iDeA: An Immersive Debugger for Actors

Aman Shankar Mathur
MPI-SWS
Germany
mathur@mpi-sws.org

Burcu Kulahcioglu Ozkan
MPI-SWS
Germany
burcu@mpi-sws.org

Rupak Majumdar
MPI-SWS
Germany
rupak@mpi-sws.org

Abstract

We present iDeA, an immersive user interface for debugging concurrent actor programs communicating through asynchronous message passing. iDeA is based on the hypothesis that debugging and understanding actor programs is a cognitive task which can be greatly facilitated by the visualization and interaction capabilities of modern *immersive* environments. The fundamental abstraction for visualization in iDeA is a *concurrent trace*: a partially ordered sequence of asynchronous messages exchanged in the execution. iDeA provides a 3D interface in virtual reality for users to visualize and manipulate program traces: users can set breakpoints, query actor state, step through traces forward and backward, and perform causal history of messages in a trace.

While the modularity of iDeA enables debugging any actor program provided that the program events are collected and communicated to the visualization end, our implementation of iDeA targets actor programs written in Akka framework in Scala.

CCS Concepts • **Software and its engineering → Software testing and debugging**; • **Human-centered computing** → *Visualization*; • **Computing methodologies** → *Distributed computing methodologies*;

Keywords actor programs, debugging, concurrency, visualization, virtual reality

ACM Reference Format:
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 (Erlang '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3239332.3242762

Erlang '18, September 29, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5824-8/18/09…$15.00
https://doi.org/10.1145/3239332.3242762

1 Introduction

Concurrent components interacting through asynchronous message passing are the basis for many modern applications—from large-scale servers to responsive user apps on the web or mobile platforms to networked embedded systems. The *actor model of programming* [2, 21] is a popular way to structure such applications. This model provides a high-level concurrency abstraction consisting of lightweight computation units called *actors*, which operate on an isolated, encapsulated state and communicate with each other by sending and receiving messages asynchronously. An application-level scheduler co-ordinates the execution by picking an actor which has an outstanding message waiting to be processed and executing it serially. Executing the message receipt results in an update of the actor's local state as well as further messages sent to the actors for future processing. The high-level concurrency abstraction relieves the programmers from explicitly considering low-level aspects of concurrency such as threading, locks, or shared global state. The popularity of the model is shown by its widespread availability, either as a language construct (e.g., Erlang, P, Scratch) or as a library on top of an asynchronous programming interface (e.g., Akka for Scala and Java, Asynchronous Agents in C#, and many others).

In spite of the higher-level concurrency abstraction, program comprehension and debugging for actor programs is not easy. While each actor processes its messages serially, the delivery of messages across different actors is concurrent. This leads to complex concurrent chains of asynchronous messages. The behavior of the system depends on the specific processing order and a subtle change in the ordering can lead to unexpected program behaviors. At the same time, modern IDEs provide rudimentary support for understanding and debugging concurrent programs at this level of abstraction. Most debugging frameworks only provide functionality appropriate for sequential or low-level concurrent code (step through code, query state or variables or the call stack, set statement-level breakpoints) or a thin layer on top (intercept messages or visualize message sequences in 2D [10, 13, 14, 37]). Consequently, the main tool for debugging today is programmer-inserted log messages and understanding concurrency from a serial log of execution.

In this paper, we present iDeA, an *i*mmersive *De*bugger for *A*ctor programs. iDeA is an immersive environment for the HTC Vive virtual reality device, and provides a simple, visual user interface for debugging actor programs written in Akka,

an actor framework for the Scala language.[1] The architecture of iDeA has two parts. The first part intercepts scheduling choices by replacing the underlying scheduler and exports the event for visualization and debugging. The second part performs trace visualization and direct manipulation through debug actions.

The two key concepts underlying iDeA are *debugging at the message level* and *immersive visualization and animation*.

Message-level Debugging. iDeA is intended for debugging situations where the serial behavior of each actor (the response to a single received request) is understood, but the *global* behavior of many messages is not understood. This is the case for many concurrent protocols where the programmer must explore and understand the set of global traces of messages. Consequently, the visual interface in iDeA presents actors as 3D visual entities arranged in 3D space, messages as arrows between objects which fade over time, and provides capabilities for playing and manipulating sequences of messages.[2] The notion of an atomic step is not statement-level execution, but the atomic handling of a message by an actor. Similarly, state of the system is defined only at schedule points; we do not provide statement-level state changes for each actor.

Thus, iDeA enables a high-level interaction with the executing program at the level of the concurrency model. Instead of stepping over a single statement or watching the value of a local variable between two statements in a message handler, users step over processing a single message (whose execution is semantically atomic in an actor program execution). Users can set breakpoints on certain actors so that execution breaks before these actors receive the next message. At any point of execution, programmers can point to an actor and query its state and the state of its message queue. Analogous to watch variables in conventional debuggers, evolution of actor state is animated along with the trace. Further, the user can "time travel" along the trace to explore the timing behavior of the execution.

While many debug actions replay the current execution deterministically, leaving the scheduling actions arbitrary, iDeA users can also influence the scheduling choice online by picking a specific actor to be scheduled in the next step. Events are executed in "logical time": we do not consider real-time behavior.

Working at the level of the concurrency model allows us to support domain-specific debugging aids backed by program analysis. As an example of a domain-specific program analysis and its visualization, we have implemented *causality analysis* as a built-in debug feature in iDeA. In causality analysis, the user can annotate a set of actors and then trace how other actors are causally related to the source actors through the exchange of messages.

[1] Akka is available from http://akka.io.

[2] For some actor programs, such as in air traffic control or IoT, there is a natural mapping from program-level actors to 3D space.

While we provide text logs for the current trace within the interface, we *do not* provide a capability to directly manipulate the code at the textual level within iDeA because current VR environments do not provide enough resolution or interaction modalities for convenient text editing.

Immersive Visualization Environment. The visual metaphor in iDeA represents actors as geometric entities in 3D space and messages as dynamically constructed arrows between these entities. Thus, traces are animations of arrow sequences with manipulation capabilities to focus attention on various aspects of the execution.

The visual interface of iDeA is predicated on the assumption that *trace animation* in an *immersive environment* can be more effective for debugging and understanding large-scale concurrent actor systems than a traditional 2D-screen interface. We justify our assumption based on recent research in visual perception experiments in immersive environments as well as on previous research on non-immersive trace visualization for program comprehension.

