Towards Advanced Debugging Support for Actor Languages

Studying Concurrency Bugs in Actor-based Programs

Carmen Torres Lopez
Vrije Universiteit Brussel
Pleinlaan 2, 1050
Brussel, Belgium
ctorresl@vub.ac.be

Stefan Marr, Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
firstname.lastname@jku.at

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Pleinlaan 2, 1050
Brussel, Belgium
egonzale@vub.ac.be

ABSTRACT

With the ubiquity of multicore hardware, concurrent and parallel programming has become a fundamental part of software development. If writing concurrent programs is hard, debugging them is even harder. The actor model is attractive for developing concurrent applications because actors are isolated concurrent entities that communicates through asynchronous message sending and do not share state, thus they avoid common concurrency bugs such as race conditions. However, they are not immune to bugs. This paper presents initial work on a taxonomy of concurrent bugs for actor-based applications. Based on this study, we propose debugging tooling to assist the development process of actor-based applications.

Keywords

Concurrency; Bug; Debugging; Actor-based languages; Event-loop concurrency

1. INTRODUCTION

Identifying the root cause of concurrency bugs is hard, perhaps even an art, and providing tooling for the debugging of concurrent programs is a challenge. Concurrent programs, unlike traditional sequential programs, often exhibit *non-deterministic* behavior, which makes debugging more complex. For instance, the communication or synchronization between concurrent entities (e.g. processes, threads, actors [2]) can be sensitive to timing, which makes it hard to reproduce bugs. The debugging mechanisms itself may also modify the behavior of the program while monitoring its execution, hampering debugging even further [21]. This is a phenomenon similar to the Heisenberg Uncertainty [17], also known as *probe effect* [9].

Our first research question is what types of concurrency bugs appear in complex concurrent programs. The answer to this question depends on the concurrency model in which the program is written. Most existing surveys of concurrency bugs focus on thread-based concurrency [19, 1, 5]. Currently, there are no similar surveys for other concurrency models, and the established terminology does not directly applied to non-shared-memory concurrency models. Therefore, in this paper we study concurrency bugs in message passing concurrent software, in particular, actor-based programs.

The actor model is attractive for concurrent programming because it avoids by design many well-known concurrency issues related to thread-based programs. Since actors do not share mutable state, programs cannot exhibit memory-level race conditions, e.g., data races. In addition, deadlocks can be avoided if communication between actors is solely based on asynchronous message passing. However, programs can still be subject to deadlocks if the actor-based language provides blocking operations (such as in Erlang).

This paper focuses on exploring debugging tooling for concurrent programs written in one of the offsprings of the actor model: communicating event loops [22]. To define the features a debugger should provide, we first study which concurrency issues appear in concurrent actor-based programs. Second, we review which features debuggers for actor-based languages currently provide. We include debuggers for Erlang,[1] Scala,[2] JavaScript [10], E [24] and AmbientTalk [4]. Finally, we propose new types of breakpoints to simplify the identification of the root cause of concurrency bugs.

The contributions of this paper are:

- We conduct a systematic study of concurrency bugs in actor-based programs. To the best of our knowledge it is the first attempt to define a taxonomy of bugs in the context of actor-based concurrent software.

- We present a catalog of features that a debugger for actor-based programs should support to help the debugging of lack of progress issues and message-level race conditions.

2. CLASSIFICATION OF CONCURRENCY BUGS IN ACTOR-BASED PROGRAMS

The actor model was first proposed by Hewitt [13]. Since then, several additional variations of it emerged. The main variants model actors as 1) active objects (e.g. ABCL [27],

[1]http://erlang.org/doc/apps/debugger/debugger_chapter.html
[2]http://scala-ide.org/docs/current-user-doc/features.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

AGERE '16 Amsterdam, Netherlands

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

AmbientTalk/1 [7]), 2) processes (e.g. Erlang [3], Scala) or 3) communicating event loops (e.g. E [22], AmbientTalk/2 [25], JavaScript). Depending on the guarantees provided by the specific actor model at hand, programs may be subject to different concurrency issues. In the remainder of this section, we review the concurrency issues for all actor models to get a full overview. We divide the study in two categories: lack of progress issues and race conditions. An overview is given in table 1.

