Multiverse Debugging: Non-deterministic Debugging for Non-deterministic Programs

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

Robbert Gurdeep Singh
Universiteit Gent, Belgium
Robbert.GurdeepSingh@ugent.be

Stefan Marr
School of Computing University of Kent, United Kingdom
s.marr@kent.ac.uk

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

Christophe Scholliers
Universiteit Gent, Belgium
Christophe.Scholliers@ugent.be

Abstract

Many of today's software systems are parallel or concurrent. With the rise of Node.js and more generally event-loop architectures, many systems need to handle concurrency. However, its non-deterministic behavior makes it hard to reproduce bugs. Today's interactive debuggers unfortunately do not support developers in debugging non-deterministic issues. They only allow us to explore a single execution path. Therefore, some bugs may never be reproduced in the debugging session, because the right conditions are not triggered.

As a solution, we propose multiverse debugging, a new approach for debugging non-deterministic programs that allows developers to observe all possible execution paths of a parallel program and debug it interactively. We introduce the concepts of multiverse breakpoints and stepping, which can halt a program in different execution paths, i.e. universes. We apply multiverse debugging to AmbientTalk, an actor-based language, resulting in *Voyager*, a multiverse debugger implemented on top of the AmbientTalk operational semantics. We provide a proof of non-interference, i.e., we prove that observing the behavior of a program by the debugger does not affect the behavior of that program and vice versa. Multiverse debugging establishes the foundation for debugging non-deterministic programs interactively, which we believe can aid the development of parallel and concurrent systems.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages; Software and its engineering → Software testing and debugging

Keywords and phrases Debugging, Parallelism, Concurrency, Actors, Formal Semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.27

Category Brave New Idea Paper

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.4`

Funding *Carmen Torres Lopez*: Funded by FWO Research Foundation Flanders (FWO), project number G004816N.
Robbert Gurdeep Singh: Doctoral fellowship from the Special Research Fund (BOF) of Ghent University (reference number: BOF18/DOC/327).

Acknowledgements We would like to thank Thomas Dupriez (ENS Paris-Saclay - RMoD, Inria, Lille-Nord Europe) for an initial implementation of the underlying visualization and reduction code.

1 Introduction

Parallelism has become an integral part of modern software ranging from large-scale server code to responsive web applications or networked embedded systems. While a wide range of high-level concurrency abstractions are available for developers, understanding and debugging parallel programs remains challenging. The main reason why parallel programs are so difficult to debug is due to their non-determinism. Since the state of a parallel program at any given moment in time can alter to one of *many* possible successor states, it is very difficult to reason about their behavior and to reproduce bugs as they may only manifest in rare execution traces.

Debugging tools for parallel and concurrent programs have been studied in the past and can be categorized in two main families [1]: event-based debuggers (also known as log-based debuggers) and breakpoint-based debuggers (also known as online or interactive debuggers). While event-based approaches generate a program trace for offline browsing or deterministic replay, breakpoint-based debuggers control the program execution allowing developers to pause/resume program execution at well-defined points (e.g. on a breakpoint), inspect program state, and perform step-by-step execution.

Despite the presence of online debuggers in modern IDEs, a recent study showed that debugging parallel applications remains very problematic [2], because debuggers do not account for the non-determinism of concurrent applications. Most of the existing tools only provide support for deterministic debugging, i.e., they support the debugging of *only* one parallel entity at a time rather than the program as a whole. This means that one run of the debugger is very likely to miss the erroneous state in which the bug manifests itself, requiring many debugging cycles before being able to reproduce the bug. Even worse, the mere presence of a debugger may affect the order in which parallel entities are executed, making the reproduction of a bug even rarer. This condition akin to the Heisenberg uncertainty principle, is known as the *probe effect* [3].

In addition to debugging techniques, static analyses have been studied for parallel and concurrent programs to detect certain types of program errors without executing the program, e.g. static analysis to verify the boundedness of actor mailboxes [4], model checkers for concurrent programs written in Erlang [5] or Scala [6], type systems to ensure type safety on reciprocal communication channels [7]. These techniques detect synchronization errors such as deadlocks [8, 9], incorrect ordering of locks [10], and incorrect interleaving of messages in actor systems [5]. However, they often put severe restrictions on the way programs are organized (e.g. on how futures are used in actor-based programs [11]). More importantly, they expect developers to have a good understanding of what caused the bug as they verify a well-defined property over a program, but they currently cannot be used interactively to explore and search for a bug with an unknown cause. Finally, static analysis techniques are almost always about approximations, when the analysis detects a bug it might be impossible to find a concrete execution path which triggered the bug.

What is needed to *debug* non-deterministic programs is a technique which: 1) allows programmers to observe *all* the possible states a parallel program can exhibit at run time and 2) is probe-effect free. In this paper, we propose *multiverse debugging*, a novel debugging technique for parallel programs which combines breakpoint-based debugging with state exploration from static techniques. The key idea of multiverse debugging is that *non-*

deterministic programs require non-deterministic debugging. Contrary to current state-of-the-art debuggers, which only execute the program in one execution path (i.e. one *universe*), a multiverse debugger can observe all possible universes. A multiverse debugger is itself a non-deterministic program which is able to explore all possible states of a parallel program while leveraging breakpoints and stepping commands of online debuggers to interactively search for the root cause of a bug. This means that regular breakpoints become *multiverse breakpoints* which are potentially triggered multiple times in different universes. As such, a multiverse debugger ensures that if a bug is in the program, it will be observed during the debugging session.

In this paper we give an overview of how to design a multiverse debugger starting from the operational semantics of a non-deterministic language. We evaluate multiverse debugging by applying it to actor-based programs written in the AmbientTalk language. On top of the existing AmbientTalk operational semantics (known as Featherweight AmbientTalk) [12], we formalized the debugging semantics and developed *Voyager*, a tool that takes as input Featherweight AmbientTalk programs written in PLT-Redex, and allows programmers to interactively browse all possible execution states by means of multiverse breakpoints and stepping commands. We also provide a proof of *non-interference*, i.e. the base language and the debugger are observational equivalent.

The key contributions of this paper are:

- Definition of multiverse debugging and multiverse stepping, i.e. novel stepping semantics which allow developers to step to all possible states in the execution of a parallel program.
- A semantics for a non-deterministic debugger & proof of non-interference.
- An implementation of applying multiverse debugging to an actor-based language including a tool to interact with the debugged program written in PLT-Redex.

2 Brave New Idea: Multiverse Debugging

The vision of multiverse debugging is to allow programmers to debug concurrent non-deterministic programs with a debugging technique that allows them to observe *all* possible states the program can exhibit at run time and to interactively explore these states for bugs in a fashion similar to breakpointed-based debuggers while being probe-effect free. Current debuggers for non-deterministic programming languages do not allow such exploration because they only follow a single path of many possible execution paths. In this paper, we provide a concrete recipe on how to build debuggers which allow programmers to observe *all* possible states of a non-deterministic program. To this end, multiverse debugging builds on the operational semantics of the language in which target programs are written.

2.1 Multiverse Debugging Recipe

We now give an overview of the basic recipe for defining the semantics of a multiverse debugger:

1. Define the operational semantics of the base language, a language which can specify programs that exhibit non-deterministic behavior.
2. Define the operational semantics of the debugger in terms of the base language semantics. This implies to:
   a. define a debugger configuration, which includes the state the debugger needs to maintain to debug a target program.

b. define the debugging operations that the debugger offers to developers to interactively explore the target program, e.g. by pausing/resuming program execution on breakpoints, or performing step-by-step execution of the target program.

In this paper, we apply this recipe to define the semantics of two multiverse debuggers. First, in Section 3, we apply this recipe to build a debugger for a small language called λ_{amb}. Afterwards, in Section 6, we show that our technique scales for a mid-size actor-based language called AmbientTalk [12], a prototype-based object-oriented language with an event loop concurrency model featured by mainstream languages such as JavaScript.

We believe that those two steps are general enough to be applicable to a wide range of programming languages. Applying this recipe to other programming languages consists of identifying where and how non-determinism originates. This is, however, tied to the language's concurrency model. The behavior and properties of concurrent entities (e.g. threads, transactions, actors) differ and hence these properties should be carefully considered when defining the operational semantics of the multiverse debugger.

2.2 Multiverse Debugging Main Challenges

It is our vision that the foundations explained in this paper, places multiverse debugging where research in program analysis and verification was three decades ago. At that time, static techniques were a new brave idea which could only be used to verify relatively small programs [13]. Likewise, the approach towards multiverse debugging explained in this paper is currently only feasible for relative small programs. Nevertheless, we hope that further research can be spawned from the seed we plant here to expand on what is possible today.

The main challenge of our approach is the growth in the number of states; the number of possible states increases exponentially by every non-deterministic step that is chosen in the program. This problem, called *state explosion*, is a well-researched problem in the context of program analysis and verification [14]. Multiverse debugging also suffers from this problem. It is, however, essential to make the complexity of these non-deterministic programs explicit to the programmer so that actions can be taken. We believe that providing a recipe on how to build multiverse debuggers is an important first step which gives developers the tools to explore *parts of this state space* interactively. After all, being able to inspect part of this enormous state space is better than to have a debugging tool which can only explore one execution path without any guarantees that the path being explored triggers the bug.

Symbolic execution and model checking have studied scalable solutions for the state explosion problem [15, 16, 17, 18]. Future research is needed to investigate how to adopt those techniques in a debugging tool to increase the scale on which multiverse debugging can be applied. In section 8.2 we further compare multiverse debugging with symbolic execution and model checking.

In our proof of concept implementation, we apply two techniques to help the programmer to keep an overview of the state space. First, we do not blindly explore all the possible states but let the developer decide which states to explore next, either explicitly or by using multiverse breakpoints. We believe this makes multiverse debugging comparable to bounded model checking [19]. Second, whenever two states are syntactically the same we merge those states into one node. Depending on the programming language other means of equality could be applied to further reduce the amount of states exposed to the programmer.

3 Multiverse Debugging for Ambiguous Programs

In this section, we apply the multiverse debugging recipe to debug ambiguous programs written in λ_{amb}. In section 3.1, we specify the base language semantics (step 1) and section 3.2 defines the semantics of a multiverse debugger on top of it (step 2). To simplify the exposition and focus on the core idea of multiverse debugging, we do not model any breakpoints, stepping commands nor user interaction with the debugger in this section. They are detailed later when we apply the idea of multiverse debugging for actor-based programs in section 4.2. There, we will specify the semantics of a base language in which to write actor-based concurrent programs (step 1), and section 6 describes the semantics of a multiverse debugger on top of it including multiverse breakpoints and stepping commands (step 2).

3.1 Syntax and Operational Semantics of the Base Language λ_{amb}

We now show how the multiverse debugging idea can be applied to the λ_{amb} calculus, a small functional language. Non-determinism in this language is introduced by a variation of McCarthy's ambiguity operator [20] called `amb` which behaves as a non-deterministic choice. Intuitively, when this operator is applied to a number of arguments it returns one of them in an unpredictable way.