Recent research in visual search [42] shows that visual search can be performed in reasonable time in an immersive environment with over a thousand distractors; comparable experiments in 2D environments scale only to less than hundred distractors. People quickly learn positions of objects in 3D space and spatial memory can guide visual search in an immersive setting [26, 34]. The importance of peripheral vision in perceptual processing has been studied extensively [47, 48]. In an immersive environment, the transference of attention from a focus area to the periphery can be seamless (and simply involves moving the eye/head/bearing), whereas in a typical 2D environment requires an explicit interaction.

iDeA assumes that evolution of state in a concurrent program can be effectively explored through animation. While animation may not be effective in every visualization situation, there is prior evidence that it can be effective in concurrent program comprehension (see, e.g., [11, 50] and the references therein). Animation has been used successfully in understanding evolution of developer workflows in the code_swarm tool [45]. Developer workflows have features very similar to actor-based programs. In the field of high-performance computing, (2D) animation has been used effectively to understand message-passing behavior of large-scale parallel programs [20, 24, 25, 50]. In contrast to many of these systems, where the focus was simply to visualize a trace, iDeA goes further in allowing interactive exploration of the trace in the context of program debugging.

The key to effective exploration of traces for large programs is the ability to *suppress* information. In iDeA, we provide two different mechanisms to focus attention. First, we provide an *area of focus* where the user can drop selected actors. The area of focus exploits the natural visual focus of the human perceptual system, while keeping other actors in peripheral vision. Second, we provide user interaction

modes to suppress messages in untracked actors. Together with time travel, this allows effective navigation of traces.

Contributions. We summarize our contributions. iDEA[3] is the first immersive debugging system for actor-based, asynchronous message-passing programs. It provides an integrated environment for post mortem and online debugging of programs, and provides functionality for:

- Recording and deterministically reproducing execution traces and visualizing them in an interactive 3D virtual environment.
- Time-travel debugging with focus of attention at the level of the concurrency model, with animation of the state and message sequences and visual ("point and click") options for debugging tasks such as breakpoints.
- *Online* control of the underlying message scheduler as well as offline playback of a trace.
- Integration with dynamic program analyses such as causality analysis on actors.

While we acknowledge that iDEA scratches the surface of immersive environments for program comprehension and leaves many questions relating to user perception of concurrent program executions open, we believe it is a useful first step towards expressive immersive program visualization and debugging environments.

2 Motivation

2.1 Programming Model

The actor model of programming is a high-level programming model that is concurrent and distributed by design. Actor programs consist of *actors*, which are independent computation units with their own local state. Actors do not share state and communicate with each other by exchanging asynchronous messages.

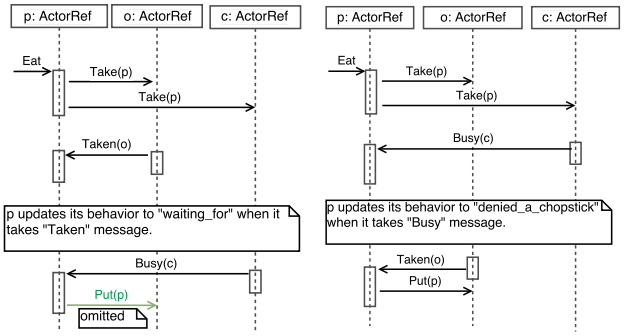
Actor model implementations associate each actor a mailbox which collects messages sent to that actor. An actor processes messages in its mailbox serially. In response to processing a message, an actor can send further messages, create new actors, and update its local state. An actor's behavior —including which messages it is prepared to handle— depends on its local state and may change when it processes a message. An actor program is scheduled by an application-level scheduler which picks which actor to run next. While the scheduler may internally use a thread pool, the lack of global state maintains the atomicity of each actor.

Due to the asynchrony and concurrency in actor programs, a program can have several different executions. Messages sent to an actor from other actors are concurrent to each other and can be delivered to its mailbox in any order. Different implementations of the actor model pose additional constraints on possible executions. In this work, we use the Akka actor library [3]. While Akka runtime has a message ordering guarantee per sender-receiver pair (i.e., for a given pair of actors, messages sent directly from the first actor to the second will be received in-order), it has no ordering guarantees on messages sent by other actors. Thus, message exchanges should be orchestrated in a way that the program behaves as expected in *all* possible orderings of messages.

[3]The videos showing the functionality of iDEA can be found at http://vr.mpi-sws.org/idea.

```
def waiting_for(chopstickToWaitFor: ActorRef,
     otherChopstick: ActorRef): Receive = {
   case Taken(`chopstickToWaitFor`) =>
     context.become(eating)
     self ! Think
   case Busy(`chopstickToWaitFor`) =>
     context.become(thinking)
     // The programmer omits the following line:
     // otherChopstick ! Put(self)
     self ! Eat
 }
```

Figure 1. Code excerpt from a buggy implementation of the dining philosophers problem. It is missing a message exchange between `Philosopher` and `Chopstick` actors.



(a) A buggy execution **(b)** A correct execution

Figure 2. The sequence diagrams for two different executions. The bug occurs when the philosopher actor receives the `Taken` message before the `Busy` message. The code for only `waiting_for` behavior is presented due to space constraints.

2.2 Motivating Example

We now present a sample buggy code to show that users need to understand communication patterns in an execution to detect problematic behavior in actor programs.

Figure 1 shows an excerpt from an implementation of the dining philosophers problem [22] in Scala using the Akka actor library. The program consists of some `Philosopher` and `Chopstick` actors. Each `Philosopher` actor communicates with two `Chopstick` actors and each `Chopstick` actor communicates with two `Philosopher` actors. A `Philosopher` sends a message to a `Chopstick` either to take the resource or to put it back. Similarly, a `Chopstick` responds by sending a notification if the actor can take the resource or if it is busy (already taken by some other `Philosopher`). The program starts with `Philosophers` sending messages to `Chopsticks`.

