What's the Problem? Interrogating Actors to Identify the Root Cause of Concurrency Bugs

Carmen Torres Lopez
ctorresl@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Louise Van Verre
Louise.Van.Verre@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Elisa Gonzalez Boix
egonzale@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

Programs written using Communicating Event-Loops (CEL) concurrency model do not suffer from low-level data races by design but are not exempt from other concurrency bugs, such as behavioral deadlocks and message order violations.

When programmers need to find the root cause of a bug, they typically ask questions about the application's behavior. However, current debugging tools are mostly operational, offering features at the source code level like breakpoints and watchpoints. Consequently, understanding the program behavior when debugging can take a lot of time for developers since questions on behaviors need to be mapped into operations in the debugger.

Inspired by interrogative debugging, this paper proposes an interactive debugging approach for actor-based programs that enable developers to reason about the program output by selecting questions from a set of predefined questions about the code and the program's execution. We present the design of the questions and answers, and we describe a prototype implementation in Apgar, an online debugger for actor-based programs written in SOMns. We define questions based on key concepts of the actor model: actors, turns, messages, and promises. The debugger then computes the answers by analyzing a recorded trace of events about the program execution.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**; **Software testing and debugging**;

Keywords: debugging, concurrency, user interface, bugs

ACM Reference Format:
Carmen Torres Lopez, Louise Van Verre, and Elisa Gonzalez Boix. 2021. What's the Problem? Interrogating Actors to Identify the Root Cause of Concurrency Bugs. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486601.3486709

AGERE '21, October 17, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9104-7/21/10…$15.00
https://doi.org/10.1145/3486601.3486709

1 Introduction

Debugging concurrent programs is a difficult and time-consuming task. Developers need to understand how the different concurrent entities interact and deal with problems inherent to concurrency, such as non-determinism and non-repeatability. Besides, the distance between the root cause of a bug and its failure can be large in the program, which requires big efforts from developers to identify and fixing the bug [14].

When a program fails, programmers usually reason backward, starting from the failure to discover the cause of the incorrect behavior. Programmers mostly observe the program's output to learn about parts of the program that have been executed. Common techniques for examining the program's execution include classical logging (e.g., print statements), online and offline debuggers. While online debuggers allow doing a controlled execution of the program through breakpoints and stepping operations, offline debuggers record program events in a log file or a trace for later inspecting the past state.

Most debuggers nowadays are, however, mostly operational, as they interact with developers based on operations like placing breakpoints on source code, step-by-step commands, or writing logical predicates in watchpoints. In order to find out the root cause of the bug, developers need to translate their hypothesis and questions about the program's output into debugging operations which provide information that may help them guess where the error resides. For example, to answer the question *"How did the program get to throw an exception?"*, an exception breakpoint could be placed to obtain a stack trace of the execution until the point where the exception is thrown. Unfortunately, if the developer chooses an operation that is irrelevant to the cause of the bug (e.g., placing a breakpoint on a line of code that halts the execution after the failure), the debugging tool will not help, and the debugging session needs to be restarted. That is even more problematic in concurrent programs due to the non-repeatability issue and the probe-effect [5].

Ko et al. proposed a debugging technique called *interrogative debugging* which can alleviate those issues by allowing developers to interactively ask *why did* or *why didn't* questions about the program output to reason about the program behavior [6, 7]. Their tool was designed for thread-based programs written in Java.

Inspired by their work, we propose in this paper complementing online debugging for actor-based programs with interrogative debugging. We design a set of questions and answers for actor-based programs based on the key concepts of the actor model, i.e., actors, messages, turns, and promises. We implement the questions and answers in an existing online debugger for actor-based programs called *Apgar*. Questions are presented to the developer in a dedicated view when a certain entity can be inspected (e.g., questions on actor state become available when an actor's execution is paused and the user selects that paused actor). The debugger then computes the answers by analyzing the trace of events recorded during the program execution. In our debugger, the analysis is based on events specified by the Kómpos protocol [10].

Like Ko et al. [6], we argue that it is crucial to ease the process of reasoning about the program output to find the root cause of concurrency bugs since developers often define correctness in terms of the output. This work is the first step towards a less operational and more interactive debugging experience for actor-based programs, which can shorten the debugging time.

2 Background

This section introduces the context of our work. It discusses the actor variant where we focus our research, terminology on concurrency bugs, and the system in which we prototyped our interrogative debugging approach.

2.1 Communicating Event-Loops (CEL) Model

The Communicating Event-Loops (CEL) concurrency model is a variant of the actor model first proposed for the E programming language [12]. This model has been adopted by languages such as AmbientTalk [19], Newspeak [2] and JavaScript [18]. In CEL, each actor is a container of objects, and it has exclusive access to its state. Each actor also contains a message queue or mailbox and a thread of control known as event-loop, which processes messages sequentially, one by one. A turn is defined as the processing of one message by an actor, that is, when the actor dequeues a message from its mailbox, delivers the message to its object and the message is executed to completion [12]. Within an actor, two objects can have direct reference to one another. Such references are called near references and their communication happens synchronously. Objects owned by different actors are called far references and communication between them is non-blocking through asynchronous messages.

Asynchronous messages return promises[1]. A promise is a placeholder for the result that is to be computed by the receiver actor of an asynchronous message. Promises can also receive asynchronous messages even if the result is not yet computed. Thus, a promise resolution can depend on another promise (returned from another asynchronous message); this is known as *promise chaining*. Messages sent to a promise are not delivered until the promise is resolved with a value or an exception. Once a promise is resolved, the asynchronous message is forwarded in order to the result value of the computation.

In this paper, we use the implementation of the CEL model in SOMns[2], a class-based dynamically-typed language for concurrency research that is based on Newspeak [2].

2.2 Concurrency Bugs in Actor-Based Programs

Actor-based programs are not free of concurrency bugs. In previous work [9], we studied concurrency bugs that have been observed in the literature. We now briefly introduce the terminology on bugs from that study which we employ in this paper.

We created a taxonomy that distinguishes two families of bugs. First, the category of *lack of progress issues* includes concurrency bugs related to the conditions in which actors are in a waiting state. In this category, we identified three subcategories:

Communication deadlock can be seen in languages that feature blocking operations, and refers to the condition in a system where two or more actors are blocked forever waiting for each other to send a message.

Behavioral deadlock is the condition in a system when two or more actors are not blocked but wait on each other, for a message to be able to progress, i.e., the message to complete the next step is never sent. This kind of bug is harder to identify than traditional deadlocks of thread-based programs because actors can still process messages from other actors.