	Lack of Progress	Race Conditions
Thread-based programs	Deadlock Livelock	Data race Bad interleaving Atomicity violation Order violation
Actor-based programs	Communication deadlock Behavioral deadlock Livelock	Bad message interleaving Message protocol violation

Table 1: Taxonomy of concurrency bugs

2.1 Lack of Progress Issues

Deadlocks and livelocks are the most common concurrency bugs that lead to a lack of progress in a concurrent system. Compared to thread-based programs, these issues manifest themselves differently in actor-based ones. A program is in a *livelock*, when actors make local progress, but they prevent the program from making global progress. For example, actors that receive and execute messages but they do not send messages to other actors, preventing global progress.

Deadlocks in thread-based programs happen when two or more threads are suspended waiting for each other to finish a computation. In contrast, a deadlock in an actor-based program happen when two or more actors *conceptually* wait on each other because the message to complete the next step in an algorithm is never sent. In this case, there is no actor which is actually suspended. We call this situation a *behavioral deadlock*, because the mutual waiting prevents local progress. However, these actors might still process messages from other actors. Since actors never actually suspend, detecting behavioral deadlocks may be much harder than detecting deadlocks in thread-based programs.

```
1 play() ->
2   Ping = spawn(fun ping/0),
3   spawn(fun() -> pong(Ping) end).
4
5 ping() ->
6   receive
7     pong -> ok
8   end.
9
10 pong(Ping) ->
11   Ping ! pong,
12   receive
13     ping -> ok
14   end.
```

Listing 1: Communication deadlock (from [6])

Certain variants of the actor model can be subject to the traditional deadlocks known from thread-based programs. For example, Erlang and Scala Actors framework [12] have blocking `receive` operations, on the contrary of Akka [3], which `receive` operation is not blocking [11]. When the `receive` operation is blocking, can lead to what in Erlang is known as *communication deadlocks* [6]. A communication deadlock occurs when an actor only has messages in its inbox that cannot be received with the currently active `receive` statement. Listing 1 shows a communication deadlock example in Erlang discussed by Christakis and Sagonas [6]. In line 12 the `pong` process is blocked because it is waiting for a message that is never sent by the `ping` process, instead the `ping` process returns 'ok'.

[3]Lightbend Inc. Akka. http://akka.io

2.2 Race Conditions

Since actors do not share memory and messages are processed serially, low-level memory race conditions common to thread-based programs cannot occur. In particular, actor programs avoid low-level data races (i.e. simultaneous memory access errors), bad interleavings (also known as atomicity violations [23]), which refer to errors cause by overlapping execution of two threads), and order violations (i.e. out-of-order memory accesses errors) [23].

Nevertheless, all actor-based programs can have *high-level* race conditions related to the order or timing of messages. We consider these *high level* to distinguish them from the memory access-level ones. In particular, we identified *bad message interleavings* and *message protocol violations*.

In the original actor model, when an actor sends a message to a recipient actor, the message is placed in a mail queue and is guaranteed to be eventually delivered by the actor system. For distributed systems on top of Scala, ActorFoundry, or AmbientTalk, however, in-order delivery of the messages is not guaranteed, i.e. communication links between actors are not enforced to work in a FIFO manner. This can be the source of bad interleavings of messages. We define a *bad message interleaving* as the condition when a message is processed in between two messages which are expected to be processed one after the other, causing some misbehavior of the application or even a crash.

Listing 2 shows an example of message sending between two actors, `Server` and `Client`, where a bad message interleaving can occur. In line 10, the `Client` sends an asynchronous message to the `Server` to store the value 1. In line 11, the `Client` does a `call`, which waits for a result, to retrieve the value from the `Server`. This code can cause problems if the server receives the first `get` and before the `set` message. In this case, the values of `v1` and `v2` will be inconsistent.

```
1 class Server extends Actor {
2   int value = 0;
3   @message void set(int v) { value = v; }
4   @message int  get()      { return value; }
5 }
6 class Client extends Actor {
7   ActorName server;
8   Client(ActorName s) { server = s; }
9   @message void start() {
10    send(server, "set", 1);
11    int v1 = call(server, "get");
12    int v2 = call(server, "get");
13    assert v1 == v2;
14  }
15 }
```

Listing 2: Bad message interleaving (based on [16])

Bad message interleavings can also occur within a single actor if programs can receive notifications for external events, e.g. events from the network or sensors. Such issues

have been reported in the context of JavaScript's event loop concurrency model by Hong et al. [14].