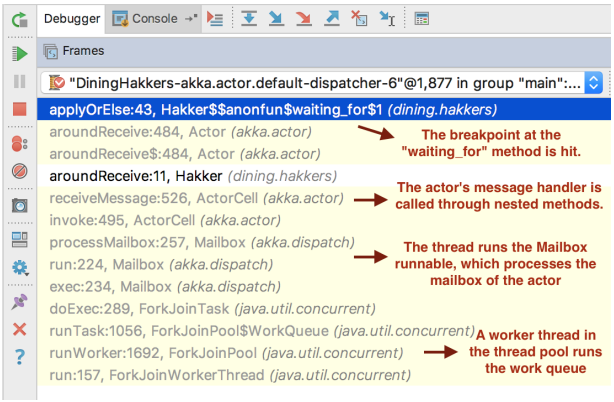


Figure 3. The thread call stack shows unrelevant information for the message communication. It lists the method calls in the framework but it does not show the sender and the content of the received message.

In the code sample, the `Philosopher` actor behavior `waiting_for` is given.[4] A philosopher reaches this behavior when it acquires one of the chopsticks (`otherChopstick` in the code). In this behavior, the actor responds to either a `Taken` or `Busy` message received from the second chopstick. The `Taken` message is sent by the chopstick if it was available when it received the philosopher's request and hence can be taken by this `Philosopher`. If this is the case, the `Philosopher` has both of the chopsticks and updates its behavior to `eating`. It also sends a `Think` message to itself so that it can update its behavior to `thinking` in the future, when it receives this message. The `Busy` message is sent by a chopstick if it is already acquired by another philosopher. In this case, the `Philosopher` updates its behavior to `thinking` and sends an `Eat` message to itself to ask for the chopticks again when it receives this message in the future.

The given code has a bug which leads to a deadlock where none of the philosophers can acquire both chopsticks. This is because the commented line, which sends a `Put` message to release the already acquired chopstick if the second one cannot be retrieved, has been omitted. Thus, one of the resources stays unavailable even though no philosopher uses it. The sequence diagrams for a buggy and a correct execution are given in Figure 2.

The buggy behavior in this example occurs only when the message from the available resource is processed before the message from the busy resource. Depending on different ordering of messages in the system, the bug may appear earlier or later in an execution trace. To analyze a buggy trace, one needs to have an understanding of the communication in the application.

[4] In Akka, an actor behavior is defined as a partial function of type `Receive` (i.e., `PartialFunction[Any, Unit]`, which accepts any message type and does not return a value). The application logic case splits on the kind of message and implements the intended behavior.

In order to debug the problem, the user should be able to easily visualize the *trace* (sequence of messages between actors) leading to the problem, potentially moving back and forward in time, be able to *query* the state of each actor, set *breakpoints* on an actor of interest, and check which actors are transitively influenced by a message (*causal flow*). Additionally, she should be able to play around with different schedule choices at different points in the trace.

2.3 Debugging with Conventional Tools

Conventional debugging tools in IDEs do not provide high-level information to reason about communication patterns in actor programs. They mainly provide the user with the ability of setting breakpoints at some statements, run program to these breakpoints, step forward in the execution and watch the variable values of interest. The main information provided by these tools (i.e., variable values and method call stacks of threads), are not sufficient to debug actor programs.

Watching values of the program variables does not reveal concurrent interactions between actors. Moreover, the current behavior of an actor (e.g., the behavior function defining how the actor handles a message) is not immediately visible from program variables. Actor behavior is implemented in the actor library using a stack of message handling methods. One has to analyze the relevant variables in the actor library to infer about the change of actor behavior.

The thread call stack information is not relevant while debugging an actor program, since the actor abstraction is not tied to underlying threads. For a sequential program, the method call stack provides the context that leads to the current point of execution. However, in actor based programs, the call stack only roots back to the message handler method invoked on a worker thread by the scheduler. For example, Figure 3 provides a sample call stack[5] obtained from IntelliJ IDE. This information gives no clue about the program state prior to the currently processed message (e.g., which actor sent this message in response to which program action).

The limitation of conventional debuggers has led to specialized debuggers for message passing programs [13, 14, 37]. iDEA shares the philosophy of these tools, but integrates debugging with an immersive visualization on the one hand and provides expressive primitives such as time travel, online control of scheduling, and dynamic analysis such as causality on the other hand. Table 1 summarizes the actions available to iDEA. We describe the architecture of iDEA and these features in the next two sections.

3 iDEA Architecture

Figure 4 shows the architecture of iDEA. It consists of two main parts: the programming infrastructure and the visualization infrastructure. These pieces are decoupled from

[5] Here we provide the call stack for Akka implementation in Scala. The information revealed in the call stack might differ in other languages.

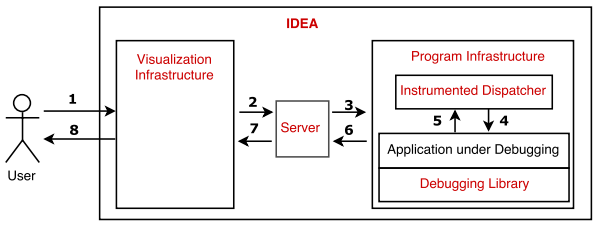


Figure 4. The architecture of our tool. 1. The user provides debugger inputs in the VR environment. 2-3. The user request is sent to program environment. 4. The dispatcher processes the user request. 5. The dispatcher collects program events. 6-7. The dispatcher sends the program response. 8. Program events generated in response to the user request are visualized.

each other by means of a simple server application which forwards user actions from the visualization interface (e.g., scheduler choices) to the running program and program events (e.g., messages sent and received by actors) to the visualization interface. We have developed a JSON-based protocol to exchange messages between the two parts.

The programming infrastructure runs the application being debugged using an instrumented message dispatcher. The instrumented dispatcher is the primary component implementing the debugger logic on the programming side. It processes the user inputs for debugging, collects relevant program events, and sends them to the visual environment for visualization. Additionally, the programming infrastructure provides a set of debugging libraries to integrate an actor program to IDEA; these include visual elements for describing an actor or displaying its state. With the instrumented message dispatcher, we can run Akka applications on the unmodified Akka runtime [3] with a configuration to use our dispatcher instrumented with debugger functionality.

The visualization environment is a 3D virtual reality environment that visualizes program events online (as they occur in execution) and presents and handles user interface (UI) options. It enables users to explore and understand message exchanges between actors and state evolution. We use the HTC Vive Virtual Reality (VR) head mounted display and hand controllers along with Unity3D Game Engine [54] to implement the immersive visualization environment.

While we focus on Akka, the loose coupling between the two parts of IDEA makes it easy to extend to other programming languages and libraries (provided that the message dispatcher in the new programming environment communicates with the visualization part with our protocol).