Livelock is a condition similar to a deadlock in which two or more actors are not able to make progress, but they continuously change their state, that is, actors repeat the same interaction in response to the message sent by each other.

Second, the category of *message protocol violations* includes bugs related to the order in which actors process messages. In this category, we also identified three subcategories:

Message order violation is the condition in which the order of exchanging messages of two or more actors are not consistent with the intended behavior of the program.

[1] Some actor languages refer to promises as futures [19].
[2] https://github.com/smarr/SOMns

Bad message interleaving occurs when a message is processed between two messages which are intended to be processed one after the other.

Memory inconsistency occurs when different actors have inconsistent views of shared resources. The effects of the turn that modifies a conceptually shared resource may not be visible to other actors, which also alter the same resource.

In [9], we also identified bug patterns and observable behaviors for 24 bugs from literature and classified them according to our taxonomy. The results of that study shaped the design of the questions about the program's behavior we describe in the next section. In a nutshell, languages that use the CEL model such as SOMns do not suffer from communication deadlocks but can still suffer from the rest of categorized concurrency bugs. The debugging technique we present here aims to help developers to find the root cause of concurrency bugs that can be seen in programs that use the CEL model.

2.3 Apgar

Apgar [8] is a message-oriented online debugger for SOMns, which provides advanced debugging techniques for actor-based programs. More concretely, Apgar provides catalogs of breakpoints and stepping operations on the level of messages, actor state inspection through variables and mailbox, asynchronous stack traces, and visualizations to show happened-before relationship of messages. In this paper, we extend Apgar with support for interrogative debugging for actors.

Apgar frontend has been implemented as a plugin in the IntelliJ IDE. Apgar backend has been implemented as part of the SOMns interpreter, which is an AST-based interpreter built on the Truffle/GraalVM runtime [20]. The interactions between the debugger's frontend and backend happen through the Kómpos protocol, which is a concurrency-agnostic debugging protocol [10]. The protocol models the messages between frontend and backend, e.g., breakpoints, stepping, pause and resume commands. It also provides details about the execution of a concurrent program by trace events. Those events will be used to compute the answers to questions we devise for actor-based programs in Apgar. In Section 5 we will also detail extensions to the protocol that were needed for our approach.

3 Designing Questions and Answers for Debugging Actor-Based Programs

This section introduces the questions and answers we designed inspired by the interrogative debugging approach [6, 7], which we believe will help the developer to debug actor-based programs.

Table 1 shows an overview of the questions and answers we devised to be integrated in an online debugger for actor-based programs. We categorized the questions into three groups according to the main entities where the questions should get activated in an online debugger, i.e., actors, messages, and variables. The first two categories, actors and messages, are related to entities in the CEL model. The third category is related to the actor's state, which is stored in variables (holding primitive types and promises). We now describe the questions for each of these categories.

3.1 Actors

Questions in this category correspond to questions that can be asked to a paused actor. Here we consider questions related to the promises owned by the actor, turns processed, and messages sent and received. We consider that an actor is paused when it has been suspended due to a breakpoint, a stepping operation, or explicitly by the user with a pause command. A paused actor will not be able to process the next message in its mailbox, but it can still receive messages.

We designed two questions for the actors category that involves the *promise resolution state* of a paused actor.

- **Which promises are resolved?** The answer to this question returns all the promises that have been resolved inside the actor.
- **Which promises are not resolved?** The answer to this question returns all the promises that have not been resolved inside the actor.

The debugger's answer for both questions should include information about the properties of promises, e.g., its identifier, information about the creation turn, and the source coordinates in the editor of the place where the promise was created.

The goal of these questions is to help the developer understand why parts of the program that only execute once the promise is resolved, are or are not executed. The answer could help to find the root cause of concurrency bugs since we have observed that, for example, unresolved promises can cause lack of progress issues such as behavioral deadlocks.

Besides questions related to promise resolutions, we designed three more questions that should be enabled for a paused actor with respect to turns and messages:

- **Which messages were received?** The answer to this question returns all messages received by the actor from other actors.
- **Which turns were processed?** The answer to this question returns all turns processed by the actor.
- **Which messages were sent?** The answer to this question returns all messages sent by the actor.

The goal of these questions is to allow developers to inspect the order in which actors process messages. From reported studies, we identified that common bug patterns for message order violations are the incorrect execution order of actors and messages [9]. Hence, inspecting messages that a paused actor is receiving could be useful to observe, for example, messages that arrive in an unexpected order. Bug

Table 1. Overview of questions and answers when debugging a CEL actor-based program.

Question	Answer
1. Actors	
1.1 Which promises are resolved?	- List of resolved promises for the selected paused actor.
1.2 Which promises are not resolved?	- List of unresolved promises for the selected paused actor.
1.3 Which messages were received?	- List of messages received by the selected paused actor.
1.4 Which turns were processed?	- List of turns processed by the selected paused actor.
1.5 Which messages were sent?	- List of messages sent by the selected paused actor.
2. Messages	
2.1 Which is the turn of message X?	- Properties about the message corresponding to that turn; this includes message id, message selector, actor that process that turn, and source coordinate of the message corresponding to that turn.
3. Variables	
3.1 Why does variable X have that value?	- Timeline of assignments. For each assignment show the new value for the variable and the turn id where it happened.
Promises	
3.2 What is the resolution value of promise X?	- Resolution value for that promise, e.g., a string, integer, a promise, etc.
3.3 When was promise X resolved?	- Timeline of promise dependencies. For each promise selected in the timeline show properties about the message corresponding to that turn and about the resolution of the promise.

patterns for bad message interleavings include the processing of one message between two other messages that should be processed one after the other. Thus, we consider these questions essential to unveil unexpected interleavings of messages, which can lead to message order violations and bad message interleavings[3].

3.2 Messages

This category corresponds to questions related to the messages located in the mailbox of a paused actor. We designed the following question:

- **Which is the turn of message X?** The question returns information about the turn in which the message was sent.

The *turn information* consists of properties such as an identifier for the turn, an identifier for the actor that processed that turn, and the message that invoked that turn.

The goal of this question is to help developers understand the happened-before relationship of messages, i.e., which turn is responsible for a particular message send. This can be very helpful when inspecting the order in which actors process messages to discover the root cause of message protocol violations.

3.3 Variables

This category includes questions related to the state in an actor accessed via variables, which store objects in memory corresponding to primitive values or objects such as promises.

To access the state of a variable, we designed the following question:

- **Why does variable X have that value?** The answer to this question should return all assignments that changed the value of the variable. It should be shown chronologically in a timeline.