Figure 1 gives an overview of the syntax and reduction rules of the λ_{amb} calculus. Expressions consist of numbers, addition, and the `amb` operator. We define an evaluation context E which dictates a left to right evaluation order of the arguments. The only values v in the language are numbers. The ADD rule shows how the addition of two numbers reduces and the AMB rule shows the `amb` operator non-deterministically picks one of its arguments.

$$\begin{array}{lcll} e & ::= & (+\ e\ e) | (amb\ e\ e) | number & \text{Expressions} \\ E & ::= & (+\ E\ e) | (+\ v\ E) | (amb\ E\ e) | (amb\ v\ E) & \text{Context} \\ v & ::= & number & \text{Values} \end{array}$$

$$\text{(ADD)}\quad \frac{n = \lfloor n_1 + n_2 \rfloor}{E[(+\ n_1\ n_2)] \rightarrow_{amb} E[n]} \qquad \text{(AMB)}\quad \frac{e_x \in [e_1, e_2]}{E[(amb\ e_1\ e_2)] \rightarrow_{amb} E[e_x]}$$

Figure 1 Semantic entities and reduction rules of the λ_{amb} calculus.

To get an intuition of the λ_{amb} calculus, consider the evaluation graph of the program `(+ (amb 1 2) (amb 3 4))` shown in fig. 2. While in a deterministic evaluator there is at most one applicable rule for each expression in a non-deterministic evaluator it is possible that multiple reduction rules apply for the same expression. In our example, this is clearly the case. In the start state, there are two possible reductions leading to two execution paths the program could take. We denote a *universe* to each distinct state in which a program can be. In this example, the top universe denotes the state in which the `amb` operator selected the value `1` while in the bottom universe it chose `2`. For these two universes, there are again two possible successor universes possible by choosing between number `3` and `4`. Finally, each of these universes can be reduced by applying the ADD rule leading to three possible end universes.

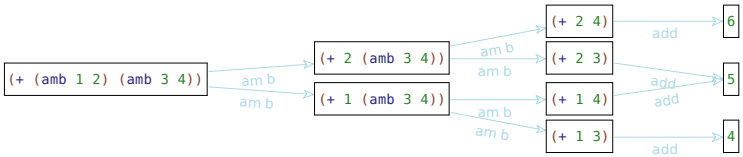

Figure 2 Multiverse evaluation graph of a λ_{amb} program.

3.2 Syntax and Operational Semantics of the Debugger D_{amb}

Armed with the semantics of our non-deterministic language λ_{amb}, we can now define a multiverse debugger for this base language, which allows us to pause a program and resume its evaluation until it reaches an end state. Resuming a program corresponds to a user stepping through the program, expression by expression.

Figure 3 gives an overview of D_{amb}, the semantics of a debugger for the λ_{amb} calculus. We first define the debugger configuration that keeps track of the state that the debugger needs to store to debug a target λ_{amb} program. In this case, the debugger is either paused or has resumed execution evaluating an expression at a time. The debugger configuration is thus a pair which consists of a *state* label (either STEP or PAUSED) and a λ_{amb} expression e.

The debugger operations are defined by two reduction rules. The STEP rule takes one evaluation step of the enclosing λ_{amb} expression e and transitions the debugger state by changing the *state* label from STEP to PAUSED. This means, we take one evaluation step, and yield to the debugger, where a user could inspect the program. Though, for simplicity, the only other operation our debugger has is the RESUME rule, which transitions a paused program back to the step state.

$$
\begin{array}{rcll}
\mathit{state} & ::= & \textsc{step} \mid \textsc{paused} & \text{Debugger State} \\
e_d & ::= & (\mathit{state},\ e) & \text{Debugger Configuration}
\end{array}
$$

$$
\text{(STEP)}\ \frac{e \rightarrow_{amb} e'}{(\textsc{step},\ e) \rightarrow_{debug} (\textsc{paused},\ e')}
\qquad
\text{(RESUME)}\ \frac{}{(\textsc{paused},\ e) \rightarrow_{debug} (\textsc{step},\ e)}
$$

Figure 3 Semantic entities and reduction rules of the D_{amb} calculus.

As an example of a multiverse debugging session, let us execute the program shown in fig. 2 in D_{amb}. The resulting session is shown in fig. 4. It starts by applying the STEP rule on the `(+ (amb 1 2) (amb 3 4))` expression. The first step will cause the reduction of the AMB rule in λ_{amb} for the `(amb 1 2)` expression. This leads to two possible reductions the debugger could take, i.e. one universe in which the AMB rule reduces to `1` and one in which it reduces to `2`. This means stepping to a next state is non-deterministic. By defining the debugger operations in terms of the non-deterministic evaluator of the base language, we automatically obtain a multiverse debugger, i.e. a step does not lead to one possible next universe but to a set of universes.

While this multiverse debugger is simplistic, it already shows two important characteristics. First, *all* evaluation steps observed in the base-level semantic are also observed in the multiverse debugger. This means that when programmers debug their programs in the multiverse debugger, the error will manifest in the debugger. Second, there are no states in

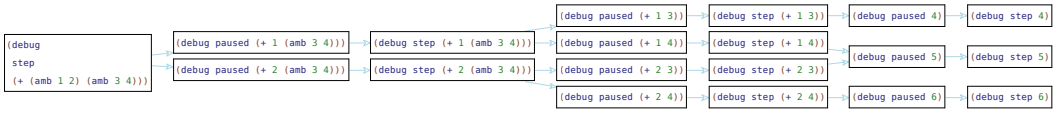

Figure 4 Multiverse debugging graph of a λ_{amb} program.

the debugger which are not observed in the base-level semantics. This means that the act of debugging the program does not introduces any state (and thus also no bugs) which are not observed in the base-level semantics. As such, a multiverse debugger is probe-effect free.

4 Communicating Event Loops (CEL)

The overall goal of this work is to improve the debugging of parallel and concurrent programs. To this end, we now apply the multiverse debugging recipe defined in section 2.1 to create a multiverse debugger for actor-based concurrent programs. To this end, we first need to specify a base language in which target programs will be written in. In this work, we use AmbientTalk [12], a distributed programming language originally designed to develop mobile peer-to-peer applications. AmbientTalk is a prototype-based object-oriented language featuring a concurrency model based on Communicating Event Loops (CEL). The language paradigm is featured by mainstream languages such as JavaScript, the Proxy API of which was actually inspired by AmbientTalk's reflective model [21].

In this section, we first describe the necessary background information on Communicating Event Loops.We then apply the first step of the multiverse debugging recipe by defining the operational semantics of AmbientTalk in section 4.2.

4.1 Communicating Event Loops Concurrency Model

The Communicating Event Loops is a non-blocking variant of the actor model [22] first introduced by the E language [23]. The model was also adopted by languages such as AmbientTalk [12], Newspeak [24], and it is embraced by the asynchronous programming model of JavaScript and Node.js [25].

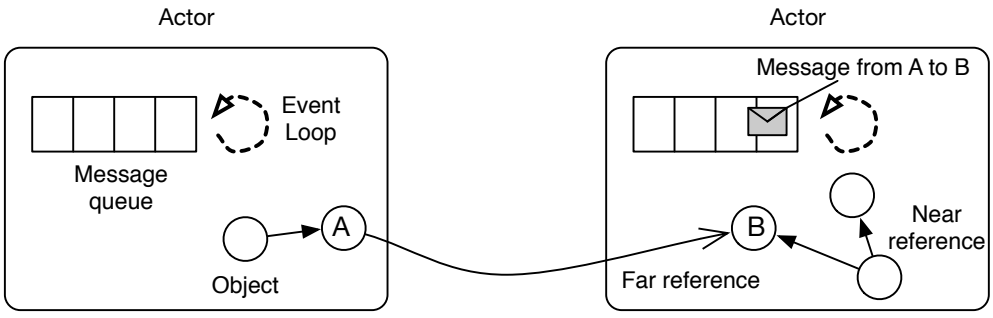

Figure 5 Overview of the CEL model (from [12]).

Figure 5 shows an overview of the CEL model. Each actor is a container of objects, a message queue (or mailbox), and an event loop (or thread of control). An actor executes sequentially messages from its mailbox, i.e. messages are processed one by one in order of

arrival. The processing of one message by an actor defines a *turn*. Actors have exclusive access to their mutable state. This means that each object is owned by one actor and only the owner can directly access it. Communication with objects owned by other actors happens using asynchronous messages via *far references*. When a far reference receives a message, it forwards it to the mailbox of the actor owning the object. Objects passed as arguments in asynchronous message sends are parameter-passed either by far reference, or by (deep) copy.

An asynchronous message send immediately returns a *future* (also known as a promise). A future is a placeholder for the result that is to be computed. The future itself is an object, which can receive asynchronous messages. Those messages are accumulated within the future object and forwarded to the result value once it is available. Once the result value is available, the future is said to be *resolved* with the value. Programmers can register a block of code with a future which is asynchronously executed when the future becomes resolved. Access to the result thus happens in a *non-blocking* way.

4.2 Syntax and Operational Semantics of the AmbientTalk Language

Recall that in order to build a multiverse debugger for non-deterministic concurrent programs, we first need to define the operational semantics of the base language (step 1 of the recipe in section 2.1). In this work, we employ the semantics of the AmbientTalk language, i.e the *Featherweight AmbientTalk* (AT^f) semantics [12]. It formalizes common features of the CEL model such as actors, objects, blocks, non-blocking functions and asynchronous message sending. Non-determinism in the CEL model is exhibited in the order in which actors process messages. Non-blocking futures also introduce additional non-determinism as messages sent to futures, while futures are not resolved, messages are forwarded to the result value once it is available.

The core calculus of AT^f consists of 30 evaluation rules (excluding helper functions). Considering that Featherweight Java [26], a minimal core calculus for Java and GJ, only has 10 evaluation rules, we believe that the AmbientTalk semantics should be not be considered a small language but at least a mid-size one. For brevity, we sketch only the parts of AT^f that are necessary to follow the contributions of this work and refer to Van Cutsem et al. [12] for the complete semantics. In a nutshell, AT^f specifies that actors evaluate messages as expressions to obtain a result value.[1] It is based on a small step operational semantics. This means that the representation of each of the steps of the program execution is atomic, i.e. there is no intermediate execution steps. This is useful because it is possible to get all possible states of the evaluation of a non-deterministic program.

Figure 6 shows the semantic entities of the operational semantics of AT^f. A configuration K represents the set of actors that are executed concurrently in the program. An actor is represented by an identity ι_a, a set of objects O, an inbox queue Q_{in} that stores the messages to be processed and an expression e the actor is currently evaluating. An object O consists of an identity ι_o, a tag t and a set of fields F and methods M. The tag distinguishes between objects passed by reference O, and passed by copy I. A future consists of an identity ι_f, a queue for the pending messages Q_{in} and a resolved value v. A resolver object allows to assign a value to its unique paired future and as such, it consists of an identity ι_r and the identity of its corresponding future ι_f. A message m is represented by an identifier ι_m, a receiver value v, a method name m and a sequence of arguments values $\overline{v}$. References to

[1] In this work we use the subset of Featherweight AmbientTalk for concurrency, i.e. without the notion of networks for distribution.

$$
\begin{array}{rcll}
K \in \textbf{Configuration} & ::= & \overline{a} & \text{Actor configurations} \\
a \in \textbf{Actor} & ::= & \mathcal{A}\langle \iota_a, O, Q_{in}, e\rangle & \text{Actors} \\
\textbf{Object} & ::= & \mathcal{O}\langle \iota_o, t, F, M\rangle & \text{Objects} \\
t \in \textbf{Tag} & ::= & \text{O} \mid \text{I} & \text{Object tags} \\
\textbf{Future} & ::= & \mathcal{F}\langle \iota_f, Q_{in}, v\rangle & \text{Futures} \\
\textbf{Resolver} & ::= & \mathcal{R}\langle \iota_r, \iota_f\rangle & \text{Resolvers} \\
\text{m} \in \textbf{Message} & ::= & \mathcal{M}\langle v, m, \overline{v}\rangle & \text{Messages} \\
Q_{in} \in \textbf{Queue} & ::= & \overline{\text{m}} & \text{Inbox queues} \\
M \subseteq \textbf{Method} & ::= & m(\overline{x})\{e\} & \text{Methods} \\
F \subseteq \textbf{Field} & ::= & f := v & \text{Fields} \\
v \in \textbf{Value} & ::= & r \mid \text{null} \mid \epsilon & \text{Values} \\
r \in \textbf{Reference} & ::= & \iota_a.\iota_o \mid \iota_a.\iota_f \mid \iota_a.\iota_r & \text{References} \\
e \in E \subseteq \textbf{Expr} & ::= & \ldots \mid e \leftarrow m(\overline{e}) \mid r & \text{Runtime Expressions}
\end{array}
$$

$$
\begin{array}{c}
o \in O \subseteq \textbf{Object} \cup \textbf{Future} \cup \textbf{Resolver} \\
\iota_a \in \textbf{ActorId}, \iota_o \in \textbf{ObjectId}, \iota_n \in \textbf{NetworkId} \\
\iota_f \in \textbf{FutureId} \subset \textbf{ObjectId}, \iota_r \in \textbf{ResolverId} \subset \textbf{ObjectId}
\end{array}
$$

Figure 6 Semantic entities of the AT^f calculus.

objects r consist of an identifier for the actor ι_a owning the referenced value and a local component that can be ι_o, ι_f or ι_r. The local component indicates that the reference refers to either an object, a resolver or a future. An expression e can include references r or an asynchronous message send $e \leftarrow m(\overline{e})$.

