value message, which returns the computed result. If the result is not yet available #value will block until it is ready. Our prototype also provides additional configuration options:

- #timeout: and #onTimeout: control the maximum run time of the promise and an optional action upon reaching this timeout.
- #defaultReturnValue: specifies the value to return in case of an exception. The default is nil.
- #openDebuggerOnError opens a promise debugger automatically when an exception is signalled and not handled. The default is to return the value specified by #defaultReturnValue:.

4.2 Creating a serialised view of events

In Pharo a process is represented by a linked list of contexts, as shown in Figure 7. The standard debugger UI traverses this list of contexts to render it. A simple approach to presenting a serialised view of multiple processes in the standard debugger is therefore to manipulate the call chain links. The sender field of the bottom context (the starting point of the process) is always nil. We can create a single virtual process out of two separate processes (Figure 1) by making the top context of one process the sender of the bottom context of the other, *i.e.*, we store a reference to the top context of another process in the sender field as depicted in Figure 2.

Manipulating the sender field of other contexts allows us to hide the contexts that are internal to the promise creation. These hidden contexts must not be lost, as they are still required *e.g.*, for correct process termination, but hiding them simplifies the view. The debugger implementation must take care to restore the original context links whenever a hidden process would be accessed in the original process. The following debugger actions can make such a modification necessary:

- resuming the process;
- restarting the process at any of the visible contexts;

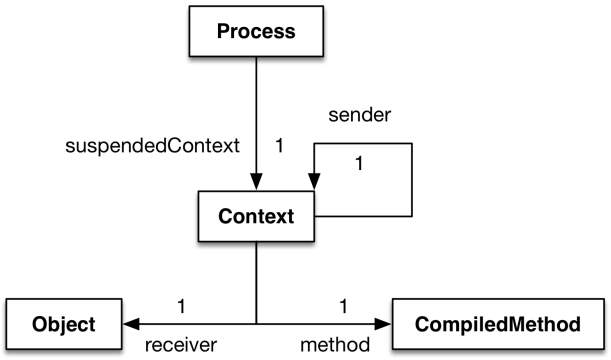

Figure 7. UML diagram of processes and their chain of contexts.

- terminating the process;
- stepping over one or more instructions;

Contexts must be relinked in these cases so that return values are correctly passed along the context chain and to ensure the evaluation of all *unwind* constructs [8], *i.e.*, constructs requiring a cleanup activity during process termination, such as closing an open file.

4.3 Discerning live and dead contexts

A dead context is one that must not be executed by the virtual machine. While it is possible to execute arbitrary contexts, it may not be possible to guarantee a state in which their execution will be successful, such as for copies of contexts whose originals have already terminated.

All dead contexts in the master process must be treated as such in the debugger while, at the same time, all live contexts must support the usual debugger actions. We therefore need a way to discern the liveness of a context on a per instance basis. The current debugger determines liveness at the process level, regardless of the context in focus. We extend the debugger and override the liveness check to be performed on the context level and take into account the process, *i.e.*, master or slave, to which a given context belongs.

4.4 Saving the master process

Upon creation of a promise we create a copy of the master process. This is necessary because we cannot predict how the promise will be used. There are different possibilities:

1. the promise is redeemed at a point in the future (possibly within a different method),
2. the promise is redeemed in a method earlier in the stack (*e.g.*, by accessing a field that was written in the preceding method invocation),
3. the promise is not redeemed in the master process (possibly in another process),
4. the promise is redeemed in the master process but with a timeout, or
5. the promise is never redeemed.

Only in the first case is the context that created the promise guaranteed to be live during the lifetime of the promise itself. For the first case we could simply record a pointer to the context in which the promise was created. That context will remain live as long as the master process is waiting on the promise. This is not true for the second case. The context we recorded still exists but has been released, *i.e.*, its internal state has been cleared. Even worse, as in case three, the master process may exit without waiting on the slave process at all. Cases four and five are special cases of case three.

To accommodate all of these cases we must create a copy of the master process. The copy of a process is simply the copy of all the contexts in the call chain. By creating a copy we ensure that all contexts retain the internal representation they had at the point of the copy creation.

4.5 Remote communication

The communication facilities we use to debug the remote process are provided by Seamless [15]. Seamless 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 processes, Seamless can use proxies, which is what our implementation relies upon.

While our promise implementation is independent of the location of execution (the promise itself is always a local construct) working with remote objects requires extra effort. There are two locations where the details of remote communication become relevant to us. The first is the case where an exception occurs in the remote process. We must ensure that all the information we need will be retained and that we send back the correct proxies. The second point is during the setup of the debugger where we have to create local objects for certain proxies for performance reasons. The proxies concerned are those that receive many messages due to UI interaction.