This question helps developers to understand why a certain variable has a certain value. Showing the *timeline of assignments for a variable and its state* helps understanding the history of updates for that variable in the program. We designed this question considering four cases that include local and global variables, which are detailed in Section 5.

3.3.1 Promises. Since promises are a relevant abstraction for synchronization and to model return values of asynchronous message sends, we consider it important to provide questions that help developers reason about the execution path that leads to a promise resolution. This can help to find the root cause of a bug, especially in those cases where promises are chained. We designed the following questions for promises:

- **What is the resolution value of promise X?** The question returns the value which resolves the promise.

The answer to this question can be either:

- the promise is resolved with a primitive value or object (which is not a promise).
- the promise is resolved with an error[4]
- the promise is resolved with another promise.

The information of the *resolution value* for a given promise will be helpful for the developer when inspecting the actor

[3]Our questions do not target heisenbugs, i.e., bugs caused by the probe-effect, in which the debugging tool itself introduces more non-determinism when observing the program execution.

[4]This promise is known as broken or ruin promise.

state. In the case when a promise is resolved with another promise, we propose a second question to learn about the dependencies for the promise:

- **When was promise X resolved?** The answer to this question consists of a timeline of promise dependencies.

The timeline shows chronologically the promises that need to be resolved before the selected promise can be resolved. Besides, the answer includes information about the message responsible for the creation of the promise, i.e., the asynchronous message sent that returned that promise. Also, we include information about the resolution of the selected promise in the timeline. In SOMns, promises can be created implicitly using the asynchronous send operator `<-:`, and explicitly by sending the `createPromisePair` message. For promises created explicitly by the developer, we include only the resolution information. The information about the *promise dependencies* can be very helpful for the developer to understand the history of promises that were resolved before the given promise, which can help to identify bugs such as message order violations.

4 A Proof-of-Concept Interrogative Debugger for SOMns Programs

This section describes our extension to Apgar (Section 2.3), an existing online debugger where we implemented the questions previously described. Our extension allows developers to choose a question generated about the program execution, and compute its answer. For this purpose, we added two new tabs in the user interface: a *Question view* and an *Answer view*. Figure 2 shows the graphical user interface of Apgar frontend with different debugging views in the IntelliJ IDE. In the next sections, we will explain how we integrated the different questions and answers presented in Table 1 in Apgar by means of a concrete application.

4.1 Running Example

Our running example consists of an order purchase application written in the SOMns programming language, which handles purchases of orders via a website. Figure 1 shows the actors and messages involved in the application when the buy message, which initiates the purchase of items in a shopping cart, is sent. The application consist of seven actors: Platform, Customer, Website, Store, Account, Shipper, and Database.

Before an order is placed, the Website actor verifies three services which are represented by three actors in the program:

- whether the requested items are still in stock (by sending the `productInStock` message to the Store actor),
- whether the customer has provided valid payment information (by sending the `checkCredit` message to the Account actor) and,

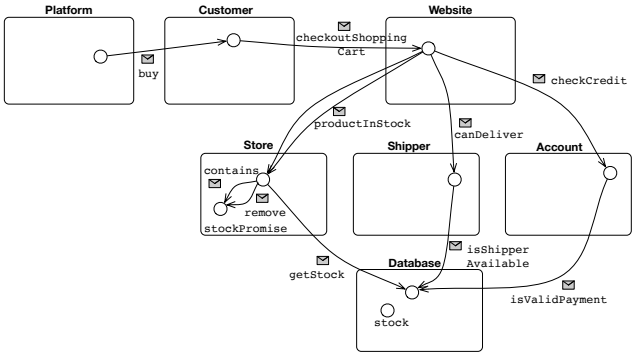


Figure 1. Overview of the actors and messages in the order purchase application.

- whether a shipper is available to ship the order in time (by sending the `canDeliver` message to the Shipper actor)

Through this section, we consider a version of the order purchase application that does not terminate because it suffers from a behavioral deadlock (see Listing 1), and we will debug it with our approach[5]. To collect the answer to the three services, a promise group is used (Line 76).

4.2 Actors

As shown in Table 1 we designed five questions related to paused actors. Those questions appear in the Questions view when the developer selects a paused actor in the Actors view (see Figure 2 (3)). This section describes the answers to the first two questions. The answer to the questions related to turns and messages have dedicated views, i.e., mailbox, turns and sentbox which are not yet linked to the Questions view in our current prototype.

In a debugging session, the developer can check the body of the callback (Line 77) with a breakpoint, e.g., in a line breakpoint or a message breakpoint such as promise resolution. However, there could be an unresolved promise in the promise group (Line 76), indicating a program misbehavior but the program would not be suspended. Developers can inspect the resolved and the unresolved promises for the Website actor upon suspension. For example, when the Website actor pauses due to a message sender breakpoint on Line 73, the questions *Which promises are resolved?* and *Which promises are not resolved?* are visualized in the Questions view. The developer can visualize the answer to a question by right clicking on the question and selecting the menu option "Answer". Figure 3 shows the answers for these questions.

In this case, the developer can observe in Figure 3 (b) that promise `productsPP` (i.e., promise with id 3221225474) is

[5]The complete implementation of the application is available also at https://github.com/ctrlpz/agere-sample-program/blob/master/OrderPurchaseBD.ns