5 Multiverse Debugging for Actor-based Programs

Having the operational semantics of AmbientTalk, we can now apply the second step of the multiverse debugging recipe to create a multiverse debugger for actor-based programs by defining the operational semantics of the debugger in terms of the AmbientTalk ones. Before detailing this operational semantics in section 6, we informally describe the debugger's breakpoints and stepping semantics. Additionally, we show a debugging session in the resulting multiverse debugger called *Voyager*.

5.1 Breakpoint-based Debugging for Actor-based Programs

As explained before, multiverse debugging allows developers to interactively explore a target program (step 2.b. in the multiverse debugging recipe of section 2.1). In this section we describe the two main features that multiverse debugging borrows from breakpoint-based debuggers to enable such an interactive exploration of the program's state, i.e. breakpoints and stepping operations.

A *breakpoint* defines a point of interest in a program in which to pause execution for further inspection. The main idea of breakpoints is to allow developers to observe the effects on an operation that may interleave with other operations in the system. Since turns run till completion, operation interleavings in CEL programs happen at message level. As such, breakpoints need to be applied to asynchronous message passing.

As a general principle, we consider the point in time right before and right after a message is processed as relevant for breakpoints. Figure 7 shows the points of interests involved in an asynchronous message send in CEL. When object A sends a message to object B, it is first placed in the sender actor's mailbox (point 1) and a future object F is immediately returned. The message is then sent to the actor hosting the receiver object B, and is placed in its

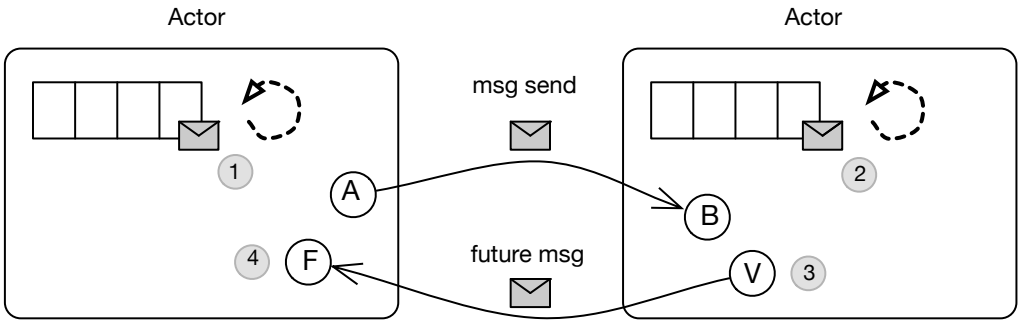

Figure 7 Points of interest for debugging actor-based programs.

mailbox (point 2). After the value for the future is computed, a message with the result value V is placed in the receiver actor's mailbox (point 3) and sent back to the actor hosting the sender object. The message carrying the result value is then placed in the sender's mailbox (point 4) and the future resolution listeners are finally executed.

Stepping is a debugger feature that allows developers to follow the execution of a program between various points of interest. In sequential programs, stepping operations typically allow to step through the program line by line. In CEL programs, stepping also needs to allow developers step through program execution concurrently, i.e. let them follow the execution between the points of interest involved in asynchronous message passing. Stepping operations can thus be applied at each of the points shown in fig. 7, and it will allow the developer to step to the next point of interest. For example, the program may be paused at point 2, before the receiver actor has processed a message. A possible stepping operation is to *step to the next turn*, which will let the actor process the message sent by A, and halt before processing the next message, i.e. at the beginning of the turn.

In prior work we have explored catalogs of breakpoint types and stepping operations for actor-based programs [27, 28]. These debugging operations are used in breakpoint-based debuggers for CEL programs written in AmbientTalk and SOMns[2], respectively. In this work, we apply the debugging operations that we proposed in [28] to our multiverse debugger.

5.2 Voyager: a Multiverse Debugger for AmbientTalk Programs

To showcase the use of a multiverse debugger for AmbientTalk programs, we implemented a tool called *Voyager*, which allows developers to interactively explore a target program through the debugger operational semantics. Voyager is a web application build on top of PLT-Redex, which executes the operation semantics of a multiverse debugger. Both the operational semantics of the debugger and AT^f are implemented in PLT-Redex. Target programs are written in PLT-Redex and can be loaded in Voyager. Voyager then asks PLT-Redex to reduce the program according to the debugger semantics, which results in the reduction graph for the program. All states in this graph are stored in a graph database[3] for easy manipulation and exploration of the reduction graph.

[2] SOMns is a Newspeak implementation build on top of the Truffle platform. `https://github.com/smarr/SOMns`

[3] Our prototype uses ArangoDB. `https://www.arangodb.com/`

```
1  def makeMath() {
2      actor:{
3          def result := 0;
4          def double(x){result := x+x};
5          def getResult(){result};
6      }
7  };
8  def makeClient1(math){
9      actor:{ |math|
10         def start(){
11           math<-double(12);
12           when: math<-getResult()@FutureMessage becomes: {|res|
13               system.println(res);
14           }}}
15 };
16 def makeClient2(math){
17     actor: { |math|
18         def start(){ math<-double(33) }}
19 };
20 def math := makeMath();
21 def client1 := makeClient1(math);
22 def client2 := makeClient2(math);
23 client1<-start();
24 client2<-start();
```

Listing 1 AmbientTalk program containing a bad message interleaving bug.

5.2.1 Debugging a Sample Program

We now show a debugging session in Voyager for the AmbientTalk program depicted in listing 1. The program shows an interaction between 3 actors: a `math` actor (created in line 20, and two client actors, `client1` (created in line 21) and `client2` (line 22). The `math` actor (lines 1 to 7) understands the messages `double`, which doubles its argument and stores the result for further operations, as well as the `getResult` message, which returns the result of a number of operations. After creating the three actors, the program sends a `start` message to both client actors (lines 23 and 24). As a result, `client1` sends a `double(12)` message followed by a `getResult` one. Concurrently, `client2` sends a `double(33)` message to the `math` actor as well.

Despite being a simple program, it contains a *bad message interleaving* bug [29], which is common for actor-based programs. It is possible that `client1` gets the result of doubling `33` instead of doubling `12`. Table 1 shows all possible interleavings that the program exhibits. It also depicts the message sender for each message. Line 13 would print the result value, which is `24` for the correct interleavings and `66` for the faulty one.

Table 1 Message interleavings for the AmbientTalk program shown in listing 1.

Faulty Interleaving	Correct Interleaving	Correct Interleaving
client 1 - `double(12)`	client 1 - `double(12)`	client 2 - `double(33)`
client 2 - `double(33)`	client 1 - `getResult()` → 24	client 1 - `double(12)`
client 1 - `getResult()` → 66	client 2 - `double(33)`	client 1 - `getResult()` → 24

5.2.2 Overview of a Debugging Session

Taking the faulty interleaving of our example program of listing 1 as example, a developer may choose to explore the issue in Voyager and identify why the unexpected result is `66`. Thus, the developer needs to find the cause of the bad message interleaving exhibited by

the program. For a screencast of the debugging session, we refer the reader to `https://tinyurl.com/VoyagerYoutube`.

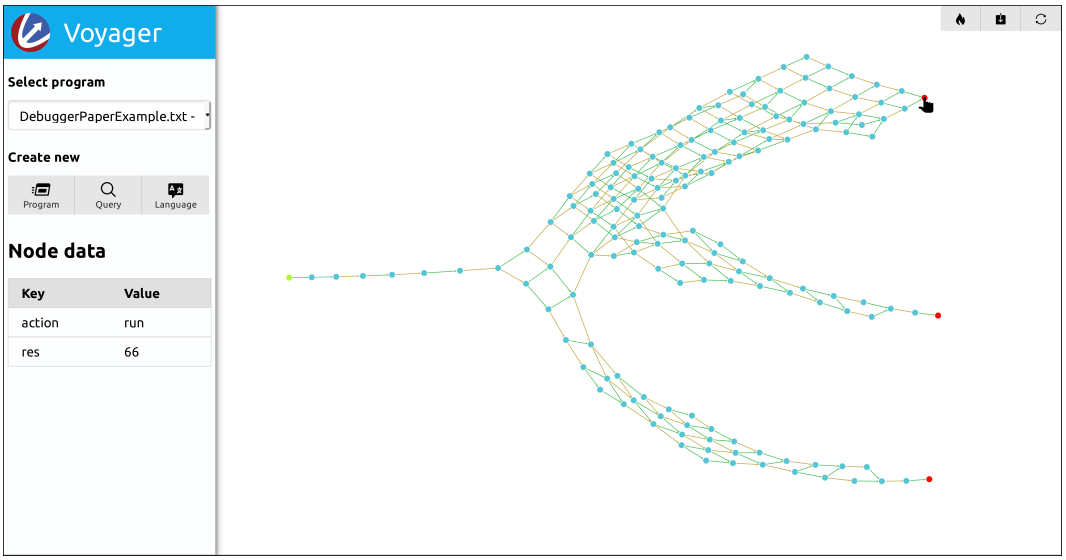

Figure 8 Overview of the Voyager tool.

Figure 8 shows the Voyager UI. The left panel allows developers to upload the target program to debug (either by selecting an existing file or creating a new one directly), and shows information on the selected node in the "Node data" section. The right panel shows the reduction graph for the target program. In this case, Voyager shows all possible universes for the sample program. The end states of the program are shown in red. As expected (cf. table 1), there are three possible end states. "Node Data" shows the information for the end state under the cursor. The selected state corresponds to the end state of an execution path with the faulty interleaving since the result stored in `res` is `66`.

Let us now start a debugging session to understand how we arrived at the result being `66`. Since `math` actor is the central point of synchronization in the program, we set a breakpoint that pauses the program's execution each time the `math` actor receives a message (before processing it). In Voyager, this is called a *message receiver breakpoint*; its semantics are shown in section 6. With this breakpoint activated, we run the program again in Voyager.