Apart from these two framework-dependent points our implementation is independent of the remote communications facility used.

4.6 Handling exceptions

Our promise implementation employs an internal exception handler that catches all unhandled exceptions in the slave process. We use this handler to prepare the default return value and the internal state for the case in which the developer requests to open a debugger on the intercepted exception. When the intercepted exception belongs to a slave process we also create local copies of all the contexts in that process. This is necessary to prevent the excessive number of mes-

sages sent to contexts from the UI from being sent to the remote image.

5. Challenges for a practical solution

The manipulations we perform on processes come at a cost. Additional costs when debugging are acceptable as long as the tools remain subjectively usable. It is important however, that these additional costs not lead to a performance degradation in the general case.

5.1 Performance overhead

The creation of a promise incurs a performance penalty largely attributable to copying of the master process. There are two sources that contribute to the penalty:

- creating and copying contexts
- context reification

The first point is obvious: for every context in the call chain a new instance of context must be created and the values of the original context must be copied to the new instance.

The second point however is more complex. In modern Smalltalk VMs an activation record is only represented as a context on demand [14]. One advantage of this is that fewer object space manipulations are necessary when activating a method, and not all activation records need to be garbage-collected [14]. In essence, process execution speed is improved by reifying contexts on demand. The downside to this approach is that context reification itself becomes more complex and therefore slower. Creating a copy of a context is a request that triggers reification [14]. Creating a copy of a context that has not yet been reified therefore entails the creation of two contexts, internal management by the VM, and the actual copy operation.

5.2 Memory overhead

The copy of a context is shallow, meaning that primitive values of the context are copied as well as first level references. The minimum size of a context in a 32-bit Pharo VM with Spur memory manager is 32 bytes consisting of the object header (12 bytes) and the instance variables sender, receiver, method, closureOrNil, pc and stackp (4 bytes each). In addition, contexts represent the method stack, whose size depends on the activated method and includes method arguments, temporary variables and the return value (4 bytes each). The memory overhead of copying contexts is therefore linear to the number of contexts. Given a stack size of 1000 contexts (a rather large process stack) with an average method stack size of 7 (3 method parameters, three temporary variables) the memory required for the copy amounts to 39 kilo bytes.

Apart from requiring additional memory, storing contexts may prevent objects referenced by these contexts from being collected by the garbage collector. The virtual machine may have to allocate more memory to compensate for the memory blocked by these objects which in turn may lead to problems if not enough memory is available.

6. Related work

The need for debuggers that make use of the relationships between processes has been recognized as early as 1986 [5, 19]. One possibility of giving developers access to the inter-process relationships are traces, which have been used for sequential debugging since 1969 [2]. Traces provide serialized records of the events that have occurred during the execution of a program. Visual concurrent debuggers use the information from traces to present visualisations to the user that attempt to highlight dependencies between processes. Both trace debugging and visual debugging of concurrent processes have their strengths but do not provide support for live debugging.

Remote debuggers exist for many languages and platforms. The GNU debugger (GDB) for instance can be configured to communicate with a remote target via a serial or network interface [17]. In Smalltalk derivatives a number of remote debuggers have been implemented [3, 7, 10]. One of the most recent of these is *Mercury* [16]. Mercury does neither specifically target the problem of debugging concurrent processes nor that of debugging promises.

Zhang *et al.* proposed a serialised view of the events of both the slave process and its master [20]. 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 creates a promise 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). Unlike our promise implementation, theirs does not support remote execution.

6.1 Overview of live debuggers

Too many programming languages exist too give an exhaustive overview of live debuggers in this paper (Leske [11] provides a comprehensive list of live debuggers and concurrency related features). Hence, we focus on debuggers that are used by many developers, and therefore represent the state of the art for a large number of users, or can display the history of promises. Table 1 shows a brief list of live debuggers and their support for showing the history of promises, both locally and remotely.

The table includes the following debuggers: Java Debugger Interface[3] (Java), Visual Studio debugger[4] (C#, C++, Visual Basic .NET, JavaScript), GDB[5] (C), Chrome development tools[6] (JavaScript), Scala asynchronous debug-

[3] http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/

[4] https://msdn.microsoft.com/en-us/library/sc65sadd.aspx

[5] https://www.gnu.org/software/gdb/

[6] https://developer.chrome.com/devtools

	promise history	remote promise history
Java Debugger Interface (JNI)	·	·
Visual Studio debugger	·	·
GDB	·	·
Chrome development tools	✓	·
Scala asynchronous debugger	✓	·
Pharo debugger	·	·
Pharo promise debugger	✓	✓

Table 1. Support for displaying the promise history in live debuggers.

ger[7] (Scala). Only the Chrome development tools and the Scala asynchronous debugger can display the history of local promises. What this shows is that the concept is an important idiom in the respective languages, more important at least than in other languages. Our promise debugger is the sole debugger to support promise history for remote promises.