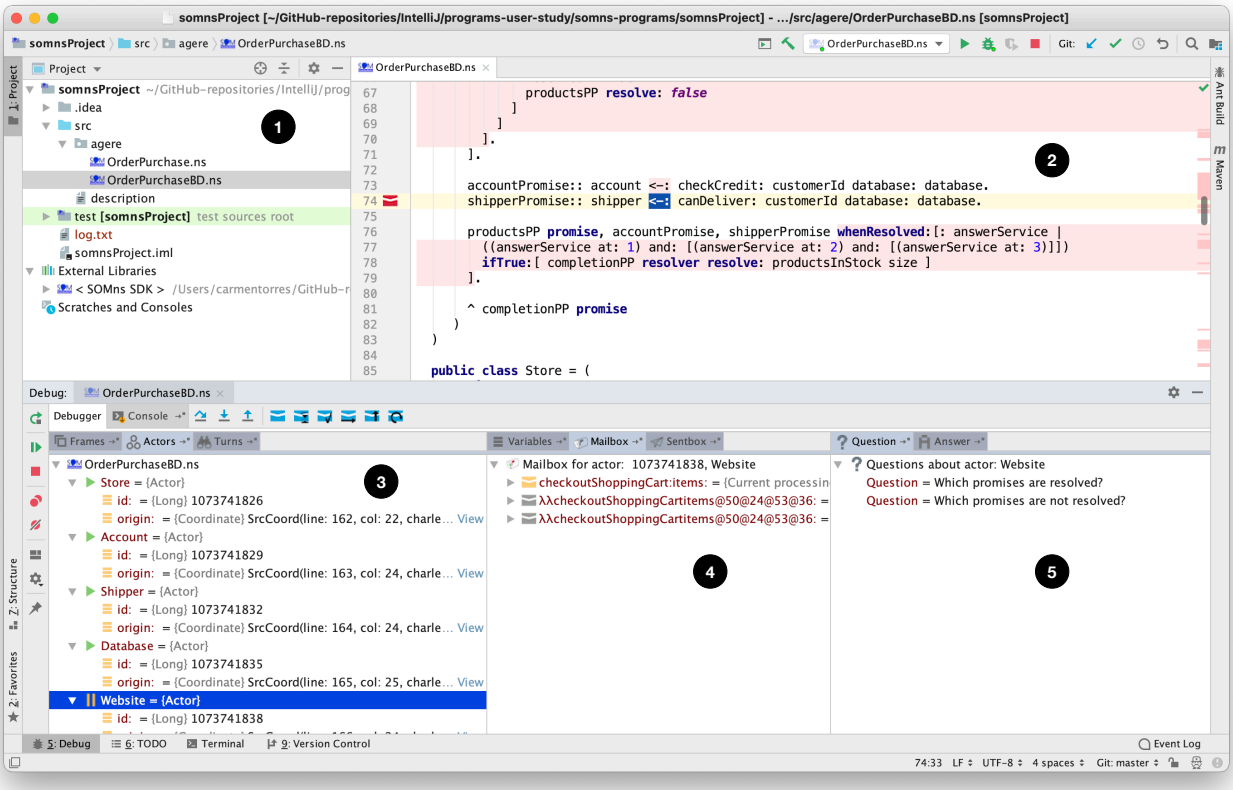


Figure 2. Graphical user interface of Apgar debugger with interrogative features. (1) The developer opens the program file in the SOMns project. (2) The developer adds a breakpoint where to pause the program's execution. The Debugger tab consists of three main parts: (3) The Actors view shows information about the actors of the program which are in run or pause state. (4) The Mailbox view shows the messages received by the selected paused actor. (5) The Question view shows the possible questions that can be asked to the debugger, in this case, about the paused actor.

unresolved (Line 43), which indicates that the root cause of the problem could be located when verifying if the products are in stock by the Store actor (Line 52).

4.3 Messages

To answer question 2.1 of Table 1 developers need to select the actor and the message in the mailbox that they want to know the turn. Figure 4 shows the answer for the `checkoutShoppingCart` message, considering the debugging session visualized in Figure 2. The actor id shown corresponds to the Customer actor, and buy is the message corresponding to the turn, which is sent by the Platform actor to the Customer actor.

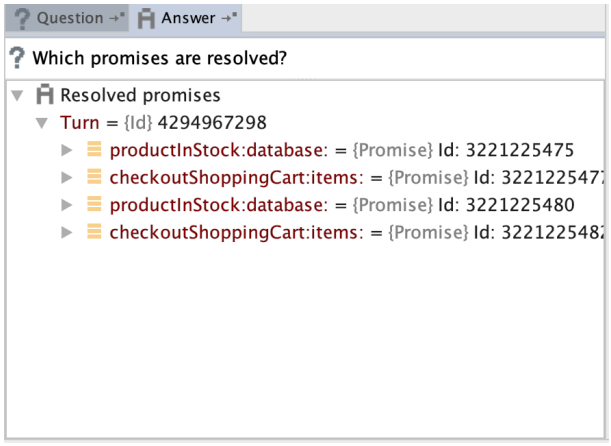
4.4 Variables

In this section, we show some of the questions and answers related to variables by means of the running example.

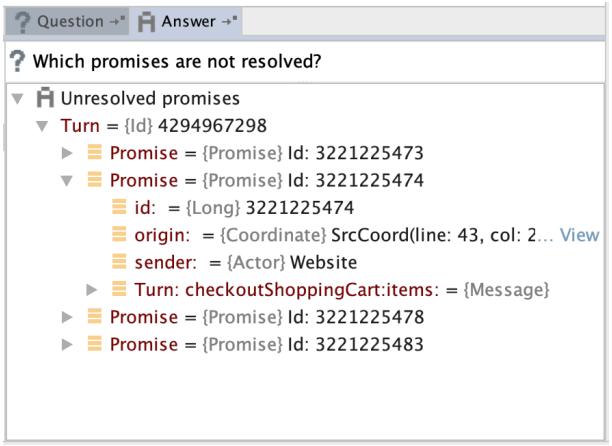
The answer to question 3.1 of Table 1 is a timeline of assignments that changed the value of the selected variable. The assignments are sorted in chronological order and grouped by turn. Thus, earlier assignments will be higher up in the timeline than later assignments. Clicking on an assignment or turn in the timeline will allow the developer to navigate to the source code coordinate of the assignment in the editor, or the message responsible for the turn, respectively.

The implementation class of the Store actor contains a global variable named `counter`. This variable is updated every time the `productInStock` method is executed. Setting a line breakpoint on Line 97 and selecting the counter variable will allow the developer to select the question *Why does variable counter have that value?*. Figure 5 shows the answer: two assignments have happened for the global variable `counter` at this point. The turn id corresponds to the id of the message that is being executed and where the assignment happened.

4.4.1 Promises. Questions 3.2 and 3.3 of Table 1 are enabled when the developer selects a promise in the Variables



(a). Resolved promises for Website actor.



(b). Unresolved promises for Website actor.

Figure 3. Answers for questions about promises owned by a paused actor.

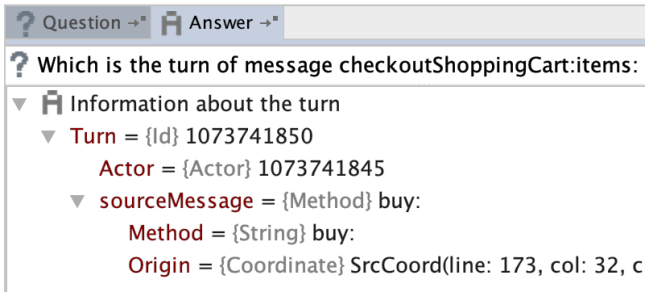


Figure 4. Information about a turn corresponding to message `checkoutShoppingCart` in the mailbox of the Website actor.