Figure 9(a) shows the new reduction graph, with the execution paused once the breakpoint was reached. The blue nodes denote the state of a running debugger executing the program. When the message receiver breakpoint on the `math` actor is reached, in one of the universes the debugger halts the execution (in that universe) and highlights the node in pink. As a result, the evaluation of the underlying AmbientTalk program pauses. The magnifier glass shows the pink node representing the triggering of the `Message-Receiver-Breakpoint` rule (later detailed in section 6). At this point in the execution, a developer can click on the node to inspect the state, resume execution, or execute one of the step commands. The debugging operations applicable at this point are accessible by means of a radial menu.

For this example, we make Voyager step to the next turn of the `math` actor. Figure 9(b) shows the resulting graph after applying that stepping command. The first pink node in fig. 9(b) corresponds to the node shown with the magnified glass in fig. 9(a). Notice that the dashed lines are used to indicate user interaction. The new graph shows how the debugger stops again at all possible universes in which the `math` actor receives the second message.

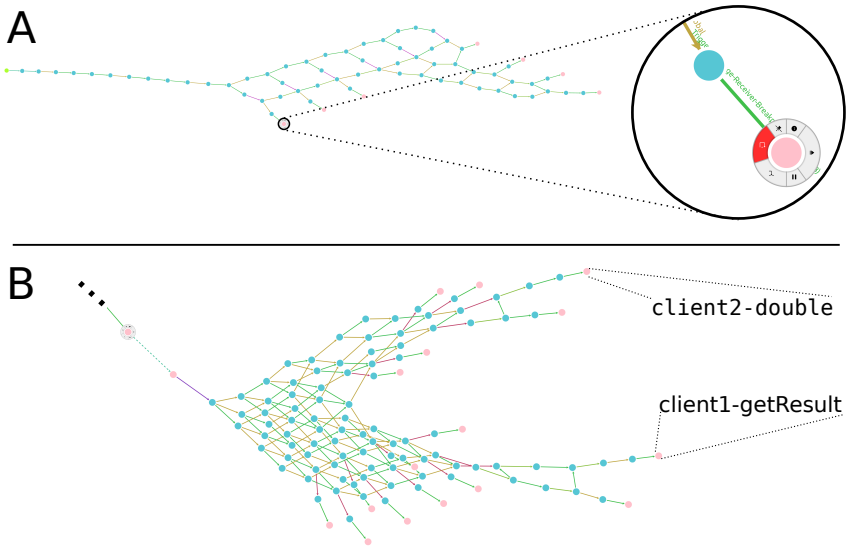

Figure 9 A debugging session in Voyager for the program displayed in listing 1.

The initial expectation of a developer may be that the next message in the mailbox of the `math` actor will always be the `getResult` message. However, in fig. 9(b) we see that there are two possible kinds of universes the base level program can evolve to. Inspecting the actors inbox reveals that in some universes the next processed message is from `client2`. As shown in the figure the top universe corresponds to the interleaving in which the `double` message from `client2` arrives first, and the bottom universe corresponds to the interleaving in which the `getResult` message from `client1` correctly arrives after the doubling message. At this point, a developer sees that the initial expectation does not hold and a fix can be developed to account for this bad interleaving.

One might be tempted to think that the program could be debugged by a traditional concurrent debugger using a deterministic message order. While single-stepping individual messages in a deterministic pattern would create a deterministic message order, traditional concurrent debuggers only keep track of one universe. Because such a debugger simply picks one of many universes determined by the execution order of messages, and it may not be the universe in which the bug manifests. Hence, traditional concurrent debuggers do not avoid the probe effect. In contrast, multiverse debugging allows developers to explore all possible non-deterministic execution paths of a concurrent application. In order to steer the state exploration, Voyager provides query facilities that we detail below.

5.2.3 Querying the state graph

As previously mentioned, the Voyager debugger stores the state graph in a graph database which we can use to query the state of the multiverse graph. The left panel of the Voyager UI (fig. 8) has a button to create new queries. Figure 10(a) shows the original graph for the program. Figure 10(b) shows a more simplified view on the multiverse after applying a query to the original graph that filters all paths except the shortest path from the start node to all end nodes, i.e., nodes that cannot be reduced any further.

Listing 2 shows the code for the query that generates the simplified graph shown in fig. 10(b) with the shortest path to the three possible end states of our sample problem. The

```
1  // make an array containing all stuck nodes
2  LET stuckNodes = (FOR n in @@nodes FILTER n._stuck == true RETURN n)
3
4  // find the path to each of them and join the results
5  LET path = FLATTEN(
6      FOR target IN stuckNodes
7          FOR n,e IN OUTBOUND SHORTEST_PATH
8              @start TO target._id GRAPH @graph
9                  RETURN {n,e})
10
11 // convert to the needed format
12 RETURN {
13     edges: (FOR d in path FILTER d.e != null RETURN DISTINCT d.e),
14     nodes: (FOR d in path FILTER d.n != null RETURN DISTINCT d.n)
15 }
```

Listing 2 AQL query to search the shortest path from the start state to all possible end states of a program.

used query language is AQL[4] using Bind parameters[5] (`@start, @graph, and @@nodes`). The query computes the shortest path to all end states as follows:

- First, it finds all stuck nodes, by querying the database for nodes that have been marked as *stuck*. The `FOR` operation returns an array of the values that have been returned by `RETURN` in its body.
- Second, line 5 to line 9 find the shortest path from the start node to each stuck node using ArangoDB's `SHORTEST_PATH`. This primitive returns an array of the nodes and edges on the shortest path between two nodes in a graph. Since the `SHORTEST_PATH` is used for each of the stuck nodes, we end up with an array of arrays which is then flattened. The `path` variable now holds an array of nodes and edges on the shortest path from the start node to a stuck node.
- Finally, in order to visualize the paths, this array is converted to an object that lists the edges and nodes to be displayed separately (line 12 to line 15).

The breakpoints and stepping operations combined with the query facilities of Voyager provide an interactive experience of browsing the multiverse graph of a program to find the root cause of bugs. It is important to note that in contrast to static analyses, a multiverse debugger allows developers to explore and query states of the concrete program execution interactively. This enables developers to focus on relevant elements and thereby directly steer the state exploration.

6 Syntax and Operational Semantics of the Voyager Multiverse Debugger

In this section we finally apply step 2 of the multiverse debugging recipe and describe the syntax and operational semantics of our multiverse debugger, Voyager. We first describe the general strategy of the debugger to be able to debug AmbientTalk programs, and we then detail the elements of the *debugger configuration* of Voyager (step 2a) and the key mechanisms for supporting breakpoints and stepping operations as the one described in the previous section (step 2b).

[4] https://docs.arangodb.com/3.4/AQL/Fundamentals/Syntax.html
[5] https://docs.arangodb.com/3.4/AQL/Fundamentals/BindParameters.html

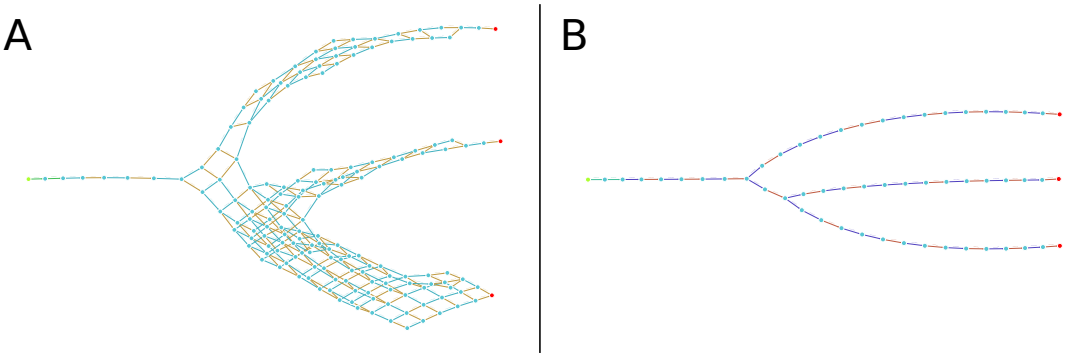

Figure 10 Application of a shortest-path-to-end-states query to the program displayed in listing 1.

6.1 Overview of the Debugger Semantics

The Voyager debugger keeps track of both the state of the underlying AmbientTalk program and its own state. The semantics of the debugger consists of a set of reduction rules which transition from one debugger state to the next one. In order to model the catalog of breakpoints and stepping operations that we proposed in [28], we define the debugger state $\mathcal{D}$, which consists of six elements, B_p, B_c, d_s, C, A_s, K.

- The first two elements B_p and B_c are respectively the list of pending breakpoints and the list of already checked breakpoints.
- To keep track of which action the debugger is performing, the debugger configuration contains a debugger state d_s for representing the state *run* and *pause*. When the debugger is in the *run* state it verifies whether there is an applicable breakpoint. When a breakpoint hits, the debugger transitions itself to the *pause* state and halts execution.
- To model the possible debugging operations offered to the user, e.g. stepping, resume, and pause, the debugger state contains a list of commands C.
- To keep track of the state of the actor the debugger configuration contains a map A_s.
- Finally, K is the state of the actor configuration being debugged, i.e. the state of the AmbientTalk program.

The general form of the reduction rules of the Voyager debugger consists of transitions between debugger states:

$$\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B'_p, B'_c, d'_s, C', A'_s, K'\rangle$$

Note that the transition relation of the debugger is denoted by $\rightarrow_d$ while the transition rules of the base language are defined as $\rightarrow_k$. The evaluation strategy of the debugger consists of traversing the list of pending breakpoints B_p one-by-one from left to right, moving the debugger to a stopped state when a breakpoint hits. When a breakpoint does not apply for the current state of the actor configuration, it is moved from the list of pending breakpoints to the list of checked ones. When there are no pending breakpoints left the debugger instructs the actor configuration to take one step and swaps the checked breakpoints with the pending breakpoints, this continues till either a breakpoint is hit or an end state is reached.

6.2 Syntax of the Debugger Semantics

Figure 11 shows the semantics of two elements that we needed to extend in the AmbientTalk semantics AT^f presented in fig. 6.

- The first element corresponds to the message entity m, which we extended with an id ι_m to identify the message.
- The second element corresponds to the send expression $e \leftarrow_{id} m(\overline{e})$ that we extended also with an identifier id to determine which message is breakpointed. This identifier is needed because different types of breakpoints can be set on the same message.

m $\in$ **Message**	::=	$\mathcal{M}\langle \iota_m, v, m, \overline{v} \rangle$	Messages
$e \in E \subseteq$ **Expr**	::=	$\ldots \mid e \leftarrow_{id} m(\overline{e})$	Runtime Expressions

Figure 11 Extended semantic entities in AT^f for debugging in Voyager calculus.

$d \in$ **Debugger**	::=	$\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K \rangle$	Debugger configurations
$B_p \in$ **Pending breakpoint**	::=	$\overline{b_u \mid b_t}$	Pending breakpoints
$B_c \in$ **Checked breakpoint**	::=	$\overline{b_t}$	Checked breakpoints
$d_s \in$ **Debugger state**	::=	run | pause	Debugger states
$C \in$ **Command**	::=	$\overline{c}$	Commands
$A_s \in$ **Actor state map**	::=	$\overline{c_s}$	Actor state map
$b_u \in$ **User breakpoint**	::=	$\mathcal{B}\langle t_{ub}, \iota_i \rangle$	User Breakpoints
$b_t \in$ **Trigger breakpoint**	::=	$\mathcal{B}\langle t_{tb}, \iota_a, \iota_i \rangle$	Trigger Breakpoints
$c \in$ **C**	::=	$\mathcal{C}\langle t_c \rangle \mid \mathcal{C}\langle t_c, n \rangle$	Commands
$c_s \in$ **Current actor state**	::=	$\mathcal{CS}\langle \iota_a, a_s \rangle$	Current actor state
$a_s \in$ **Actor state**	::=	run | pause | hold | step n	Actor states
$t_{ub} \in$ **User breakpoint tag**	::=	msb | mrb	User breakpoint tags
$t_{tb} \in$ **Trigger breakpoint tag**	::=	mrb-trigger	Trigger breakpoint tags
$t_c \in$ **Command tag**	::=	step-next-turn ι_a | resume | pause	Command tags

$\iota_i \in$ **BreakpointId**

Figure 12 Semantic entities of the Voyager calculus.