3.1 Instrumented Dispatcher

The core logic of our debugger is implemented in a custom Akka message dispatcher which we have instrumented with the ability to enforce a specific delivery order of messages and to collect program events. The message dispatcher subclasses `akka.dispatch.Dispatcher` from the Akka framework and is invoked by the Akka runtime each time a message is sent to an actor to deliver the sent message to the recipient actor's mailbox. We have modifed the standard dispatcher to intercept all messages to program actors and collect them in buffers. The dispatcher delivers a message from the buffer to its recipient actor only when there is a user request for this dispatch. To serialize the processing of messages in the application, the dispatcher is configured to use a single thread. However, since actors do not share state, this does not rule out potential behaviors.[6]

The dispatcher communicates with the `DebuggingHelper` actor to handle user requests from the server. We define a message hierarchy to refer to different functionality requests from the visualization infrastructure. Whenever the dispatcher receives such a message, it handles the associated request.

Our dispatcher supports the following user requests:

- Dispatching the next message for delivery (either a message selected at runtime or replaying the next message in a recorded trace). The dispatcher delivers the message and processes the mailbox of the recipient actor synchronously. During the processing of the message, it collects program events for created actors and messages sent. After message processing is finished, it sends the collected list of program events as the response back to the server.
- Querying the behavior and the local state of an actor: the dispatcher retrieves the actor state using a programmer-specified handler or by reflection.
- Setting a breakpoint on an actor: the dispatcher internally maintains a list of actors with breakpoints and, in the execution of the recorded trace, the execution breaks before a breakpointed actor receives its next message.
- Going back to a previous point in the execution: the dispatcher records a history of events at each point in the execution. When the user goes back in execution history, the dispatcher repeats sending the same program events executed until the current point in actual execution is reached.

Note that IDEA explores the non-determinism of an actor program at the scheduling level. Thus, an atomic step in IDEA is the processing of a single message, and not the execution of a single instruction. A step forward in the execution results in handling of a message, during which messages may be sent to some actors, new actors can be constructed, and the recevier's state is updated atomically.

[6] In principle, Akka actors can run arbitrary Scala code, including threads and shared state between actors. Our current implementation ignores the additional source of non-determinism arising out of low-level shared memory concurrency.

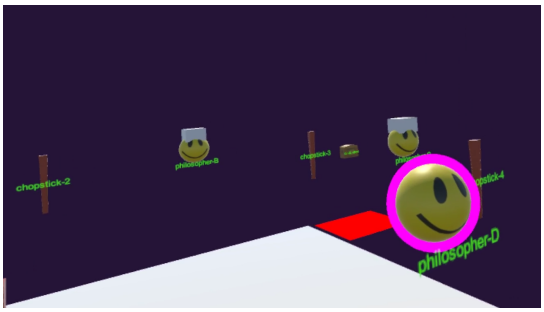


Figure 5. The visualization environment. One of the actors is highlighted suggesting it is involved in the current action.

3.2 Visualization Environment

Visualization Metaphors. In IDEA, actors are represented as 3D visual entities in space. The shape of an actor is customizable, and can range from simple geometric bodies such as a cube, sphere, or cylinder) to specialized 3D meshes. By default, the shape is fixed for a class, therefore instances of the same kind of actor have the same shape and are easy to identify.

Messages between actors are represented as dynamic arrows between the sender and the receiver. The arrows fade over time, thus the current visual state maintains a short-term history of messages in the system. Virtual time (scheduling steps) coincides with actual animation time: at any point, the set of geometric bodies represent the "current state" and evolution of the state is represented as animation of message arrows.

The iconography uses shape, color, and visual boxes for visual representation of state and to guide attention. For example, a small box representing an actor's mailbox comes up on top of actors that have one or more pending messages and a screen displaying the internal state opens up below an actor when its state is queried. When an actor is involved in an action, as a sender or as a recipient, it is highlighted so as to guide attention to it. Breakpoints are visualized using arrows above the actor, and a transparent box represents "source events" for causality.

3D Workspace. The virtual environment consists of a workspace (7m x 6m) where all actors are instantiated. In addition, there is an area marked red (1m x 1m) where system actors (all actors are ~(0.2m x 0.2m x 0.2m)) are instantiated (see Figure 5). This demarcation is done to avoid system actors from cluttering the primary workspace. A virtual console adjacent to the workspace explicitly lists all events happening online in text form. Users can freely move around the workspace by either walking or by teleporting to a particular location. A portion of the workspace is demarcated as a focus area by highlighting the ground. Actors can be brought to the focus area through point-and-click.

Interaction Mechanism. Two tracked hand controllers are used for interacting with the system. One of them has a laser pointer attached to it and is used for functionality that requires a user to point at something (teleporting to a point, querying a particular actor, etc.). The other controller is used for functionality that does not require pointing (next step in trace, increasing/decreasing simulation speed, etc.). The controllers can also be used for picking up actors and reorganizing them in the workspace.

Programmer API for Visualization. The visualization library in IDEA provides a user API that enables users to provide customized information and visualization options for the debugging process. Briefly, users can use API methods for:

- Specifying which information is to be displayed/visualized. Developers usually have suspicions about certain actors, variables, or messages being responsible for unexpected program behavior. Users can use the API to customize which information is visualized when an actor state or message content is queried.
- Specifying how to display information. Users can specify some 3D shapes to be used for visualizing certain actors. They can also specify a topography of the actor system by providing coordinates for actors to be displayed in.
- Specifying timer scheduled messages. Users must call the specific scheduling methods in the API instead of default timed scheduler methods provided by the Akka library to enable our dispatcher to control the order of timed messages.
- Configuring options. Configuration parameters such as which trace is to be replayed, granularity of virtual time advancement, etc. can be provided by either using the API or the debugging configuration file.

4 Debugging in IDEA

In this section, we explain the debugging functionality provided by IDEA, summarized in Table 1.

