Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Improving live debugging of concurrent threads through thread histories ☆

Max Leske [a], Andrei Chiş [b,*], Oscar Nierstrasz [a]

[a] Software Composition Group, University of Bern, Switzerland
[b] Feenk GmbH, Switzerland

ARTICLE INFO

Article history:
Received 19 January 2017
Received in revised form 2 October 2017
Accepted 10 October 2017
Available online xxxx

Keywords:
Concurrency
Debugging
Promises
Smalltalk
Domain-specific tools

ABSTRACT

Concurrency issues are inherently harder to identify and fix than issues in sequential programs, due to aspects like indeterminate order of access to shared resources and thread synchronisation. Live debuggers are often used by developers to gain insights into the behaviour of concurrent programs by exploring the call stacks of threads. Nevertheless, contemporary live debuggers for concurrent programs are usually sequential debuggers augmented with the ability to display different threads in isolation. To these debuggers every thread call stack begins 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. This requires developers to manually link stack traces belonging to related but distinct threads, adding another burden to the already difficult act of debugging concurrent programs. To improve debugging of concurrent programs we address the problem of incomplete call stacks in debuggers through a thread and debugger model that enables live debugging of child threads within the context of their parent threads. The proposed debugger operates on a virtual thread that merges together multiple relevant threads. To better understand the features of debuggers for concurrent programs we present an in-depth discussion of the concurrency related features in current live debuggers. We test the applicability of the proposed model by instantiating it for simple threads, local and remote promises, and a remote object-oriented database. Starting from these use cases we further discuss implementation details ensuring a practical approach.

1. Introduction

According to Pennington [29], 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 [3] in particular, such as erroneous conditional expressions or invalid state transitions, the ability to navigate the call stack helps developers to identify and fix

☆ This work is an extended version of a previous work: A promising approach for debugging remote promises. In: Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies (IWST 2016), http://doi.acm.org/10.1145/2991041.2991059.

* Corresponding author.
E-mail addresses: maxleske@gmail.com (M. Leske), chisvasileandrei@gmail.com (A. Chiş).
URLs: http://andreichis.com (A. Chiş), http://scg.unibe.ch/staff/oscar (O. Nierstrasz).

https://doi.org/10.1016/j.scico.2017.10.005
0167-6423/

issues. 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.

Concurrent programs, however, distribute computation by forking and joining multiple threads. A single thread is in itself a sequential program that can create new threads, which are executed concurrently. We call the original thread the *parent* and any thread created by the parent a *child*. Parent threads may await termination of their child threads—which is called "joining"—or they may not. Child threads can themselves be parents of other threads, thus forming a hierarchy rooted in the first thread of the operating system's launch process, which has no parent. In this context, reasoning about control flow in concurrent programs requires developers to investigate different types of relations between threads, such as their histories, how they interact and exchange data, or how they synchronize their executions. We use the term *history* to describe the complete call stack of a thread including the activation records of all its parent threads.

The need for debuggers that show these types of relationships between threads has been recognised as early as in 1986 [37,7]. One possibility for giving developers access to the inter-thread relationships are traces, which have been used for sequential debugging since 1969 [2]. Traces provide serialised records of the events that have occurred during the execution of a program. 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 [37] give an excellent, albeit outdated, overview of parallel visual debuggers). Trace debugging offers a powerful way to analyse issues in a program, especially related to concurrency. However, traces are usually costly to process, due to the large amount of data they contain, 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. Examples include debuggers for Lisp [18], and Smalltalk [15], where this functionality is built into the language environment, as well as debuggers for Java[1] and C++. For all their strengths, live debuggers have not yet reached their full potential when it comes to debugging threads. Developers would certainly profit from being presented with more information about threads and their relationships in live debuggers.

Towards this goal, in this paper we explore how to integrate into live debuggers one type of relationship between threads, namely the complete history of a thread. While approaches exist to improve debugging of concurrent programs, most conventional debuggers still present only a view on the failed child thread, which is independent of the parent thread. This fails to capture the relationship between the parent and the child. For threads 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 [8]. In a remote execution setup however, threads are additionally separated by a communication channel which makes it harder to discover the inter-thread relationships, and raises challenges related to remote communication. Concurrent models like promises or actors that build on top of threads, while simplifying the creation of concurrent programs, introduce further challenges for recovering a thread's history. This happens as live debuggers need to take into account how threads are used internally by those concurrency models.

To improve debugging of concurrent programs using both local and remote threads, as well as other concurrent models, we propose to address the problem of incomplete thread history through a unified thread and debugger model that enables live debugging of child threads within the context of their parent threads. To achieve this, when an exception is raised in the child thread, instead of showing the call stack of either the parent or the child, the debugger presents to the developer the call stack of *a single virtual thread*. This virtual thread merges the child thread with its parents, to give a serial view of events. If the parent thread is waiting for the completion of multiple child threads, the virtual stack only shows the history of the child thread that contained the error. We retain the history of a child thread by creating a copy of the parent's call stack at the point where the child is being created. While this idea is conceptually straightforward, it raises many design challenges. Memory footprint, performance of copying activation records, and ensuring live debugging actions (*e.g.*, step into, step over) in the presence of a virtual thread, are some of the most important.

To investigate the practical applicability of a debugger model that works with a virtual thread, we apply the proposed model to four different contexts:

- *Local threads:* We start with the basic case where a parent thread creates a child thread in the same address space. We present a prototype implementation of a debugger in the Pharo programming environment [5] that extends the thread model provided by Pharo;
- *Local promises:* We extend the base approach by replacing the child thread with a promise. We rely on the TaskIt[2] library for working with promises and build a debugger that takes into account TaskIt promises. As promises represent another type of concurrency model for synchronizing communication in a concurrent application, this shows that the presented approach is not just tied to basic threads;
- *Remote promises:* In the previous two cases both threads exist in the same address space. We explore challenges that arise when the child thread is located in a remote environment by building an implementation of remote promises

[1] https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.
[2] https://github.com/sbragagnolo/taskit.

Please cite this article in press as: M. Leske et al., Improving live debugging of concurrent threads through thread histories, Sci. Comput. Program. (2017), https://doi.org/10.1016/j.scico.2017.10.005