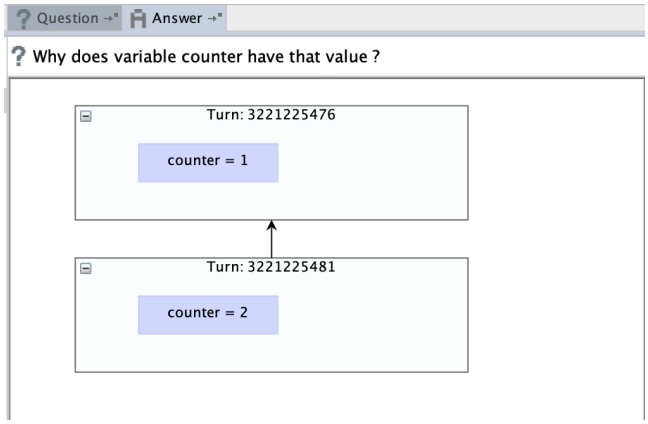


Figure 5. Timeline of assignments for a global variable.

view. In the context of our order purchase program, the developer can add a line breakpoint on Line 74 of Listing 1 (or a message sender breakpoint) and select the `accountPromise` local variable in the Variables view. Consequently, the Question view displays the questions related to the `accountPromise`.

Figure 6 shows the answer to the question *What is the resolution value of promise* `accountPromise`*?*. In this case, the answer consists of another promise because the promise is resolved with the promise resulting from the `isValidPayment` message in the Account actor (see Line 115).

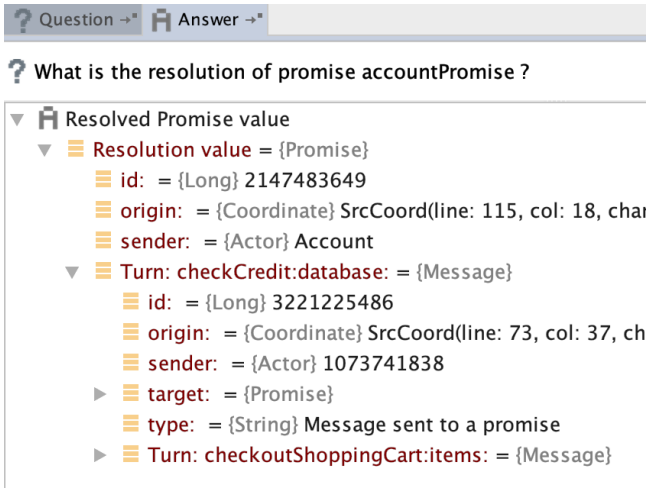


Figure 6. Resolution value for a promise.

The developer would also want to know why that promise obtained that resolution value. This can be answered by selecting question 3.3 of Table 1. Figure 7 shows the answer which consists of the timeline of all promise dependencies corresponding to the `accountPromise` promise. Each rectangle in the timeline shows the method selector corresponding to a message that returns a promise and its id. The rectangles denoted with the Argument word show promise ids of promises passed as an argument in a message. On the left

panel, information about the selected promise in the timeline (denoted with a green border) is given. More concretely, information about the turn where it was created, and its resolution value is shown. In this case, the selected promise is the one for the `new:website:` message sent to the Platform actor and the resolution value is a far reference to the Customer actor (see Line 171).

5 Implementation

This section provides relevant details about the implementation of our interrogative debugging approach for actor-based programs in Apgar [6]. We focus the discussion on how to obtain the answers for the different categories of questions. To compute the answers, the debugger analyses the information obtained from the events generated by the Kómpos protocol regarding the different concurrency concepts, e.g., actors, turns, promises, messages, and send operations [10]. We also extended the Kómpos protocol to capture more information about the program execution. In particular, we added two events: `MessageReception` and `Assignment`. In addition, we needed to extend the information captured in the `SendOperation` event.

5.1 Actors

The information about promises that is needed to answer questions 1.1 and 1.2 of Table 1 was extracted from the passive entities provided in the trace. For every promise event that the debugger frontend receives, our implementation checks which promise was created for the selected actor and if it was resolved or not. To know if a promise was resolved, we check if the send operation event corresponding to promise resolution exists for the specified promise in the trace. Finally, we visualize the resolved and unresolved promises for the selected paused actor in the Actor view.

In our current debugger frontend, the Question view does not offer questions 1.3, 1.4, and 1.5 when an actor is selected in the Actors view. However, we have implemented the analysis to provide answers to these questions based on the events of the Kómpos protocol, which we detail in what follows.

To know which *messages were received* (question 1.3), we added the trace event `MessageReception` to the protocol, which records the actor id (that is, the receiver actor of the message) and the message id. In the debugger backend, meaning, the SOMns interpreter, we record a message reception at two points, first, when the actor appends the received message in its mailbox, and second when the actor is about to process the messages in its mailbox. We need to record both points because the actor can be paused and still receive messages. Thus, the debugger keeps a list of messages in the order the messages are received by the actor, in other words, in the order of arrival [8]. The list of received messages is visualized in the frontend in the Mailbox view (see Figure 2(4)).

To know which *turns were processed* by an actor (question 1.4), we use the information of the trace event corresponding to dynamic scope `ScopeStart`, that is, we use the information of the message id corresponding to the turn and actor id of the actor that processes the message. We visualize the turns processed by an actor in the Turns view in Apgar, which is a graph visualization in a space-time diagram showing the happened-before relationship of messages [8].

Finally, the information about which *messages were sent* by the selected paused actor (question 1.5) is extracted from the `SendOperation` event. When reading this event from the trace, we obtain the turn where the message (which can be sent to a far reference or to a promise) was sent. Thus, we keep all send operations (or messages) sent in a turn. We show the information of messages sent by turn in the Sentbox view [8]. The list of messages sent is updated when the developer clicks a turn in the graph shown in the Turns view.

5.2 Messages

To get the information about the *turn* in which a message was sent (question 2.1), we parsed in the debugger frontend the trace event that indicates the beginning and end of the turn. In particular, we saved each event per actor and reordered the events because it can happen that the buffers where the events are written may be out of chronological order from the perspective of an actor because an actor can be executed on different threads over time [8]. We obtain the properties of the message sent corresponding to that turn from the trace event `SendOperation`, such as the id, its selector, its source coordinate. The actor that executes that turn corresponds to the sender actor of the message that was sent in the turn, i.e., we obtain the turn actor from the creation activity of the `SendOperation` event of the message sent in the turn. We needed to extend the `SendOperation` event with the message selector and the source coordinate because they were not provided by the original implementation of the Kómpos protocol in SOMns.