4.1 Basic Debugging Functionality

Querying Actor State and Mailbox. An actor encapsulates its local variables, behavior (i.e. how it handles the received messages) and its mailbox (which keeps the messages delivered to it). To observe the actor state using a traditional debugger, one needs to log or watch the values of specific local variables, analyze the contents of the behavior stack and the mailbox implemented in the actor framework. IDEA provides quick access to this information by pointing at an actor and querying its state (see Figure 6). Once queried, a black console appears below the actor's visual representation which lists its local variable values. The variables and the state information to be displayed can be configured to avoid clutter in the visualization. Also, the actor metaphor

Table 1. Goals in IDEA and the corresponding operations in traditional debuggers (e.g., IntelliJ commonly used for Scala)

Goal	Description	Traditional Debugger (e.g., IntelliJ Debugger)
Basic debugging functionality		
Query actor state	Query the local variables, current behavior and the mailbox of an actor	Watch specific program variables, including the variables for actor behavior and the mailbox defined in the actor framework
Step forward	Go forward in the execution until the next message is received by an actor	Step to next statement/block in the execution until the next message is received, or use additional breakpoints
Set/unset a breakpoint	Mark an actor so that the execution suspends before that actor receives a message	Add/remove statement-level breakpoints to message handlers
Continue until breakpoint	Run the program until a breakpointed actor receives a message	Continue until breakpoint (causes all instances of the actors using the breakpointed statement to hit the breakpoint unless additional checks are applied)
Trace manipulation		
Display execution history	Display the sequence of messages exchanged between the actors and created/destroyed actors	Explicitly add log statements
Backtrack to a previous step	Go back to a previous step in the execution trace	Not directly supported
Manipulating schedule	Enforce a selected actor to receive its next message	Not directly supported
Visualization helpers		
Positioning Actors	Prespecify a topography of actor metaphor positions or move them on the runtime	-
Tracking Causality	Track the messages caused by a certain actor/message transitively	-
Suppressing actors	Suppress the visualization of unrelevant actor messages	-
Focus Area	Move a selected set of actors to a focus area	-

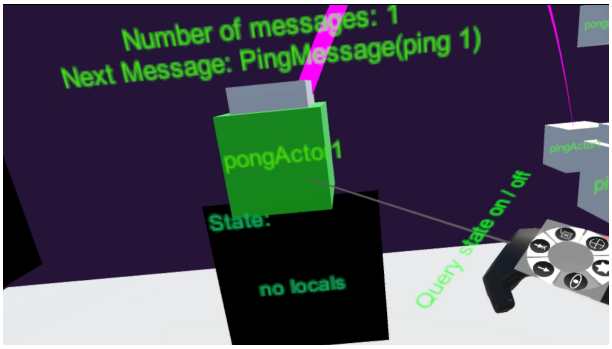


Figure 6. State of `pongActor1` is queried.

color is updated, representing its behavior. As color is an easily identifiable feature [15], it can be used as an abstraction to encode relevant information (e.g. how active an actor is, what behavior it currently exhibits, if it is busy or free, etc.). The mailbox information (i.e. number of messages and the next message) is displayed above the actor's mailbox.

Stepping Forward. IDEA supports message-oriented stepping which is shown to be effective for message passing programs [37, 41, 51, 52]. A step is defined by the atomic processing of a single actor message and the user can progress forward in the execution by stepping to the next message. After processing each step, IDEA visualizes the program actions (i.e., creation/destroyal of actors and sending/receiving or dropping of messages) executed in this step. As an example, Figure 7 shows an actor `Entity-E` sending a message to `Resource-1` and `Resource-5` during processing of its message.

Deterministic Record/Replay of Traces. The instrumented dispatcher can deterministically record an execution trace since it processes and records only a single message at a time. Since an execution of an actor program can be defined as a sequence of messages processed, it suffices to record only the messages and the order in which they are received and processed in the execution.

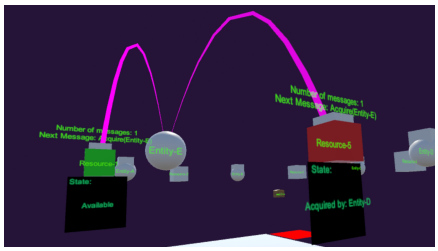


Figure 7. Looking at the arcs, it is easy to identify that `Entity-E` sends a message to `Resource-1` and `Resource-5`.

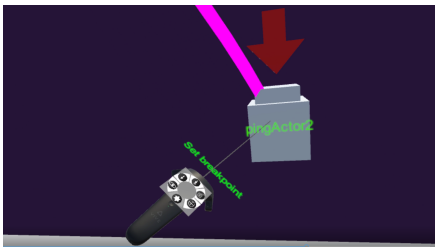


Figure 8. A breakpoint is set on `pingActor2`.

Figure 9. The white bullet corresponding to an atomic step can be selected to go back to that point in history.

A recorded execution trace can be replayed deterministically by (i) replaying the exact same order of message exchange as in the recorded trace, (ii) determinizing the order of timed scheduler messages w.r.t. actor messages, and (iii) determinizing other sources of nondeterminism (e.g., randomness using the same randomization seed). In some applications, certain messages are sent to actors after some time delay or periodically over time. Due to timing nondeterminism, the ordering of these messages with respect to other actor messages is not deterministic. Hence for (ii), we introduce a virtual timed scheduler. This scheduler uses a timer actor that always has a message in its mailbox that advances time when received. As messages to the timer actor are controlled as well as other actors, the advancement of time (and hence when to send a timed message to an actor's mailbox) is determinized.

Setting/Unsetting Breakpoints. Similar to the stepping operation, breakpoints are supported in message-oriented notion, which are hit just before certain messages are received. iDeA allows its users to set breakpoints on actors, so that it is hit before the selected actor receives its next message. Breakpoints are visually represented as an arrow on top of the breakpointed actor, as in Figure 8.

Continue until Breakpoints. By default, iDeA suspends the execution after processing each message. The continue operation provides running the program uninterruptedly until a breakpoint is hit. This is useful for debugging long traces where only certain actor messages are of interest.

4.2 Trace Manipulation

Display Execution History. The sequence of executed program actions are listed in a console in the virtual environment (see Figure 9). The program actions executed as part of processing the same message are grouped and labeled with bullets. This allows the programmer to see the atomically executed actions. Each time the user steps forward in the execution, a new bullet with the recent program actions is prepended to the console.

Enforce a Schedule. By default, iDeA runs an execution replaying a recorded schedule (e.g., a schedule determined to be problematic). However, iDeA also allows the user to deviate from the recorded trace by selecting the next message to be delivered. The user can point at an actor and click a certain button to enforce its next message to be processed. This enables exploring the consequences of different schedules to better understand the problem in an execution.