Another kind of race condition related to message ordering is what we call a *message protocol violation*. This issue can appear when two or more actors exchange messages that are not consistent with the intended *protocol* of an actor. This issue is a generalization of bad message interleavings, and is also known as *ordering problems* [16, 18]. They typically are caused by actors only supporting a subset of all possible message sequences. Messages that come out of order or in unexpected interleavings can then cause inconsistent states or high-level race conditions.

Listing 3 shows an example of message protocol violation in AmbientTalk. The example tries to transfer money between two accounts, which are realized as actors. To ensure transactional semantics, the `txMng` transaction manager actor is used. First, a transaction is started for the two involved customers. Once the transaction is ready, i.e., when the resulting future is resolved (line 3), three messages are sent to the involved actors: `withdraw`, `deposit`, and `finishTransaction` (lines 4-6). However, this example does not use futures to make sure that `withdraw` and `deposit` are finished before completing the transactions.

```
1 when: txMng<-startTransaction(c1, c2)@FutureMessage
2 becomes: { |tx|
3   tx.from<-withdraw(10);
4   tx.to<-deposit(10);
5   txMng<-finishTransaction(tx);
6 }
```

Listing 3: Message protocol violation example in AmbientTalk

3. DEBUGGING TOOLS FOR ACTOR LANGUAGES

This section reviews the state of the art in debuggers for actor languages and identifies missing features that could help developers to identify the root cause of concurrent bugs.

3.1 State of the Art

Causeway [24] is a post-mortem debugger for distributed communicating event-loop programs in E [22]. It focuses on displaying the *causal relation of messages* to enable developers to determine the cause of a bug. Causality is modeled as the partial order of events based on Lamport's happened-before relationship [15].

REME-D [4] is an online debugger for distributed communicating event-loop programs written in AmbientTalk [25]. REME-D provides message-oriented debugging techniques such as the *state inspection*, in which the developer can inspect an actor's mailbox and objects, while the actor is suspended. It also supports a catalog of breakpoints, which can be set on asynchronous and future-type messages sent between actors. Like Causeway, REME-D allows inspecting the history of messages that were sent and received when an actor is suspended, also known as *causal link browsing* [4].

In the context of JavaScript, the Chrome DevTools online debugger supports Web Workers,[4] which are actors that communicates with the main actor through message passing [10]. The Chrome debugger allows to pause *workers*. In the case of *shared workers* it also provides mechanisms to inspect, terminate, and set breakpoints.[5] For debugging messages and promises on the event loop, Chrome also supports *asynchronous stack traces*. This means, it shows the stack at the point a callback was scheduled on the event loop. Since this works transitively, it allows to infer the point and context of how a callback got executed.

Erlang has an online debugger which supports *line, conditional, and function breakpoints*. It also supports stepping through processes and inspected their state.

Scala also features an online debugger supporting stepping, line and conditional breakpoints. Furthermore, it has an extension to facilitate debugging of actor programs.[6] One can follow a message send and *stop in the receiving actor*. Additionally, the debugger supports asynchronous stack traces similar to Chrome [8].

3.2 Position Statement

Section 2 showed that actor-based programs are not immune to lack of progress issues and race conditions. Those concurrency bugs can thus cause a program to misbehave or even to crash. Nonetheless, our literature study reveals that actor-specific debugger features are rare.

REME-D allows online debugging based on message breakpoints, stepping message sends, and inspecting message history. Causeway emphasizes tracking the causal relation of messages for post mortem analysis. Scala's and Chrome's online debugger provides asynchronous stack traces and the Scala debugger features minimal message stepping.

We argue that these features are rather basic compared to the possible complexity of concurrency bugs in actor-based programs. Further research is needed in order to improve the debugging process, in particular to browse and manipulate message histories to combine them with online features such as rich stepping and breakpoint options.

In order to help developers to more directly identify the cause of bugs, we believe that it is necessary to increase the flexibility of message breakpoints both on the sender and the receiver side, enable syntax-token-level granularity of breakpoints, together with recording the causality of messages. For example, we believe that displaying the causal relationships of messages might help identifying the concurrency issue shown in listing 2 which illustrates a bad message interleaving. Ideally, a visualization could also highlight based on the source code that certain messages are independent of each other, because there is no direct ordering relationship between them.