5.3 Variables

For obtaining information related to *variables assignments* (question 3.1), we added the `Assignment` event in the trace. This event consists of a variable id, a location, and the new value to which the variable is updated. We use as variable id the source coordinate corresponding to that variable declaration. The location refers to the place in the code where the assignment happens. Thus, the debugger backend records the variable information in the trace every time a variable update occurs, i.e., when AST node writing for local variables and global variables occurs. In the debugger frontend, when parsing the `Assignment` event from the trace, we resolve the

[6] The prototype we present here was implemented with version 3 of Apgar. It remains as future work to upgrade the implementation to its last version.

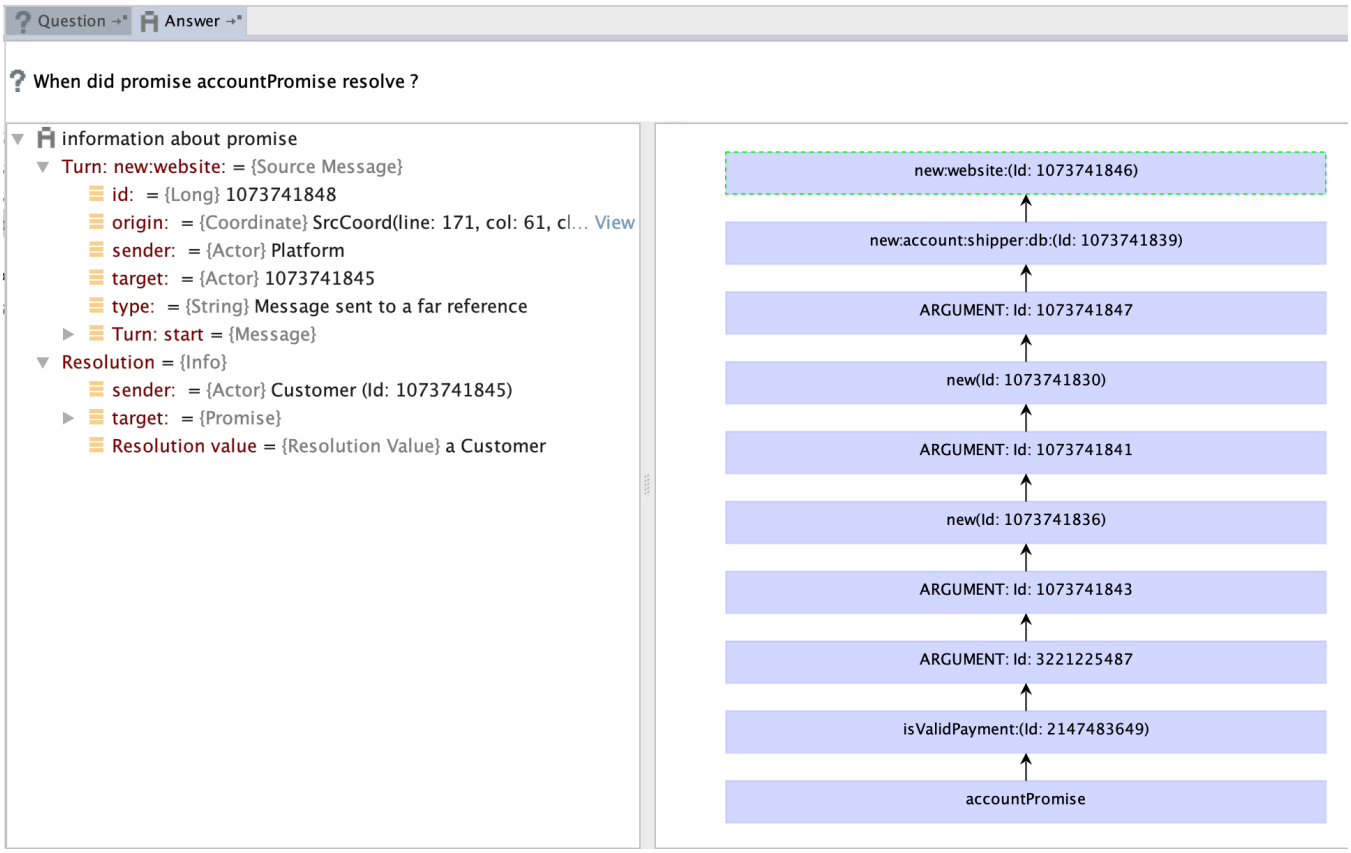


Figure 7. Timeline of promise dependencies and information about the selected promise.

turn for this event similarly as explained for the messages in the previous section.

We considered four cases to answer the question of why a variable has a certain value[7]. These cases are relevant for CEL languages because they are based on general concepts, e.g., turns and actors.

- When a variable is local and is declared in one turn: in this case variables are introduced during a method invocation in which the assignments happen in the same turn.
- When the variable is global: in this case variables state can be updated in multiple turns in an actor.
- When the variable is in a loop: this case refers to different assignments on the same local variable in a loop.
- When the variable can be updated in different turns: this case can be seen in SOMNs programs in particular for promise callbacks defined by the `whenResolved:` message, because the operations inside a `whenResolved:` block can still access the local variables outside the block.

[7] At the moment, our prototype supports case 1 and 2. Case 3 and 4 remain as future work.

To obtain the information related to the *resolution of the promise* (question 3.2) we extended the `SendOperation` event with the promise resolution value. In particular, this is needed for the cases when a promise is resolved with a value or an error. When a promise is resolved with another promise, the debugger backend uses the entity id field of the `SendOperation` event to save the promise id that is used to resolve another promise.

Finally, to obtain the information about *promise dependencies* for a selected promise until its resolution (question 3.3) we added support in the debugger frontend for finding promise dependencies using the trace information, in particular we used `SendOperation` and `PassiveEntityCreation` events. Our implementation searches for chained promises, and it stops when in the chain a promise is resolved with a value that is not a promise. We identified three cases of dependencies:

- Sibling dependency: when the selected promise depends on the resolution of another promise created in the same turn.
- Parent dependency: when the selected promise depends on a promise that needs to be resolved in a previous turn. For example, in SOMNs this happens when a promise is created in the body of a `whenResolved`

block. Here there is a dependency to the promise that needs to be resolved before the callback is executed.
- Child dependency: when the selected promise depends on the resolution of another promise that is created inside the turn of the message that returns the selected promise.

As we can observe, the child dependency is different from the sibling dependency in that the dependent promise is resolved in another turn. Parent dependencies refer to promises created inside callbacks. Listing 1 shows a sibling dependency of `buyPromise` on the Line 173 which depends on promise `customer` on Line 171. A parent dependency can be observed in the promise created on Line 100, which depends on the promise in Line 98. A child dependency can be observed in `accountPromise` on Line 73 which depends on the promise on Line 115.