Backtrack to a Previous Step. Time travel debugging enables users to go back a few steps and reexamine part of the execution of interest. In iDeA, backtracking functionality is integrated onto the console-like screen which lists steps that have occurred so far. This makes a review of prior events easy and also offers an intuitive way to backtrack to a particular step by simply selecting it.

We implement backtracking by collecting the states of the actors for the executed steps. The state of an actor encapsulates its behavior and the valuation of some local variables configured by the user. At each step of the execution, the state of the actor which processed the message is updated while saving the previous state information. When the programmer backtracks to a previous step in the execution, the actor states and the mailboxes corresponding to that step are visualized. Our backtracking implementation does not cause high memory overhead since (i) only a single actor state is updated at each step and (ii) the variables of interest configured by the programmer are collected instead dumping all program variables.

4.3 Visualization Helpers

Positioning Actors. To prevent clutter in the virtual environment as the number of actors increase, iDeA allows for the customized organization of the workspace in the following ways:

- Specifying a general topography using the programmer API. Figure 5 shows the ring topology for a system where an actor interacts primarily with adjacent actors. This example also uses programmer specified icons for visualization of the actors.
- Assigning a physical location to an actor while creating it. This is useful for IoT applications where each actor has a known geographic location.

- Picking up and placing actors individually inside the virtual environment to logically organize actors online.

As locomotion inside VR is a known cause of fatigue and VR induced nausea, IDEA provides shortcuts to avoid the need for movement inside the virtual environment. Essentially, IDEA can also be used to its full extent seated and with limited hand movement. Speed of animation and playback can also be adjusted to avoid or reduce simulation related sickness. Latency is almost negligible because of the simplicity of geometric shapes and animations used.

Focus Area and Visualization Suppression. Quite frequently, for a given execution of an asynchronous program, there are a few actors that are of specific interest (while a general idea of what other actors are doing is still required). Focusing on these relevant actors can be quite easily achieved in the virtual environment with the abstraction of a focus area. Users are allowed to snap certain actors into/out of focus area– which provides a handy way to focus attention on certain actors while still engaging the peripheral visual system to maintain track of actors not in focus (see Figure 10).

Along similar lines, to aid in the debugging process, there is a provision of suppressing unnecessary visualizations, for example initialization messages, networking messages, internal protocol messages and messages between actors that are known to be bug-free. This can be done in code while programming or online inside the virtual environment. Selective visualization makes the overall debugging experience clearer by de-congesting the environment and removing distractions due to unimportant events.

Tracking Causality. Tracking the causal relationship between messages helps in identifying problematic executions arising from asynchronous communication. IDEA allows the user to mark some actors with colors so that every message outgoing from these actors in the future is marked with that color. As shown in Figure 11, this allows to visually track messages caused transitively by a specific actor.

This feature is useful for detecting *atomicity violations*, where two messages are intended to be processed one after the other but another concurrent message interleaves them. Such an interleaving or violation will be explicit (see Figure 12) when actors are marked with different colors.

4.4 A Sample Debugging Scenario

Debugging an application using IDEA basically involves navigating the execution replayed from a buggy trace and using the debugger functionalities at any point of execution. Consider the buggy execution explained in Section 2. The immersive visualization makes it easier to notice the problem at the point where a philosopher omits to release the chopstick. There will be no visual message interaction between that `Philosopher` and the acquired `Chopstick` when the `Philosopher` goes into the `thinking` state. Whenever another `Philosopher` asks for that `Chopstick`, state queries on the actors will make the problem explicit. i.e., the state and the red color of the `Chopstick` will show that it is `busy`, although none of the actors currently acquires it.

During the debugging process, the users additionally might want to backtrack in the execution to observe the anomaly, or use breakpoints to fastforward some parts of the trace. To focus on the problem, the users might move the actors with suspicious behavior closer to the focus area or suppress messages of some other actors.

5 Related Work

Debugging Message Passing Systems. Several tools have been proposed for improving the debugging of message passing programs [8, 16, 18, 23, 28, 36, 44]. Closer to our focus of debugging actor programs, Scala debuggers for actor programs [6, 13] allow for stepping over a message and present additional information useful to extract the asynchronous communication history in the system. Briefly, it saves the stack frames at some specific points of execution (e.g., when a message is sent to an actor) and displays them at relevant breakpoints. Erlang debugger [14] allows for displaying the message queue contents and shows the sent/received messages by processes. A recentErlang debugger CauDEr [31] allows causally consistent reversible debugging, which enables the user to undo the actions of a process. REME-D [10] is implemented for AmbientTalk distributed applications running on mobile networks. It provides message level breakpoints, allows for querying actor states and the history of messages. While these debuggers improve debugging of actor programs, they do not provide execution visualization capabilities.

Postmortem debuggers analyze the recorded execution traces to extract useful information for the identification of software errors. Causeway [51], designed for communicating event loop programs, extracts and displays the causal relation of messages in the system. It presents a user interface with different views for displaying the process order, message order, stack explorer, and source code. However, being a postmortem debugger, it does not provide online debugging functionality.

The works in [40, 41] focus on supporting proper debugger abstractions for different models of concurrency. They design a protocol which customizes the debugger functionality (e.g., breakpointing at messages, transactions or atomic blocks) depending on the underlying concurrency model. Another work by the same authors [38] studies the bugs in actor programs and identifies (but does not implement) the main debugging techniques for analyzing actor programs. IDEA implements the functionality discussed in these papers. Concurrent to our work, Actoverse [49] is designed to provide debugger functionality for actor programs together with a 2D visualization of the exchanged sequence of messages.

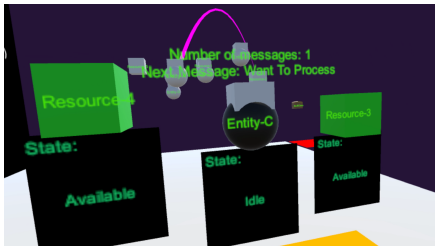


Figure 10. `Resource-4`, `Entity-C` and `Resource-3` are placed in focus area for specific attention.

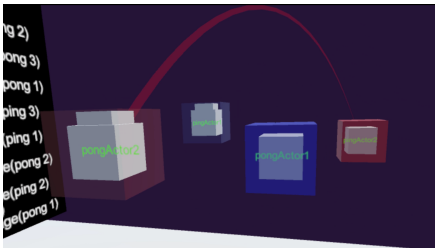


Figure 11. When `pingActor2` (marked red), sends a message to `pongActor2`, it will cause `pongActor2` to be marked red as well.