Figure 12 shows an overview of the elements of the Voyager calculus. More concretely, it includes all the entities of the semantics that are needed by the six elements of the debugger configuration D, i.e. b_u, b_t, c, c_s, a_s, t_{ub}, t_{tb}, t_c.

- To define a breakpoint b_u we use a two-element tuple consisting of a breakpoint tag t_{ub} and an expression id ι_i.
- Additionally, we define breakpoints at the level of the debugger semantics, i.e. breakpoints which are defined by the semantics itself rather than by the developer debugging a target program. We call these breakpoints *trigger* breakpoints to distinguish them from the *user* ones aforementioned. A trigger breakpoint b_t consists of a tuple of three elements, a breakpoint tag t_{tb}, an actor id ι_a, and an expression id ι_i.

- A command c is defined by a tag t_c. In the case of a step to next turn we need to define also the number of steps n the command needs to take in the evaluation of the program, i.e. $\mathcal{C}\langle c, step\ n\rangle$.
- The map of actors A_s keeps a list of pairs $\mathcal{CS}\langle \iota_a, a_s\rangle$ consisting of the id of the actor ι_a and the current state a_s.
- An actor can be in *run*, *pause* or *hold* state. In addition, an actor can have a state *step n*.
- The breakpoint tags t_{ub} indicate the tags a user can identify when defining a breakpoint.
- The trigger breakpoint tags t_{tb} correspond to the tags built-in in the semantics to actually trigger the breakpoint.
- The command tags t_c refer to the debugging commands the user can specify to debug the program, i.e. several stepping operations and resume/pause commands.

6.3 Operational Semantics of the Voyager Debugger

Having defined the syntax of the Voyager debugger and the debugger configuration (step 2a of the multiverse debugging recipe), we can now define the semantics of the debugger operations that Voyager offers to developers to interactively explore the target program (step 2b).

The reduction rules of Voyager can be separated in five groups:

1. Reduction rules for modeling the connection of the debugger with the base level language (cf. section 6.3.1)
2. Reduction rules for breakpoints (cf. section 6.3.2), including rules needed to model breakpoints which require trigger breakpoints for their functioning.
3. Bookkeeping reduction rules (cf. section 6.3.3), i.e. rules that are related to the actor state when breakpoints are not applicable and when new actors are created.
4. Reduction rules for the stepping operations (cf. section 6.3.4), consists of the rules for stepping commands that can be applied on the level of messages, futures, and turns.
5. Reduction rules for other debugging commands (cf. section 6.3.5), i.e. rules that will resume and pause the program's execution.

For brevity, the following sections focus on the rules required to define the semantics for the message receiver breakpoint and the *step to next turn* command employed in the debugging session shown in section 5.2. The complete set of reduction rules is included in appendix A.

6.3.1 Connection with the Base Level Language

Recall that the semantics of a multiverse debugger is defined in terms of the underlying base language semantics, AT^f in the case of Voyager. Two reductions rules (shown below) govern the connection of Voyager's semantics with AT^f: CEL-STEP-GLOBAL and CEL-STEP-LOCAL. The CEL-STEP-GLOBAL rule transitions the actor configuration K to the actor configuration K' by applying the global AmbientTalk reduction relation ($\rightarrow_k$). This relation governs all the actor transitions rules that affect two or more actors, i.e. sending asynchronous messages and creating new actors. The CEL-STEP-LOCAL rule, on the other hand, non-deterministically picks an actor a from the actor configuration K and transitions it to an actor a' by applying the local multi-step AmbientTalk relation $\xrightarrow{*}_a$. This reduction relation applies one or more single-step local reduction which can be applied to the actor, all these single-step reductions are deterministic. Finally, we require that the actor which we transition is in a local running

state and update it accordingly, i.e. when the actors local state is (*step n*) the *update* meta-function will update the actors state to (*step* $n-1$).

Both the CEL-STEP-GLOBAL and CEL-STEP-LOCAL rule can only be triggered when all the pending breakpoints are checked. Note that after taking a step in AT^f, the checked breakpoints and the pending breakpoints are swapped. At certain points during the execution it could be that both CEL-STEP-GLOBAL and CEL-STEP-LOCAL are applicable at the same time. This is intentional and is part of the non-deterministic nature of executing the AmbientTalk semantics that we want to capture in the debugger.

(CEL-STEP-GLOBAL)

$$\frac{K \rightarrow_k K' \qquad not-applicable-add-new-actor}{\mathcal{D}\langle(), B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \text{run}, C, A_s, K'\rangle}$$

(CEL-STEP-LOCAL)

$$\frac{K = K' \dot{\cup} \{a\} \qquad a \xrightarrow{*}_a a' \qquad A'_s = update(A_s, a) \qquad not-applicable-add-new-actor}{\mathcal{D}\langle(), B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \text{run}, C, A'_s, K' \dot{\cup} a'\rangle}$$

6.3.2 Reduction Rules for Breakpoints

As mentioned before, the Voyager semantics features two families of breakpoints: *user breakpoints* denote breakpoints that are activated by the user while *trigger breakpoints* denote breakpoints generated by the debugger. As an example of user breakpoint consider the *message receiver breakpoint*, which we explained in the debugging session shown in section 5.2. It halts execution of an actor before it processes a message (identified by a unique id).

Generally, we only know during program execution which actor hosts the receiver object of a message. Therefore, the debugger monitors the program and inserts a new trigger breakpoint when the id of the receiver actor becomes known. The trigger breakpoint is used by the debugger semantics to later halt the execution when the message is actually received at the receiver side.

The SAVE-MRB reduction rule below governs the semantics of transforming a message receiver breakpoint into a trigger message receiver breakpoint. When the message is about to be sent the user breakpoint $\mathcal{B}\langle mrb, \iota_i\rangle$ that is in the list of pending breakpoints, the SAVE-MRB is triggered if the actor id of the breakpoint corresponds to the actor id of the receiver actor. In this case, the breakpoint is removed from the pending list and a trigger breakpoint $\mathcal{B}\langle mrb - trigger, \iota_{a'}, \iota_i\rangle$ is added in the list of checked breakpoints. Note that the sender and receiver actors of that message continue with *run* state, but the addition of the trigger message breakpoint will make the execution of the debugger pause at the receiver actor (the actor id of which is included in the trigger breakpoint itself).

(SAVE-MRB)

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, e_\Box[\iota_{a'}.\iota_o \leftarrow_{\iota_i} m(\overline{v})]\rangle \in K}{\mathcal{D}\langle \mathcal{B}\langle mrb, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle mrb - trigger, \iota_{a'}, \iota_i\rangle, \text{run}, C, A_s, K\rangle}$$

The next reduction rule is TRIGGER-MRB and it governs the semantics of the trigger breakpoint added for a message receiver breakpoint. Back in the example debugging session, fig. 9 showed that the triggering of this rule resulted in the pink node under the magnifying

glass. In the trigger breakpoint the id of the actor ι_a is saved to identify the actor the user wants to halt, and the ι_i is saved to identify in which message. When the message arrives in the queue of the receiver actor, the trigger breakpoint is removed from the pending list B_p and the debugger and the receiver actor changes its state to *pause*. Note that the receiver actor cannot process local operations but it can execute global ones, e.g. receive a new message from another actor. The two operations of the message receiver breakpoint for saving the information needed when the message is about to be sent and triggering the breakpoint are shown in appendix A, i.e. SAVE-MRB and TRIGGER-MRB.

(TRIGGER-MRB)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, v\rangle \in K \qquad A'_s = A_s + \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle \mathcal{B}\langle mrb - trigger, \iota_a, \iota_i\rangle \cdot B_p, B_c, run, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

6.3.3 Bookkeeping Reduction Rules

For each of the breakpoint triggering rules there should be a rule which instructs the debugger to move the breakpoint to the list of checked breakpoints when the breakpoint does not hit. Instead of listing all these individual rules we compressed them into one rule called BREAKPOINT-NOT-APPLICABLE which should be triggered when the breakpoint at the head of the list is not applicable. The BREAKPOINT-NOT-APPLICABLE rule is shown Appendix A.

Appendix A also includes the ADD-NEW-ACTOR reduction rule for the creation of new actors. This rule basically updates the A_s map when an actor is created.

6.3.4 Reduction Rules for Stepping Operations

Similarly to the formalisation of the message sender breakpoint, some stepping commands need to be encoded with several reduction rules. For example, the step-next-turn employed in debugging session in section 5.2, is formalised with two reduction rules: PREPARE-STEP-NEXT-TURN and TRIGGER-STEP-NEXT-TURN. The PREPARE-STEP-NEXT-TURN rule is triggered when the debugger is in the paused state and transitions a particular actor with id ι_a from the paused state to the (*step* 1) state indicating that the actor is allowed to take exactly one local step (cf. CEL-STEP-LOCAL in section 6.3.1).

The TRIGGER-STEP-NEXT-TURN rule is triggered when a particular actor with id ι_a is in the state (*step* 0). When this rule is triggered, the debugger moves form the *run* state to the *paused* state. At the same time, the local actor state is also changed to the *pause* state.

(PREPARE-STEP-NEXT-TURN)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, e\rangle \in K \qquad A'_s = A_s \dot{\cup} \{\mathcal{CS}\langle \iota_a, (step\ 1)\rangle\}}{\mathcal{D}\langle B_p, B_c, \text{pause}, (StepNextTurn\ \iota_a) \cdot C, A_s \dot{\cup} \mathcal{CS}\langle \iota_a, (pause)\rangle, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, (StepNextTurn\ \iota_a) \cdot C, A'_s, K\rangle}$$

(TRIGGER-STEP-NEXT-TURN)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, v\rangle \in K \qquad A'_s = A_s \dot{\cup} \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle B_p, B_c, \text{run}, (StepNextTurn\ \iota_a) \cdot C, A_s \dot{\cup} \mathcal{CS}\langle \iota_a, (step\ 0)\rangle, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

Note that other breakpoints and stepping commands can be encoded in a similar vain as we have shown for the message receiver breakpoint and step to next turn. Some of these breakpoints such as the message sender breakpoint (cf. appendix A) are easier because they

do not require any bookkeeping, i.e. all the information to pause the execution is known at the start of the program.

6.3.5 Reduction Rules for Basic Debugging Commands

Finally, we show below the rules which govern basic debugging commands to control the execution of a program, namely pause and resume. The RESUME-EXECUTION rule guarantees that the execution of the program continues from any pause state of the debugger. As such, the debugger state transits from *pause* to *run*. The rule updates the state of the local actors to *run*.

The PAUSE-EXECUTION rule halts the execution of all actors in the actor configuration, transitioning the debugger state from *run* to *pause*. The rule updates the state of the local actors to *pause*.