7. Conclusions and future work

Debugging exceptions in remote promises is a challenging task as developers have to reason about both the process that started the promise and the process that executed the promise. To address this aspect, this paper proposed the use of a single virtual process that links the two processes at the point where the promise was created. We integrated this approach into a concrete debugger from the Pharo IDE.

We are currently working on further improving the visualisation of the different parts of the virtual process, so that developers can more easily tell the difference between master and slave processes. We are also looking at incorporating support for showing all processes that are waiting on the promise that is being debugged.

The idea of visualising process relationships is interesting enough to experiment with a more general implementation that would allow to see these relationships for any given process, not just for promises. Such an implementation would require a setting to turn it on and off however, to prevent performance degradation. The virtual machine can provide facilities to mitigate performance degradation.

[7] `http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html`

There is currently work being done on an official implementation of a remote debugger for Pharo and we are discussing the idea of bringing the serialised view of master and slave processes to this tool.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

References

[1] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, Aug. 1977. ISSN 0362-1340. doi: 10.1145/872734.806932. URL `http://doi.acm.org/10.1145/872734.806932`.

[2] R. M. Balzer. Exdams: Extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, AFIPS '69 (Spring), pages 567–580, New York, NY, USA, 1969. ACM. doi: 10.1145/1476793.1476881. URL `http://doi.acm.org/10.1145/1476793.1476881`.

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

[4] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL `http://pharobyexample.org`.

[5] T. Cargill. Pi: A case study in object-oriented programming. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 350–360, Nov. 1986.

[6] P. Chen. Debugging asynchronous JavaScript with Chrome DevTools, July 2014. `http://www.html5rocks.com/en/tutorials/developertools/async-call-stack/`.

[7] K. Clohessy, B. Barry, and P. Tanner. New complexities in the embedded world - the OTI approach. In *Object-Oriented Technologys*, pages 472–478. Springer, 1997.

[8] D. P. Friedman and C. T. Haynes. Constraining control. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 245–254, New York, NY, USA, 1985. ACM. ISBN 0-89791-147-4. doi: 10.1145/318593.318654. URL `http://doi.acm.org/10.1145/318593.318654`.

[9] D. P. Friedman and D. S. Wise. Aspects of applicative programming for file systems (preliminary version). *SIGSOFT Softw. Eng. Notes*, 2(2):41–55, Mar. 1977. ISSN 0163-5948. doi: 10.1145/390019.808310. URL `http://doi.acm.org/10.1145/390019.808310`.

[10] E. Keremitsis and I. J. Fuller. HP Distributed Smalltalk: A tool for developing distributed applications. *HEWLETT PACKARD JOURNAL*, 46:85–85, 1995.

[11] M. Leske. Improving live debugging of concurrent threads. Masters thesis, University of Bern, Aug. 2016. URL `http://scg.unibe.ch/archive/masters/Lesk16a.pdf`.

[12] 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, pages 260–267, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.54016. URL `http://doi.acm.org/10.1145/53990.54016`.

[13] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, Dec. 1989.

[14] E. Miranda. Context management in VisualWorks 5i. Technical report, ParcPlace Division, CINCOM, Inc., 1999.

[15] N. Papoulias. *Remote Debugging and Reflection in Resource Constrained Devices*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2013.

[16] N. Papoulias, N. Bouraqadi, L. Fabresse, S. Ducasse, and M. Denker. Mercury: Properties and design of a remote debugging solution using reflection. *Journal of Object Technology*, page 36, 2015.

[17] R. Stallman, R. Pesch, S. Shebs, et al. Debugging with GDB. *Free Software Foundation*, 51:02110–1301, 2002.

[18] M. Sung, S. Kim, S. Park, N. Chang, and H. Shin. Comparative performance evaluation of Java threads for embedded applications: Linux thread vs. Green thread. *Information Processing Letters*, 84(4):221 – 225, 2002. ISSN 0020-0190. doi: 10.1016/S0020-0190(02)00286-7. URL `http://www.sciencedirect.com/science/article/pii/S0020019002002867`.

[19] P. S. Utter and C. M. Pancake. *Advances in Parallel Debuggers: New Approaches to Visualization*, volume 18. Cornell Theory Center, Cornell University, 1989.

[20] L. Zhang, C. Krintz, and P. 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, pages 175–184, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-672-1. doi: 10.1145/1294325.1294349. URL `http://doi.acm.org/10.1145/1294325.1294349`.