6 Related Work

To the best of our knowledge, our approach, is the first proposal to interactively debug actor-based programs using interrogative debugging features. This section summarizes visualization approaches from the state of the art for actor-based programs in two categories, standalone tools and as part of a debugger.

6.1 Visualization in Standalone Tools

Miriyala et al. [13] proposed the use of *predicate transition nets* for visualizing actors' execution in the classic actor model, in particular actor's behavior and sent messages. Coscas et al. [4] presented a similar approach in which the predicate transition nets are used to simulate actors' execution in a step-by-step mode.

Alimadadi et al. [1] proposed *visual graphs* based on dynamic program analysis for promises in asynchronous JavaScript programs. Their goal is to give developers a tool to understand the execution flow of promises and also identify anti-patterns, e.g., unresolved promises. Similarly, Sun et al. [17] proposed asynchronous graphs to visualize the asynchronous execution flow of a Node.js application.

Clark et al. [3] proposed *filmstrips pattern* visualization and implemented it for an actor language that follows the semantics of the classic actor model. A filmstrip is denoted as a sequence of snapshots of objects and relationships of the program described in state transitions derived from system operation calls.

6.2 Visualization as Part of a Debugger

The Causeway debugger visualizes asynchronous messages sent in different views based on a trace [16]. It provides a *process order* view that shows all asynchronous messages executed for each actor in chronological order. Besides, it offers a *message order* view that shows causal messages for a message sent, i.e., other messages that have been executed before the message was sent and provoked the sending of the message we want to debug.

Shibanai et al. [15] proposed a debugger for Akka programs that visualizes message dependencies through *sequence diagrams*. The authors mention that their approach provides an interactive aid for showing the arrival order of messages and it enables developers to inspect past states.

The IDeA debugger [11] uses *an immersive visualization in 3D* for debugging an Akka program at the message level. The authors argue that representing actors as geometric entities in 3D space requires less effort than a 2D environment because it involves only moving the eye-head-bearing.

As we can observe, the visualizations that have been created as part of a debugger for actors have been focused mainly on representing the happened-before relationship of messages. This feature has inspired the design of some of our questions, e.g., 1.3, 1.4, 1.5, and 2.1 (see Table 1). Nevertheless, the mentioned approaches are limited to represent other concurrency concepts in the context of actors, such as unresolved promises and promises dependencies. Some standalone tools have considered these concepts in their features, but they are not included in a debugging session. The questions and answers we presented in this paper allow developers to inspect variables, messages, and turns for a paused actor in combination with online debugging features, i.e., using breakpoints and stepping operations.

7 Conclusion and Future Work

In this paper, we investigate how to make online debugging of actor-based programs more interactive by means of interrogative debugging. We focus on programs written in the Communicating Event-Loops model, which can still suffer from concurrency bugs.

We designed a set of questions and answers that developers can choose from when debugging actor-based programs related to the main features of the concurrency model: actors, messages, turns and promises. We implemented our approach by extending the Apgar online debugger for SOMns. To this end, we extended the Kómpos protocol with trace events to provide the necessary debugging information to answer the questions.

The combination of online and interrogative debugging features presented in this paper aims to fill the gap between the developers' interpretation of a failure and speculations of where the root cause of the bug is. Hence, we believe such a debugging approach could shorten the debugging time. And even more, it could avoid that developers reason about non-related paths to the root cause of the bug, e.g., using the timelines of variable assignments and promise dependencies.

As future work, we think it will be valuable to evaluate our proposal of questions and answers through a user study and compare it with other debugging tools for actor-based programs in terms of time and usability.

Acknowledgments

We would like to thank Stefan Marr for his feedback on this work and his technical support about SOMNs. We would like to thank Kevin De Porre and the anonymous reviewers for their comments and feedback on the text.

A Order Purchase Application

```
(* Written by Carmen Torres Lopez
* (NOTE: This version contains a behavioral deadlock)
* This implementation has been inspired by other actor language implementations
* such as E and AmbientTalk.
* - Stanley, T., Close, T., & Miller, M. S. (2009). Causeway: A message-oriented distributed debugger.
* - Gonzalez Boix, E., Noguera, C., & De Meuter, W. (2014). Distributed debugging for mobile networks.
* Journal of Systems and Software, 90, 76-90.
*)

class OrderPurchaseBD usingPlatform: platform = Value (
| private actors = platform actors.
  private Vector = platform kernel Vector.
  private TransferArray = platform kernel TransferArray.
| )(

  public class Customer new: customerId website: web = (
    | private customerId = customerId.
      private website = web.
    | )(

    public buy: items = (
      | checkoutPromise |
      checkoutPromise:: website <-: checkoutShoppingCart: customerId items: items.

      ^ checkoutPromise whenResolved: [:result |
          result > 1
          ifTrue:[('- The order has been placed for '+ result + ' products.') println.]
          ifFalse:[('- The order has been placed for ' + result + ' product.') println.].
      ]
    )
  )

  public class Website new: store account: account shipper: shipper db: database = (
    | private store = store.
      private account = account.
      private shipper = shipper.
      private database = database.
    |)(

    public checkoutShoppingCart: customerId items: items = (
      | shoppingCart completionPP accountPromise shipperPromise productsPP productsInStock resolved |
      completionPP:: actors createPromisePair.
      productsPP:: actors createPromisePair.
      productsInStock:: Vector new.
      resolved:: false.

      shoppingCart:: items.
      ('- You will buy '+ (shoppingCart size) + ' products. ') println.

      shoppingCart do:[:product |
      | existPromise |
      existPromise:: store <-: productInStock: product database: database.
      existPromise whenResolved:[: available |
        available ifTrue:[
          productsInStock append: product.
          productsInStock size = shoppingCart size
            ifTrue:[
              resolved ifFalse:[
                resolved:: true.
                productsPP resolve: true
              ]
            ]
          ]
          ifFalse:[
            resolved ifFalse:[
              resolved:: true.
              productsPP resolve: false
            ]
          ]
        ].
      ].

      accountPromise:: account <-: checkCredit: customerId database: database.
      shipperPromise:: shipper <-: canDeliver: customerId database: database.

      productsPP promise, accountPromise, shipperPromise whenResolved:[: answerService |
      ((answerService at: 1) and: [(answerService at: 2) and: [(answerService at: 3)]])
      ifTrue:[ completionPP resolver resolve: productsInStock size ]
      ].

    ^ completionPP promise
    )
  )

  public class Store = (
    | private counter ::= 0.
    |)(

    public productInStock: item database: database = (
      | stockPromise existPP |

      existPP:: actors createPromisePair.

      counter:: counter + 1.


      stockPromise:: database <-: getStock.
      (stockPromise <-: contains: item) whenResolved:[:exist |
          exist ifTrue:[
            stockPromise <-: remove: item.
          ].



        (* existPP resolve: exist.*)
        ].

      ^ existPP promise
    )
```

```
110   )
111
112   public class Account = ()(
113
114     public checkCredit: customerId database: database
        = (
115       ^ database <-: isValidPayment: customerId
116     )
117   )
118
119   public class Shipper = ()(
120
121     public canDeliver: customerId database: database =
        (
122       ^ database <-: isShipperAvailable: customerId
123     )
124   )
125
126   public class Database = (
127     | private stock = init. |
128   )(
129
130     private init = (
131       | s |
132         s:: Vector new.
133         s append: 'hdd'.
134         s append: 'ipad'.
135         s append: 'phone'.
136         s append: 'screen'.
137         s append: 'laptop'.
138       ^ s
139     )
140
141     public getStock = (
142       ^ stock
143     )
144
145     public isValidPayment: customerId = (
146       ^ true
147     )
148
149     public isShipperAvailable: customerId = (
150       ^ true
151     )
152
153   )
154
155    public main: args = (
156      | customer store account shipper website database
        items buyPromise timeout completionPP |
157      timeout:: 3000.
158
159      completionPP:: actors createPromisePair.
160
161      '[ORDER PURCHASE APPLICATION] Starting...\n'
        println.
162      items:: TransferArray new: 2.
163      items at: 1 put: 'phone'.
164      items at: 2 put: 'laptop'.
165
166      store:: (actors createActorFromValue: Store) <-:
        new.
167      account:: (actors createActorFromValue: Account)
         <-: new.
168      shipper:: (actors createActorFromValue: Shipper)
         <-: new.
169      database:: (actors createActorFromValue: Database
        ) <-: new.
170      website:: (actors createActorFromValue: Website)
        <-: new: store account: account shipper: shipper
        db: database.
171      customer:: (actors createActorFromValue: Customer
        ) <-: new: 'Joe' website: website.
172
173      buyPromise:: customer <-: buy: items.
174
175      (* actors after: timeout do: [
176       'Program exit due to TIMEOUT' println.
177        completionPP resolve: 1.
178      ]. *)
179
180      completionPP resolve: buyPromise.
181
182      ^ completionPP promise whenResolved: [:result |
183            '\n[ORDER PURCHASE APPLICATION] Ending.'
        println.
184      ]
185    )
186 )
```