(RESUME-EXECUTION)

$$\frac{A'_s = run(A_s)}{\mathcal{D}\langle B_p, B_c, \text{pause}, Resume \cdot C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, C, A'_s, K\rangle}$$

(PAUSE-EXECUTION)

$$\frac{A'_s = pause(A_s)}{\mathcal{D}\langle B_p, B_c, \text{run}, Pause \cdot C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

6.3.6 Discussion

There are a number of design decisions and limitations worth discussing. First, in an early prototype of the debugger semantics [30], we did not separate the global from the local AT^f reduction rules. This turned out to be problematic because it makes it hard to pause a specific actor from processing messages while still allowing it to receive messages in its inbox. Separating the global from the local semantics simplified the semantics significantly.

Second, the early prototype used the single-step operational semantics to transition the local actor semantics [30], while in the final version reported here, we are using a multi-step relation. As previously mentioned, in the actor model the only points where the non-determinism matters is when messages are being exchanged between the actors. However, when using the single-step local reduction relation a lot of additional and irrelevant non-determinism is introduced. This made working with the Voyager debugger very tedious and reduced its usefulness for larger programs. By switching to the local multi-step relation the amount of states being shown to the end user is significantly reduced while the non-deterministic behavior due to message passing is completely preserved.

Finally, it is worth noting that even though the multi-step relation alleviates the problem of growing number of states, the number of states still grows depending on the program size, as previously mentioned. Further research is needed to investigate ways to reduce the number of states without removing relevant sources of non-determinism in the program. To this end, advances in the context of static techniques like symbolic execution and model checking can be employed as starting point (cf. Section 8.2). We believe that with the current hardware evolution of multicore machines, the size of programs which can be debugged with multiverse debugging is steadily growing as well. At this point, we have used the Voyager tool to debug programs of the size of dining philosophers. Further research is also needed to investigate techniques to guide the exploration of the state graph, e.g. novel stepping semantics that work

at the level of universes. Of course, applying this technique to industrial-strength languages will also required further work. But the goal would be that a traditional breakpoint-based debugger can be a foundation for such multiverse debugging.

7 Proof of Non-Interference

In this section we provide a proof of *non-interference* for the semantics of the Voyager debugger. More specifically, we prove observational equivalence between the debugger and the base language semantics. Intuitively, this means that any execution of the Voyager debugger corresponds to an execution of an AmbientTalk program, and any execution of an AmbientTalk program is observed by Voyager. Formally,

▷ Theorem 1. (Equivalence of evaluation steps) Let K be an actor configuration in the AT^f semantics, for which there exists a transition to an actor configuration K'. Let D be a debugging configuration for K and B_p, B_c, d_s, C, A_s elements of D such that the commands C resume all the paused actors then:

$$(K \rightarrow_k K') \iff$$

$$(\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K\rangle \rightarrow_{d_k} \mathcal{D}\langle B'_p, B'_c, d'_s, C', A'_s, K'\rangle)$$

The left handside of the biconditional relation represents the evaluation of the program in the AmbientTalk semantics AT^f, i.e. the configuration of actors K, to another program state K'. Where $\rightarrow_k$ corresponds to the evaluation regarding the reduction rules of the base language.

The right handside of the biconditional relation represents the evaluation of the program in the debugger semantics $\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K\rangle$, which yields in an another debugger configuration $\mathcal{D}\langle B'_p, B'_c, d'_s, C', A'_s, K'\rangle$. Where $\rightarrow_{d_k}$ represents one or more evaluation *steps* taken by the debugger transition rules in K, until the debugger configuration $\mathcal{D}\langle B'_p, B'_c, d'_s, C', A'_s, K'\rangle$ is reached.

To prove the biconditional relation of Theorem 1 we divide our proof in two parts, which corresponds to the two implications of the relation.

Implication 1. An evaluation step in the AmbientTalk semantics implies equivalent evaluation steps in the debugger semantics

Proof sketch: We proceed by induction over the set of pending breakpoints B_p.

Base case: In this case the list B_p is empty. Either the actor is in running or in the paused state. By assumption, when the debugger is in a pause state, the commands C will un-pause the debugger.

In general a step in the base-level language can be done in two modes. In case the base level semantics performed a global reduction, there is a corresponding transition in the debugger by taking a step with CEL-STEP-GLOBAL. Similarly if it was a local rule, there is a possible transition with the CEL-STEP-LOCAL rule.

Inductive case: Assuming that there is list of pending breakpoints B_p leading to the actor configuration K'. When adding one breakpoint to that list we need to consider two cases. Either the breakpoint is applicable or it is not. When the breakpoint does not apply the corresponding NOT-APPLICABLE-BREAKPOINT rule will move the breakpoint to the list of

checked breakpoints and the induction hypothesis applies. In the other case the breakpoint applies, in which by assumption the commands C will transition the debugger back to the run state at which point the induction hypothesis can be applied.

Implication 2. An evaluation step in the debugger semantics implies an equivalent evaluation step in the AmbientTalk semantics

Proof sketch: By construction, the only two rules CEL-STEP-LOCAL and CEL-STEP-GLOBAL where the debuggers K field transitions to K' directly rely on the underlying AmbientTalk semantics.

8 Related Work

To the best of our knowledge, multiverse debugging is the first debugging approach that allows developers to interactively browse all execution paths of parallel programs. In this section, we compare our work to other efforts on formalizing debuggers for actor languages and other programming paradigms. We also relate our work to static analysis techniques for debugging non-deterministic programs such as model checking and symbolic execution.

8.1 Formal specifications for debuggers

The first formal specification for debuggers was proposed by Da Silva [31]. He used a structural operational semantics that considers a debugger as a system, which transitions from one state to another using an evaluation history. He defines the semantics of his debugging approach on top of a deterministic relation specification of a programming language. To prove debugger correctness, Da Silva presented a proof of equivalence between two debugger approaches. This work served as inspiration for multiverse debugging, but we focus on proving the equivalence between the base language and the debugger, i.e., their non-interference. While Da Silva does not address non-deterministic languages, he argues that non-repeatability of evaluation can be avoided by recording all choices where more than one evaluation rule could be chosen. However, to the best of our knowledge Da Silva never put this theory into practice. Our approach differs from Da Silva by embracing the non-deterministic nature of the base language and using it to derive a non-deterministic debugger.

Bernstein et al. [32] developed a debugging semantics based on transitions for a deterministic functional programming language. The evaluation steps in the debugging session correspond to executing subexpressions of the program. Similar to Voyager, developers can select terms (represented as nodes in the graph) corresponding to the program states and create new programs from them to debug. Bernstein et al. did not apply their techniques to non-deterministic languages.

In the context of distributed systems, Ferrari et al. [33] proposed a debugging calculus for mobile ambients. Similar to our approach, they model a debugger as an extension of the operational semantics of an underlying programming language. Their operational semantics is a causal model of behaviors which they represent using Petri nets. In a later work, Ferrari et al. [34] proposed Causal Nets which allows the developer to query a causal message graph generated by the execution of a set of distributed processes. We have experimented with converting the multiverse execution graph into a Petri net, but due to the size of the execution graphs the resulting Petri nets offered few additional insight into the program behavior.

In the context of algorithmic debugging, Luo et al. [35] proposed a formal model of tracing for functional programs. The authors proved correctness of evaluation dependency trees to

identify faulty nodes, i.e. a node with erroneous computation. They consider correctness when the debugging algorithm detects a faulty node that matches the answer of the user. In contrast to multiverse debugging, this approach does not show an exploration of different non-deterministic paths, but the exploration of one path of execution of a functional program based on a trace. Similarly, Caballero et al. [36] uses a technique of algorithmic debugging to detect liveness issues in Erlang programs. Their approach can analyze sequential and concurrent programs using a calculus based on proofs to build execution trees.

Li et al. [37] introduced a formal semantics for debugging synchronous message-passing programs, e.g. MPI, Occam, and JCSP. They propose a structural operational semantics for a tracing procedure and bug/fix locating procedure. The goal of these procedures is to record useful information that helps to build the execution history of the program. More concretely, the tracing procedure records to one execution path in the evaluation of the program, ignoring non-determinism. In contrast, our approach considers all possible execution paths.

Giachino et al. [38] provide a causal consistent reversible semantics for the μOz language, featuring thread-based concurrency and asynchronous communication over ports. These semantics however, do not explore different paths of the execution of a concurrent program. Following the idea of reversible semantics, Lanese et al. [39] proposed a causal consistent reversible debugger for Erlang processes. More concretely, they use a reversible semantics for Erlang [40], in which they record a history of all the computed expressions, corresponding to each execution step. In contrast, our semantics only keep track of the state of actors and breakpoint information. In addition, the rules related to the reversible semantics are said to be non-deterministic, but no concrete exploration examples of different execution paths are included in the paper.

In the context of Petri nets, Van Mierlo et al. [41] proposed a debugging tool for observing erroneous states of non-deterministic behavior. The tool takes a model of a system as input, and builds a *Petri net reachability graph* which can be debugged in an interactive way. Similar to our approach, they provide online debugging operations, e.g., breakpoints and stepping, to explore specific program states. Multiverse debugging however, takes as input programs based on the operational semantics of the programming language and allows to debug the *execution graph* of the program.

8.2 Static Analysis Techniques

Since multiverse debugging allows developers to explore all possible paths of execution of an application, it can be considered closely related to static analysis techniques such as model checking and symbolic execution. Below, we provide an overview of such techniques, with a focus on actor-based approaches, and compare them to multiverse debugging.

Model checking. Model checking is a static technique for automatically verifying correctness properties of programs given a specification of the property. It has been studied for thread-based concurrency models to verify safety requirements such as the absence of deadlocks [42, 43, 44]. Ignoring recursion or relying on finite state models avoids undecidability problems due to synchronization as well as applying bounded analysis using testing or bounded model checking techniques [45].

Model checking has also been explored in the context of actor-based languages to verify properties like boundedness of actor mailboxes, and incorrect interleavings of messages. In the context of Erlang programs, Fredlund et al. [4] proposed a model checker that verifies boundedness of their mailboxes and process spawning. Other approaches have focused on verifying the property of mutual exclusion in Erlang programs [46, 47] or to analyze message

interleavings between Erlang processes [48]. There exist model checkers for other actor-based languages such as Basset [49], a model checker that can analyze message schedules in actor-based programs written in Scala and ActorFoundry library for Java. Tasharofi et al. proposed an algorithm based on partial order reduction to prune the number of message interleavings in Scala and ActorFoundry programs [50]. Additionally, a more theoretic approach uses model checking to verify actors behavior based on compositional analysis of schedules [51].

Model checking tools excel at finding a set of bugs of which the programmer knows exactly how to describe them. Multiverse debugging is meant for debugging and interactively exploring the state space in order to discover bugs for which the programmer may not have a good description. Similar to model checking, multiverse debugging can suffer from the state explosion problem. As mentioned before, our approach does not blindly explore all the possible states but lets the developer decide which states to explore next, either explicitly or by using multiverse breakpoints, which makes multiverse debugging similar to bounded model checking [19]. Other techniques in model checking have been proposed to handle the state explosion problem including symbolic model checking with binary decision diagrams, partial order reductions and counter example guided abstraction refinement [15].