4. KÓMPOS: A DEBUGGER FOR COMMUNICATING EVENT-LOOP LANGUAGES

Kómpos is a breakpoint-based online debugger for programs written in communicating event-loop languages. It is meant to be a research platform to experiment with new debugger features that are meant to help developers to identify concurrency bugs in actor-based languages. Our current prototype is built it for the SOMns language [20], a Newspeak offspring implementing event-loop concurrency based on the Truffle language implementation framework [26].

[4]https://www.w3.org/TR/workers/

[5]http://blog.chromium.org/2012/04/debugging-web-workers-with-chrome.html

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

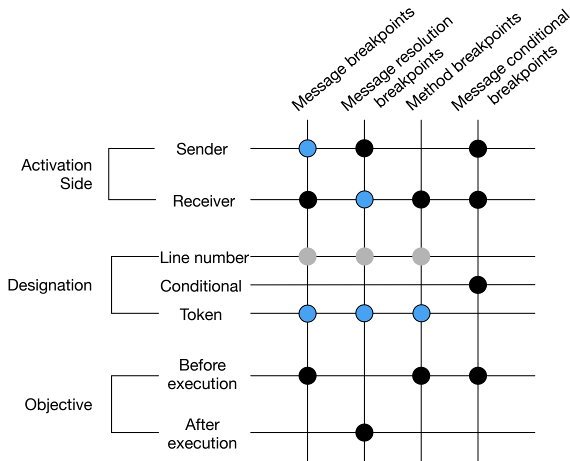

Figure 1: Extension of REME-D's breakpoint catalog. Black dots represent features from REME-D, gray dots represent features not added in Kómpos, and blue dots are features new in Kómpos.

Kómpos builds on REME-D's features and extends its breakpoints catalog to support additional debugging scenarios. Figure 1 depicts the breakpoint types, which are classified by the following three properties:

- The *activation side* determines the place where the breakpoint is triggered. A *sender-side breakpoint* triggers before the message is sent. A *receiver-side breakpoint* triggers before the actor processes the received message.
- The *designation* distinguishes how breakpoints are defined. *Line number* and *syntax token* characterize breakpoints based on source locations. Syntax token breakpoints are especially useful to set breakpoints on the message-send operator.

  Conditional breakpoints are triggered when a user-defined pre or post condition is fulfilled for a turn, i.e., before or after a message is processed.
- The *objective* defines whether the goal of the breakpoint is to halt the execution *before* or *after* a message is processed.

In the remainder of this section, we discuss the new breakpoints in Kómpos that go beyond the ones in REME-D.

Message breakpoint are defined on the asynchronous message send operators. In listing 3 one could set for instance a breakpoint on the <− operator in line 3. If it was defined for a sender activation side, an actor's execution is suspended right before the message would be sent. If it was defined with a receiver activation side, the actor's execution is suspended before the message is processed.

Distinguishing these two activation sides for message breakpoints allows us to debug the sender side by inspecting whether the message to be sent has the correct values, as well as it enables us to debug the receiver side. The receiver breakpoint allows us to see whether the expected actor received the message and whether the actor is in the expected state before processing the message. By distinguishing these two breakpoints, developers have fine-grained control over program execution, which hopefully helps to identify bugs where the root cause is related to an actors state or behavior. Such bugs could result in a wide variety of symptoms including but not limited to lack of progress issues and race conditions.

A message resolution breakpoint is defined on the operation that creates a future. In Kómpos, this means we can put a breakpoint on SOMns' asynchronous send operator, or an explicit operation to create a future. The sender-side breakpoint on futures pauses execution before the computed value is used to resolve the future, i.e., before it is sent. Similarly to REME-D's message resolution breakpoints, this allows developers to inspect the sender and its state after the message is processed. The receiver-side breakpoint on futures, on the other hand, pauses the execution potentially at many different places. All actors that have registered a callback to be executed when the future is resolved will be suspended before executing the callback. An actor may be thus suspended more than once, once per each callback it registered. Since SOMns features future pipelining, the breakpoint also causes suspension of actors before processing previously scheduled messages that are sent eventually to the future's result once becomes resolved.