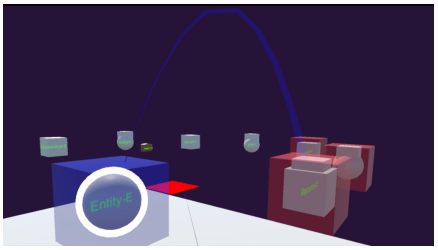


Figure 12. `Entity-C` (marked blue) is sending a message to `Resource-5` (marked red) (possibly problematic).

Visualization for Software Debugging. Research shows that software visualization improves program comprehension and debugging [7]. Along this direction, several program analysis and debugging tools are proposed for exploring program behaviour or performance issues [12, 19, 27, 46] as well as exploring concurrency properties such as thread interleavings or data races in multithreaded programs [35, 39, 53]. We focus on visualization for debugging of distributed and message-passing programs.

Most existing visualization tools for message passing systems visualize the executions in 2D environments. Poet [29] and Shiviz [1, 9] focus on visualization of the message communication in distributed systems. Though these tools differ in implementation and use, they basically instrument the distributed system with some virtual clock information [30] and visualize the order of messages as a 2D diagram. Viva [33] is a visualization tool for event-based systems which displays the exchanged messages and the dependencies between them extracted by a hybrid of dynamic and static analysis. Akka-viz [5] is an experimental tool developed for Akka programs. It provides different views including a 2D graph showing the exchanged messages, state machine diagram for the state transitions of actors, and allows inspecting the local states of the actors. It allows for monitoring the execution at runtime but is not integrated with debugging functionality for actor programs.

Although debugging and visualization of distributed systems has been popular recently, to the best of our knowledge, iDeA is the first tool designed for actor programs (i) to visualize the asynchronous interactions and (ii) to provide debugging functionality interactively in an immersive virtual reality environment.

6 Discussion

We have presented iDeA, the first immersive debugging and visualization environment for actor programs. iDeA exposes a rich set of visualization and manipulation capabilities for inspecting and navigating program traces in the debugger. It also provides a number of abstractions based on attention and perceptual organization to effectively debug programs in an immersive environment.

The debugging infrastructure of iDeA scales to all Scala features and many number of actors. We have used iDeA for debugging large real world Akka programs with some known concurrency bugs, such as Gatling stress testing tool [17] (~19750 LOC) and Raft protocol implementation [4] (~2930 LOC).

We now discuss some limitations of our current implementation as well as directions for future work.

First, while VR devices have made significant progress with respect to capabilities, the resolution of current headsets is not good enough to read detailed text, especially when objects are far away. As a direct repercussion of this, raw variable values are sometimes difficult to read. A provision to read relevant source code has been therefore purposefully omitted in the virtual environment. Future work may partly address this by displaying selected or relevant parts of the code (i.e., the message handler of an actor). A related point is that because of limited UI options, it is not possible to edit code inside the virtual environment. To make changes to the code, the user is expected to take off the headset and use a standard text editor or IDE. This limitation may however not exist when using an augmented reality headset such as Microsoft Hololens [43].

While this rules out certain forms of debugging, for example, tricky sequential reasoning within an actor's message handling code, we believe that a large class of debugging problems fall into the category where the sequential message handling is understood and the debugging only involves visualizing and understanding concurrent ordering of messages. We expect that in the coming years, VR will mature as a medium and some of these current hardware limitations shall be solved.

In terms of domain-specific visualization, we do not have experience on how the system would scale as number of actors increase (over ~1000), especially in terms of conceptual understanding of message provenance and actor states over time. There has been some preliminary research into this [42], but more work remains to be done in this direction so

as to understand perceptual limits to tracking visual objects efficiently in an artificial immersive environment. A comprehensive treatment of large-scale message visualization and compression goes beyond the scope of this paper, but related work in 2D visualization interfaces [24] can be adapted in our system.

Increasing the number of actors also increases challenges in keeping and directing user attention. For example, in the presence of a large number of actors, it becomes difficult to follow which actor is active. This is partially dealt with by highlighting active actors. However, when the active actor is not directly in field-of-view of the user, we currently do not have ways to guide attention to it unobtrusively. Automatically moving "hot" actors to the focus area can lead to too much peripheral motion and induce *VR sickness* [32]. Further, we do not yet provide automatic means of clustering actors and messages. We leave these aspects to future work.

IDEA also leads to inter-disciplinary problems at the boundary of software analysis, visualization, and perception. For example, many fundamental questions about perceptual organization in artificial 3D immersive environments, such as search performance or attentional mechanisms, are not understood. In companion work in collaboration with cognitive psychologists, we are exploring the perceptual models underlying debugging tasks, such as search and attention in a 3D world.

References

[1] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D Ernst. 2014. Shedding light on distributed system executions. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 598–599.

[2] Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report. Massachusetts Institute of Technology, Cambridge Artificial Intelligence Lab.

[3] Akka. 2017. Akka. http://akka.io. Accessed: 2017-07-10.

[4] Akka-raft. 2017. Akka based implementation of Raft concensus algorithm. https://github.com/ktoso/akka-raft. Accessed: 2017-12-01.

[5] Akka-viz. 2017. A visual debugger for Akka actor systems. https://github.com/blstream/akka-viz. Accessed: 2017-08-21.

[6] Asynchronous Debugger. 2017. Akka Asynchronous Debugger. http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html. Accessed: 2017-08-22.

[7] Ron Baecker, Chris DiGiano, and Aaron Marcus. 1997. Software Visualization for Debugging. *Commun. ACM* 40, 4 (April 1997), 44–54. https://doi.org/10.1145/248448.248458

[8] Peter C Bates and Jack C Wileden. 1983. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software* 3, 4 (1983), 255–264.

[9] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. 2016. Debugging distributed systems. *Queue* 14, 2 (2016), 50.

[10] Elisa Gonzalez Boix, Carlos Noguera, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt. 2011. Reme-d: A reflective epidemic message-oriented debugger for ambient-oriented applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 1275–1281.

[11] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. 2011. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering* 37, 3 (2011), 341–355.

[12] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. 2002. Visualizing the execution of Java programs. *Software Visualization* (2002), 647–650.

[13] Iulian Dragos. 2013. Stack retention in debuggers for concurrent programs. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA.