Symbolic execution. Symbolic execution is a static technique to test whether certain predefined properties can be violated by a program [52]. A key idea in symbolic execution is to explore programs taking as input *symbolic* values rather than concrete ones. According to [53], most symbolic execution techniques can be categorized in either techniques that create and search in a *subset* of the concrete search space (i.e. an under-approximation), or techniques that create and search in a *superset* of it (i.e. an over-approximation). Under-approximation techniques lead to false negatives (i.e. missing real errors) but are preferred over over-approximation techniques because the latter ones introduce false positives (report errors that do not exist) and do not scale as well because of the cost of handling infeasible states. Many research efforts on symbolic execution for multi-threaded programs have focused on improving the efficiency of over-approximation techniques [54, 55].

Concolic execution is a mix of concrete and symbolic execution which has been throughly studied for thread-based programs [52, 56, 57, 58]. In the context of actors, much work has also focused on concolic execution. Sen et al. [59] proposed a testing algorithm based on concolic execution together with runtime partial order reduction for detecting deadlock states in a language related to the actor semantics. Albert et al. [60] developed a test case generation framework which avoids redundant computations when exploring the order of several tasks. More recently, Albert et al. [61] proposed a variant of a dynamic partial order reduction algorithm which can be used when searching for deadlocks. Their algorithm aims to reduce state space exploration by distinguishing between two sources of non-determinism: actor selection and task selection. Recently, Li et al. proposed an exploration of the state space using symbolic execution based on heuristics that consider paths where only interact with a small number of actors [18].

Like multiverse debugging, symbolic execution can explore all possible execution paths of a program. While the use of abstract states alleviates the state explosion problem, that may imply missing execution paths (i.e. universes) containing a bug due to under approximation. In contrast to symbolic execution, multiverse debugging models the program execution only with concrete values, and can not miss executions paths. While multiverse debugging does not solve the state explosion problem, developers can pause and resume the program, and select themselves the different execution paths to explore. In the context of thread-based programs, some work in symbolic execution has studied solutions for reducing the complexity of path

exploration based on merging paths [16], state pruning [62], probabilistic computations [17] and search heuristics [18].

9 Conclusion

We proposed multiverse debugging as a new debugging approach to tackle the problem of non-determinism in concurrent and parallel programs. Contrary to traditional concurrent debugging approaches, multiverse debugging allows developers to explore non-deterministic execution paths corresponding to the evaluation of a program. This is meant to simplify the reproduction and inspection of concurrency bugs, because it removes chance and probability from the equation of hitting the problematic interleaving. Instead, an execution path that can lead to a bug can be explored interactively and a developer can see the state in all possible universes.

To build a multiverse debugger, we provided a recipe with two steps. First, we need to define the operational semantics of a non-deterministic base language. Second, we need to define a debugger configuration and its operational semantics in terms of the base language semantics. In this paper, we have applied this recipe to provide a proof-of-concept multiverse debugger for actor-based programs called *Voyager*. Voyager uses as input a PLT-Redex program implemented in the AmbientTalk operational semantics and gives as output the reduction graph corresponding to all possible universes of the program. To make this exploration manageable, the graph can be explored interactively as one would do in a classic breakpoint-based debugger. Besides providing the semantics of a multiverse debugger, we also demonstrate that there is no interference between the debugger and the target program by proving non-interference. This shows that the debugger is probe-effect free.

We consider multiverse debugging to be a good basis for further experiments in debugging non-deterministic concurrent programs. Voyager's debugging operations merely scratch the surface of a new branch in debugging tools. The main open research question is how to make multiverse debugging practical for complex concurrent applications. While we believe that the interactive nature alleviates some of the scalability issues of static analyses, exploring the multiverses of larger programs can become unwieldy. Thus, research is needed to guide the exploration of the state graph, e.g. novel stepping semantics that work at the level of universes. Furthermore, the technique needs to be applied to a concrete language implementation beyond a PLT-Redex-based formalism. Efficient runtime techniques will be cornerstone to make it practical, but we may be able to leverage work of static analyses and back-in-time debugging [63].

References

References

1. C. E. McDowell, D. P. Helmbold, Debugging concurrent programs, ACM Computing Surveys (CSUR) 21 (4) (1989) 593–622. doi:10.1145/76894.76897.
2. M. Perscheid, B. Siegmund, M. Taeumel, R. Hirschfeld, Studying the advancement in debugging practice of professional software developers., Software Quality Journal 25 (1) (2016) 83–110. doi:10.1007/s11219-015-9294-2.
3. J. Gait, A probe effect in concurrent programs, Software: Practice and Experience 16 (3) (1986) 225–233. doi:10.1002/spe.4380160304.
4. L.-A. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, G. Chugunov, A verification tool for erlang., STTT 4 (4) (2003) 405–420. doi:10.1007/s100090100071.

5 M. Christakis, A. Gotovos, K. Sagonas, Systematic testing for detecting concurrency errors in erlang programs, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 154–163. doi:10.1109/ICST.2013.50.

6 S. Lauterburg, M. Dotta, D. Marinov, G. Agha, A framework for state-space exploration of java-based actor programs, in: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 468–479. doi:10.1109/ASE.2009.88.

7 K. Takeuchi, K. Honda, M. Kubo, An interaction-based language and its typing system, in: C. Halatsis, D. Maritsas, G. Philokyprou, S. Theodoridis (Eds.), PARLE'94 Parallel Architectures and Languages Europe, Springer Berlin Heidelberg, Berlin, Heidelberg, 1994, pp. 398–413. doi:10.1007/3-540-58184-7_118.

8 C. Boyapati, R. Lee, M. Rinard, Ownership types for safe programming: Preventing data races and deadlocks, in: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '02, ACM, New York, NY, USA, 2002, pp. 211–230. doi:10.1145/582419.582440.

9 C. Flanagan, S. N. Freund, Type-based race detection for java, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, ACM, New York, NY, USA, 2000, pp. 219–232. doi:10.1145/349299.349328.

10 T. Balabonski, F. Pottier, J. Protzenko, Type soundness and race freedom for mezzo, in: M. Codish, E. Sumii (Eds.), Functional and Logic Programming, Springer International Publishing, Cham, 2014, pp. 253–269. doi:10.1007/978-3-319-07151-0_16.

11 E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt, P. Y. H. Wong, Deadlock analysis of concurrent objects: Theory and practice, in: E. B. Johnsen, L. Petre (Eds.), Integrated Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 394–411. doi:10.1007/978-3-642-38613-8_27.

12 T. Van Cutsem, E. Gonzalez Boix, C. Scholliers, A. Lombide Carreton, D. Harnie, K. Pinte, W. De Meuter, Ambienttalk: programming responsive mobile peer-to-peer applications with actors, Computer Languages, Systems & Structures 40 (3-4) (2014) 112–136. doi:10.1016/j.cl.2014.05.002.

13 J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic model checking: 10$\hat{2}$0 states and beyond, in: LICS, IEEE Computer Society, 1990, pp. 428–439. doi:10.1016/0890-5401(92)90017-A.

14 A. Valmari, The state explosion problem, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, Ch. 9, pp. 429–528. doi:10.1007/3-540-65306-6_21.

15 E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Progress on the state explosion problem in model checking., in: R. Wilhelm (Ed.), Informatics, Vol. 2000 of Lecture Notes in Computer Science, Springer, 2001, pp. 176–194. doi:10.1007/3-540-44577-3_12.

16 C. Cadar, K. Sen, Symbolic execution for software testing: Three decades later, Commun. ACM 56 (2) (2013) 82–90. doi:10.1145/2408776.2408795.

17 K. S. Luckow, C. S. Pasareanu, M. B. Dwyer, A. Filieri, W. Visser, Exact and approximate probabilistic symbolic execution for nondeterministic programs., in: I. Crnkovic, M. Chechik, P. Grünbacher (Eds.), ASE, ACM, 2014, pp. 575–586. doi:10.1145/2642937.2643011.

18 S. Li, F. Hariri, G. Agha, Targeted test generation for actor systems., in: T. D. Millstein (Ed.), ECOOP, Vol. 109 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 8:1–8:31. doi:10.4230/LIPIcs.ECOOP.2018.8.

19 A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic model checking without bdds, in: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS '99, Springer-Verlag, Berlin, Heidelberg, 1999, pp. 193–207. doi:10.1007/3-540-49059-0_14.

20 J. McCarthy, A basis for a mathematical theory of computation, preliminary report, in: Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference, IRE-

AIEE-ACM '61 (Western), ACM, New York, NY, USA, 1961, pp. 225–238. doi:10.1145/1460690.1460715.

21 S. Mostinckx, T. Van Cutsem, S. Timbermont, E. Gonzalez Boix, E. Tanter, W. De Meuter, Mirror-based reflection in ambienttalk, Softw. Pract. Exper. 39 (7) (2009) 661–699. doi:http://dx.doi.org/10.1002/spe.v39:7.

22 C. Hewitt, P. Bishop, R. Steiger, A Universal Modular ACTOR Formalism for Artificial Intelligence, in: IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1973, pp. 235–245.

23 M. S. Miller, E. D. Tribble, J. Shapiro, Concurrency among strangers, in: International Symposium on Trustworthy Global Computing, Springer, 2005, pp. 195–229. doi:10.1007/11580850_12.

24 G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, E. Miranda, Modules as Objects in Newspeak, in: T. D'Hondt (Ed.), ECOOP 2010 – Object-Oriented Programming, Vol. 6183 of LNCS, Springer, 2010, pp. 405–428. doi:10.1007/978-3-642-14107-2_20.

25 S. Tilkov, S. Vinoski, Node.js: Using javascript to build high-performance network programs, IEEE Internet Computing 14 (6) (2010) 80–83. doi:10.1109/MIC.2010.145.

26 A. Igarashi, B. C. Pierce, P. Wadler, Featherweight java: a minimal core calculus for java and gj., ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450. doi:10.1145/503502.503505.

27 E. Gonzalez Boix, C. Noguera, W. De Meuter, Distributed Debugging for Mobile Networks , Journal of Systems and Software 90 (2014) 76–90. doi:10.1016/j.jss.2013.11.1099.

28 S. Marr, C. Torres Lopez, D. Aumayr, E. Gonzalez Boix, H. Mössenböck, A concurrency-agnostic protocol for multi-paradigm concurrent debugging tools., in: D. Ancona (Ed.), Proceedings of the 13th Symposium on Dynamic Languages, ACM, 2017, pp. 3–14. doi:10.1145/3133841.3133842.

29 C. Torres Lopez, S. Marr, E. Gonzalez Boix, H. Mössenböck, A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs, Springer International Publishing, Cham, 2018, Ch. 6, pp. 155–185. doi:10.1007/978-3-030-00302-9_6.

30 C. Torres Lopez, E. Gonzalez Boix, C. Scholliers, S. Marr, H. Mössenböck, 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, AGERE!'17, ACM, 2017, pp. 41–49. doi:10.1145/3141834.3141839.

31 F. Q. B. da Silva, Correctness proofs of compilers and debuggers: an approach based on structural operational semantics., Ph.D. thesis, University of Edinburgh, UK, british Library, EThOS (1992).

32 K. L. Bernstein, E. W. Stark, Operational semantics of a focusing debugger, Electronic Notes in Theoretical Computer Science 1 (1995) 13 – 31, mFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference. doi:http://dx.doi.org/10.1016/S1571-0661(04)80002-1.

33 G. Ferrari, E. Tuosto, A debugging calculus for mobile ambients, in: Proceedings of the 2001 ACM Symposium on Applied Computing, SAC '01, ACM, New York, NY, USA, 2001, p. 2. doi:10.1145/372202.380701.

34 G. L. Ferrari, R. Guanciale, D. Strollo, E. Tuosto, Debugging distributed systems with causal nets, ECEASST 14 (2008) 1–10. doi:10.14279/tuj.eceasst.14.190.181.