Listing 1. Order purchase application in SOMns.

References

[1] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding Broken Promises in Asynchronous JavaScript Programs. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 162 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276532

[2] Gilad Bracha. 2009. *Newspeak programming language draft specification version 0.06.* Technical Report. Technical report, Ministry of Truth.

[3] Tony Clark, Balbir Barn, Vinay Kulkarni, and Souvik Barat. 2019. Making Sense of Actor Behaviour: An Algebraic Filmstrip Pattern and Its Implementation. In *Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Software Engineering Conference)* (Pune, India) *(ISEC'19)*. Association for Computing Machinery, New York, NY, USA, Article 13, 10 pages. https://doi.org/10.1145/3299771.3299783

[4] Patrick Coscas, Gilles Fouquier, and Agnes Lanusse. 1995. Modelling Actor Programs using Predicate/Transition Nets. In *Proceedings Euromicro Workshop on Parallel and Distributed Processing*. 194–200. https://doi.org/10.1109/EMPDP.1995.389129

[5] Jason Gait. 1986. A Probe Effect in Concurrent Programs. *Softw. Pract. Exp.* 16, 3 (1986), 225–233. https://doi.org/10.1002/spe.4380160304

[6] Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 301–310. https://doi.org/10.1145/1368088.1368130

[7] Andrew J. Ko and Brad A. Myers. 2010. Extracting and Answering Why and Why Not Questions about Java Program Output. *ACM Trans. Softw. Eng. Methodol.* 20, 2, Article 4 (Sept. 2010), 36 pages. https://doi.org/10.1145/1824760.1824761

[8] Carmen Torres Lopez. 2021. *Advanced Debugging Techniques to Handle Concurrency Bugs in Actor-based Applications.* Ph.D. Dissertation. Vrije Universiteit Brussel.

[9] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs. In *Programming with Actors - State-of-the-Art and Research Perspectives*, Alessandro Ricci and Philipp Haller (Eds.). Lecture Notes in Computer Science, Vol. 10789. Springer, 155–185. https://doi.org/10.1007/978-3-030-00302-9_6

[10] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools. *CoRR* abs/1706.00363 (2017). arXiv:1706.00363 http://arxiv.org/abs/1706.00363

[11] Aman Shankar Mathur, Burcu Kulahcioglu Ozkan, and Rupak Majumdar. 2018. IDeA: An Immersive Debugger for Actors. In *Proceedings of*

the 17th ACM SIGPLAN International Workshop on Erlang (St. Louis, MO, USA) *(Erlang 2018)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3239332.3242762

[12] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. 2005. Concurrency Among Strangers. In *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3705)*, Rocco De Nicola and Davide Sangiorgi (Eds.). Springer, 195–229. https://doi.org/10.1007/11580850_12

[13] Shakuntala Miriyala, Gul Agha, and Yamina Sami. 1992. Visualizing actor programs using predicate transition nets. *Journal of Visual Languages & Computing* 3, 2 (1992), 195–220. https://doi.org/10.1016/1045-926X(92)90015-E

[14] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2016. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2016), 83–110. http://dblp.uni-trier.de/db/journals/sqj/sqj25.html#PerscheidSTH17

[15] Kazuhiro Shibanai and Takuo Watanabe. 2017. Actoverse: A Reversible Debugger for Actors. (2017). https://doi.org/10.1145/3141834.3141840

[16] Terry Stanley, Tyler Close, and Mark Miller. 2009. *Causeway: A message-oriented distributed debugger*. Technical Report. HP Labs. 1–15 pages.

[17] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. 2019. Reasoning about the Node.js Event Loop using Async Graphs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 61–72. https://doi.org/10.1109/CGO.2019.8661173

[18] Stefan Tilkov and Steve Vinoski. 2010. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* 14, 6 (Nov 2010), 80–83. https://doi.org/10.1109/MIC.2010.145

[19] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures* 40, 3-4 (2014), 112–136. https://doi.org/10.1016/j.cl.2014.05.002

[20] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587