[14] Erlang. 2017. Erlang Debugger. http://erlang.org/doc/apps/debugger/debugger_chapter.html. Accessed: 2017-08-21.

[15] John M Findlay. 2004. Eye scanning and visual search. *The interface of language, vision, and action: Eye movements and the visual world* 134 (2004).

[16] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2012. Theia: Visual Signatures for Problem Diagnosis in Large Hadoop Clusters. In *Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques (lisa'12)*. USENIX Association, Berkeley, CA, USA, 33–42. http://dl.acm.org/citation.cfm?id=2432523.2432526

[17] Gatling. 2017. Open source load and performance testing tool for web applications. https://gatling.io. Accessed: 2017-12-01.

[18] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 289–300.

[19] Paul Gestwicki and Bharat Jayaraman. 2005. Methodology and architecture of JIVE. In *Proceedings of the 2005 ACM symposium on Software visualization*. ACM, 95–104.

[20] M. Heath and J. Etheridge. 1991. Visualizing the performance of parallel programs. *IEEE Software* 8 (1991), 29–39.

[21] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Advance Papers of the Conference*, Vol. 3. Stanford Research Institute, 235.

[22] C.A.R. Hoare. 1985. *Communicating sequential processes*. Prentice-Hall.

[23] UC Berkeley ICSI. 2016. Minimizing faulty executions of distributed systems. In *Symposium on Networked Systems Design and Implementation (NSDIâĂŹ16)*. 291.

[24] K.E. Isaacs. 2015. *Analysis of parallel traces via logical structure*. Ph.D. Dissertation.

[25] Katherine E. Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. 2014. Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time. *IEEE Trans. Visualization and Computer Graphics* 20(12) (2014).

[26] D. Kit, L. Katz, B. Sullivan, K. Snyder, D. Ballard, and M. Hayhoe. 2014. Eye Movements, Visual Search and Scene Memory, in an Immersive Virtual Environment. *PLoS ONE* 9(4) (2014).

[27] Andrew J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 151–158.

[28] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. 2000. Deterministic replay of distributed java applications. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE, 219–227.

[29] Thomas Kunz, James P. Black, David J. Taylor, and Twan Basten. 1997. Poet: Target-system independent visualizations of complex distributed-application executions. *Comput. J.* 40, 8 (1997), 499–512.

[30] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[31] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. 2018. CauDEr: A Causal-Consistent Reversible Debugger for Erlang. In *International Symposium on Functional and Logic Programming*. Springer, 247–263.

[32] Steven M Lavalle. 2016. Virtual Reality. (2016).

[33] Youn Kyu Lee, Jae young Bang, Joshua Garcia, and Nenad Medvidovic. 2014. ViVA: A Visualization and Analysis Tool for Distributed Event-based Systems. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 580–583. https://doi.org/10.1145/2591062.2591074

[34] C. Li, M.P. Aivar, D.M. Kit, M.H. Tong, and M.M. Hayhoe. 2016. Memory and visual search in naturalistic 2D and 3D environments. *Journal of Vision* 16(8) (2016).

[35] Xiangqi Li and Matthew Flatt. 2015. Medic: metaprogramming and trace-oriented debugging. In *Proceedings of the Workshop on Future Programming*. ACM, 7–14.

[36] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M Frans Kaashoek, and Zheng Zhang. 2008. D3S: Debugging Deployed Distributed Systems.. In *NSDI*, Vol. 8. 423–437.

[37] C Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and E Gonzalez Boix. 2016. Towards Advanced Debugging Support for Actor Languages: Studying Concurrency Bugs in Actor-based Programs, October 2016. *Presentation, AGERE* 16 (2016).

[38] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. 2017. A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs. *arXiv preprint arXiv:1706.07372* (2017).

[39] Gowritharan Maheswara, Jeremy S Bradbury, and Christopher Collins. 2010. Tie: An interactive visualization of thread interleavings. In *Proceedings of the 5th international symposium on Software visualization*. ACM, 215–216.

[40] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. Kómπoς: A Platform for Debugging Complex Concurrent Applications. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*. ACM, 2.

[41] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. A Concurrency-agnostic Protocol for Multi-paradigm Concurrent Debugging Tools. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages (DLS 2017)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/3133841.3133842

[42] Aman Shankar Mathur, Rupak Majumdar, and Tandra Ghose. 2017. Study of Visual Search in 3D Space using Virtual Reality (VR). In *European Conference on Visual Perception (ECVP)*.

[43] Microsoft. 2017. Microsoft Hololens. https://www.microsoft.com/en-us/hololens. Accessed: 2017-08-14.

[44] Robert HB Netzer and Barton P Miller. 1995. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing* 8, 4 (1995), 371–388.

[45] M. Ogawa and K.-L. Ma. 2009. code_swarm: A design study in organic software visualization. *IEEE Trans. on Visualization and Computer Graphics* 15 (2009), 1097–1104.

[46] Steven P Reiss and Manos Renieris. 2005. JOVE: Java as it happens. In *Proceedings of the 2005 ACM symposium on Software visualization*. ACM, 115–124.

[47] R. Rosenholtz. 2011. What your visual system sees where you are not looking. *Proc. SPIE: Human Vision and Electronic Imaging, XVI* (2011).

[48] R. Rosenholtz. 2016. Capabilities and limitations of peripheral vision. *Annual Review of Vision Science* 2 (2016), 437–457.

[49] Kazuhiro Shibanai and Takuo Watanabe. 2017. Actoverse: a reversible debugger for actors. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. ACM, 50–57.

[50] C. Sigovan, C.W. Muelder, and K.-L. Ma. 2013. Visualizing large-scale parallel communication traces using a particle animation technique. In *Eurographics Conference on Visualization*, Vol. 32(3). Blackwell.

[51] Terry Stanley, Tyler Close, and Mark S Miller. 2009. Causeway: A message-oriented distributed debugger. (2009).

[52] Carmen Torres Lopez, Elisa Gonzalez Boix, Christophe Scholliers, Stefan Marr, and Hanspeter Mössenböck. 2017. A principled approach towards debugging communicating event-loops. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. ACM, 41–49.

[53] Jonas Trümper, Johannes Bohnet, and Jürgen Döllner. 2010. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th international symposium on Software visualization*. ACM, 133–142.

[54] Unity. 2017. Unity - Game Engine. http://unity3d.com/. Accessed: 2017-07-10.