35 Y. Luo, O. Chitil, Proving the correctness of algorithmic debugging for functional programs., in: H. Nilsson (Ed.), Trends in Functional Programming, Vol. 7 of Trends in Functional Programming, Intellect, 2006, pp. 19–34.

36 R. Caballero, E. Martin-Martin, A. Riesco, S. Tamarit, Declarative debugging of concurrent erlang programs, Journal of Logical and Algebraic Methods in Programming 101 (2018) 22 – 41. doi:https://doi.org/10.1016/j.jlamp.2018.07.005.

37 H. Li, J. Luo, W. Li, A formal semantics for debugging synchronous message passing-based concurrent programs, Science China Information Sciences 57 (12) (2014) 1–18. doi:10.1007/s11432-014-5150-4.

38 E. Giachino, I. Lanese, C. A. Mezzina, Causal-consistent reversible debugging., in: S. Gnesi, A. Rensink (Eds.), FASE, Vol. 8411 of Lecture Notes in Computer Science, Springer, 2014, pp. 370–384. doi:10.1007/978-3-642-54804-8_26.

39 I. Lanese, N. Nishida, A. Palacios, G. Vidal, CauDEr: A Causal-Consistent Reversible Debugger for Erlang, in: J. P. Gallagher, M. Sulzmann (Eds.), Functional and Logic Programming, Vol. 10818 of FLOPS'18, Springer, Cham, 2018, pp. 247–263. doi:10.1007/978-3-319-90686-7_16.

40 N. Nishida, A. Palacios, G. Vidal, A reversible semantics for erlang., in: M. V. Hermenegildo, P. López-García (Eds.), LOPSTR, Vol. 10184 of Lecture Notes in Computer Science, Springer, 2016, pp. 259–274. doi:10.1007/978-3-319-63139-4_15.

41 S. Van Mierlo, H. Vangheluwe, Debugging non-determinism: a petrinets modelling, analysis, and debugging tool, in: CEUR workshop proceedings, Vol. 2019, 2017, pp. 460–462.

42 K. Havelund, T. Pressburger, Model checking java programs using java pathfinder, International Journal on Software Tools for Technology Transfer 2 (4). doi:10.1007/s100090050043.

43 M. Musuvathi, S. Qadeer, Iterative context bounding for systematic testing of multithreaded programs., in: J. Ferrante, K. S. McKinley (Eds.), PLDI, ACM, 2007, pp. 446–455. doi:10.1145/1250734.1250785.

44 M. Kokologiannakis, O. Lahav, K. Sagonas, V. Vafeiadis, Effective stateless model checking for c/c++ concurrency., PACMPL 2 (POPL) (2018) 17:1–17:32. doi:10.1145/3158105.

45 E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem (Eds.), Handbook of Model Checking., Springer, 2018. doi:10.1007/978-3-319-10575-8.

46 F. Huch, Verification of erlang programs using abstract interpretation and model checking, in: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP '99, ACM, New York, NY, USA, 1999, pp. 261–272. doi:10.1145/317636.317908.

47 E. D'Osualdo, J. Kochems, C. H. L. Ong, Automatic verification of erlang-style concurrency, in: F. Logozzo, M. Fähndrich (Eds.), 20th International Symposium on Static Analysis, SAS 2013, Springer, 2013, pp. 454–476. doi:10.1007/978-3-642-38856-9_24.

48 M. Christakis, A. Gotovos, K. F. Sagonas, Systematic testing for detecting concurrency errors in erlang programs., in: ICST, IEEE Computer Society, 2013, pp. 154–163. doi:10.1109/ICST.2013.50.

49 S. Lauterburg, R. K. Karmani, D. Marinov, G. Agha, Basset: A Tool for Systematic Testing of Actor Programs, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, ACM, New York, NY, USA, 2010, pp. 363–364. doi:10.1145/1882291.1882349.

50 S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, G. Agha, TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs, in: H. Giese, G. Rosu (Eds.), Formal Techniques for Distributed Systems: Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings, Springer, 2012, pp. 219–234. doi:10.1007/978-3-642-30793-5_14.

51 M. M. Jaghoori, F. S. de Boer, D. Longuet, T. Chothia, M. Sirjani, Compositional schedulability analysis of real-time actor-based systems., Acta Inf. 54 (4) (2017) 343–378. doi:10.1007/s00236-015-0254-x.

52 R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Comput. Surv. 51 (3) (2018) 50:1–50:39. doi:10.1145/3182657.

53 Y. Lin, Symbolic execution with over-approximation, Ph.D. thesis, The University of Melbourne (2017).

54 T. Bergan, D. Grossman, L. Ceze, Symbolic execution of multithreaded programs from arbitrary program contexts., in: A. P. Black, T. D. Millstein (Eds.), OOPSLA, ACM, 2014, pp. 491–506. doi:10.1145/2660193.2660200.

55 Y. Lin, T. Miller, H. Søndergaard, Compositional Symbolic Execution: Incremental Solving Revisited, in: A. Potanin, G. C. Murphy, S. Reeves, J. Dietrich (Eds.), APSEC, IEEE Computer Society, 2016, pp. 273–280. doi:10.1109/ASWEC.2015.32.

56 T. Avgerinos, A. Rebert, S. K. Cha, D. Brumley, Enhancing symbolic execution with veritesting., in: P. Jalote, L. C. Briand, A. van der Hoek (Eds.), ICSE, ACM, 2014, pp. 1083–1094. doi:10.1145/2568225.2568293.

57 P. Godefroid, N. Klarlund, K. Sen, Dart: Directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, ACM, New York, NY, USA, 2005, pp. 213–223. doi:10.1145/1065010.1065036.

58 K. Sen, G. Agha, Cute and jcute: Concolic unit testing and explicit path model-checking tools, in: T. Ball, R. B. Jones (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 419–423. doi:https://doi.org/10.1007/11817963_38.

59 K. Sen, G. Agha, Automated systematic testing of open distributed programs., in: L. Baresi, R. Heckel (Eds.), FASE, Vol. 3922 of Lecture Notes in Computer Science, Springer, 2006, pp. 339–356. doi:10.1007/11693017_25.

60 E. Albert, P. Arenas, M. Gómez-Zamalloa, Test case generation of actor systems., in: B. Finkbeiner, G. Pu, L. Zhang (Eds.), ATVA, Vol. 9364 of Lecture Notes in Computer Science, Springer, 2015, pp. 259–275. doi:10.1007/978-3-319-24953-7_21.

61 E. Albert, P. Arenas, M. Gómez-Zamalloa, Systematic testing of actor systems., Softw. Test., Verif. Reliab. 28 (3). doi:10.1002/stvr.1661.

62 S. Guo, M. Kusano, C. Wang, Conc-ise: Incremental symbolic execution of concurrent software, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, ACM, New York, NY, USA, 2016, pp. 531–542. doi:10.1145/2970276.2970332.

63 E. T. Barr, M. Marron, E. Maurer, D. Moseley, G. Seth, Time-travel debugging for javascript/node.js, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, 2016, pp. 1003–1007. doi:10.1145/2950290.2983933.

A Reduction Rules of the Operational Semantics of Voyager

In this appendix we give an overview of all the reduction rules of the Voyager debugger to debug the example application. We split the reduction rules into five groups:

1. Reduction rules for modeling the connection of the debugger with the base level language (cf. section 6.3.1)
2. Reduction rules for breakpoints (cf. section 6.3.2), including rules needed to model breakpoints which require trigger breakpoints for their functioning.
3. Bookkeeping reduction rules (cf. section 6.3.3), i.e. rules that are related to the actor state when breakpoints are not applicable and when new actors are created.
4. Reduction rules for the stepping operations (cf. section 6.3.4), consists of the rules for stepping commands that can be applied on the level of messages, futures, and turns.
5. Reduction rules for other debugging commands (cf. section 6.3.5), i.e. rules that will resume and pause the program's execution.

$$\frac{K \rightarrow_k K' \qquad not - applicable - add - new - actor}{\mathcal{D}\langle((), B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \text{run}, C, A_s, K'\rangle} \text{(CEL-STEP-GLOBAL)}$$

$$\frac{K = K' \dot\cup \{a\} \qquad a \xrightarrow{*}_a a' \qquad A'_s = update(A_s, a) \qquad not - applicable - add - new - actor}{\mathcal{D}\langle((), B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \text{run}, C, A'_s, K' \dot\cup a'\rangle} \text{(CEL-STEP-LOCAL)}$$

Figure 13 Reduction rules for connecting the debugger with the base language.

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, e_\square[\iota_{a'}.\iota_o \leftarrow_{\iota_i} m(\overline{v})]\rangle \in K \qquad A'_s = A_s + \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle \mathcal{B}\langle msb, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle} \text{(TRIGGER-MSB)}$$

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, e_\square[\iota_{a'}.\iota_o \leftarrow_{\iota_i} m(\overline{v})]\rangle \in K}{\mathcal{D}\langle \mathcal{B}\langle mrb, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle mrb - trigger, \iota_{a'}, \iota_i\rangle, \text{run}, C, A_s, K\rangle} \text{(SAVE-MRB)}$$

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, v\rangle \in K \qquad A'_s = A_s + \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle \mathcal{B}\langle mrb - trigger, \iota_a, \iota_i\rangle \cdot B_p, B_c, run, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle} \text{(TRIGGER-MRB)}$$

Figure 14 Reduction rules for breakpoints.

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, e_\square[\text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\}]\rangle \in K \qquad \mathcal{CS}\langle \iota_{new}, a_s\rangle \notin A_s}{\mathcal{D}\langle B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, C, A_s \cdot \mathcal{CS}\langle \iota_{new}, run\rangle, K\rangle} \text{(ADD-NEW-ACTOR)}$$

$$\frac{not - applicable - breakpoint}{\mathcal{D}\langle \mathcal{B}\langle t_{ub}, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle t_{ub}, \iota_i\rangle, \text{run}, C, A_s, K\rangle} \text{(NOT-APPLICABLE-BREAKPOINT[TRIGGER-MSB,SAVE-MRB,TRIGGER-MRB])}$$

Figure 15 Reduction rules for bookkeeping information about the program state needed for breakpoints and stepping operations.

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, e\rangle \in K \qquad A'_s = A_s \dot\cup \{\mathcal{CS}\langle \iota_a, (step\ 1)\rangle\}}{\mathcal{D}\langle B_p, B_c, \text{pause}, (StepNextTurn\ \iota_a) \cdot C, A_s \dot\cup \mathcal{CS}\langle \iota_a, (pause)\rangle, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, (StepNextTurn\ \iota_a) \cdot C, A'_s, K\rangle} \text{(PREPARE-STEP-NEXT-TURN)}$$

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, v\rangle \in K \qquad A'_s = A_s \dot\cup \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle B_p, B_c, \text{run}, (StepNextTurn\ \iota_a) \cdot C, A_s \dot\cup \mathcal{CS}\langle \iota_a, (step\ 0)\rangle, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle} \text{(TRIGGER-STEP-NEXT-TURN)}$$

Figure 16 Reduction rules for stepping operations.

$$\frac{A'_s = run(A_s)}{\mathcal{D}\langle B_p, B_c, \text{pause}, Resume \cdot C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, C, A'_s, K\rangle} \text{(RESUME-EXECUTION)}$$

$$\frac{A'_s = pause(A_s)}{\mathcal{D}\langle B_p, B_c, \text{run}, Pause \cdot C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle} \text{(PAUSE-EXECUTION)}$$

Figure 17 Reduction rules for basic debugging commands.