5. CONCLUSION

Although the actor model avoids data races and deadlocks by design, it is still possible to find lack of progress issues and message-level race conditions on actor programs. Our systematic review of concurrency bugs showed that actor-based programs written in languages like Erlang and Scala can exhibit communication deadlocks because the actor implementation still features blocking operations. Such kinds of communication deadlocks cannot happen in event loop concurrency, but nevertheless, bad message interleaving and message protocol violation can still occur.

When reviewing the state of the art on debugging support for actor languages we observe that some debuggers provide features based on setting message breakpoints, inspecting the history of messages and support for asynchronous stack traces. We argued that better tools for debugging concurrent programs are needed in order to identify the cause of bugs. To this end, we propose Kómpos, an experimental platform for online debugging concurrent programs based on the communicating event-loop model. Kómpos propose an extension of REME-D catalog, adding the possibility of stopping at sender and receiver side of an asynchronous message, and enabling syntax-token-level granularity. We believe that those features could be very helpful when debugging actor based programs.

In the future we will focus on finishing the implementation of the breakpoint catalog and we will work on the combination of strategies such as recording the causality of messages with some of the breakpoints implemented. We aim to explore the notion of a debugger that can adapt its features to the concurrency model in which a program is written.

6. ACKNOWLEDGMENTS

This research is funded by a collaboration grant of the Austrian Science Fund (FWF) with the project I2491-N31 and the Research Foundation Flanders (FWO Belgium).

References

[1] S. Abbaspour, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Software Quality Journal*, pages 1–34, 2016.

[2] G. Agha. *Actors: A model of concurrent computation in distributed systems*. PhD thesis, MIT, Artificial Intelligence Laboratory, June 1985.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[4] E. G. Boix, C. Noguera, and W. De Meuter. Distributed debugging for mobile networks. *Journal of Systems and Software*, 90:76–90, 2014.

[5] M. Brito, K. R. Felizardo, P. Souza, and S. Souza. Concurrent software testing: A systematic review. *On testing software and systems: Short papers*, page 79, 2010.

[6] M. Christakis and K. Sagonas. Static Detection of Deadlocks in Erlang. Technical report, June 2011.

[7] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. In *European Conference on Object-Oriented Programming*, pages 230–254. Springer, 2006.

[8] I. Dragos. Stack retention in debuggers for concurrent programs. In *Scala Workshop*, 2013.

[9] J. Gait. A debugger for concurrent programs. *Software: Practice and Experience*, 15(6):539–554, 1985.

[10] I. Green. *Web Workers: Multithreaded Programs in JavaScript*. " O'Reilly Media, Inc.", 2012.

[11] P. Haller. On the Integration of the Actor Model in Mainstream Technologies. *AGERE! 2012*, pages 1–5, Nov. 2012.

[12] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, Feb. 2009.

[13] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[14] S. Hong, Y. Park, and M. Kim. Detecting Concurrency Errors in Client-Side Java Script Web Applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, pages 61–70. IEEE, Mar. 2014.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[16] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of java-based actor programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 468–479. IEEE Computer Society, 2009.

[17] C. H. LeDoux and D. S. Parker Jr. Saving traces for ada debugging. In *ACM SIGAda Ada Letters*, number 2 in ACM, pages 97–108. Cambridge University press, 1985.

[18] Y. Long, M. Bagherzadeh, E. Lin, G. Upadhyaya, and H. Rajan. On ordering problems in message passing software. In *Proceedings of the 15th International Conference on Modularity*, pages 54–65. ACM, 2016.

[19] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. .

[20] S. Marr and H. Mössenböck. Optimizing communicating event-loop languages with truffle, October 2015. URL http://stefan-marr.de/papers/agere-marr-moessenboeck-optimizing-communicating-event-loop-languages-with-truffle/.

[21] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989. ISSN 0360-0300. .

[22] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers. In *International Symposium on Trustworthy Global Computing*, pages 195–229. Springer, 2005.

[23] S. Park. Debugging non-deadlock concurrency bugs. In *Proceedings of the 2013 international symposium on software testing and analysis*, pages 358–361. ACM, 2013.

[24] T. Stanley, T. Close, and M. Miller. Causeway: A message-oriented distributed debugger. Technical report, HP Labs, Apr. 2009.

[25] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12. IEEE Computer Society, 2007.

[26] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. In *ACM SIGPLAN Notices*, volume 48, pages 73–82. ACM, 2012.

[27] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in abcl/1. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 258–268, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. .