Advanced Debugging Techniques to Handle Concurrency Bugs in Actor-based Applications

Carmen Torres López

Dissertation submitted in fulfillment of the
requirement for the degree of Doctor of Sciences

June 29, 2021

Promotor:
Prof. Dr. Elisa González Boix, Vrije Universiteit Brussel

Jury:
Prof. Dr. Beat Signer, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Kris Steenhaut, Vrije Universiteit Brussel, Belgium
Prof. Dr. Philipp Haller, KTH Royal Institute of Technology, Sweden
Dr. Stéphane Ducasse, Inria RMoD Group, France

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

© 2021 Carmen Torres López

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789493079892

NUR 989

All rights reserved. No part of this publication may be produced in any form by print, photoprint, microfilm, electronic or any other means without permission from the author.

To Harrison

“As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."
— Maurice Wilkes, *"The Design and Use of the EDSAC". Lecture delivered at the Digital Computer Museum, September 23, 1979.*

Abstract

With the advancements of multicore hardware, concurrent and parallel programming has become an essential part of software development. The actor model is an attractive foundation for developing concurrent applications because they can avoid data races by design since actors are isolated concurrent entities that do not share state. However, actor-based programs are not immune to concurrency bugs, e.g., deadlocks can still happen at the message level. Identifying concurrency bugs is a challenging task that consumes time and effort. Unfortunately, the non-deterministic behavior of concurrent programs makes it hard to reproduce bugs. Besides, the mere presence of a debugger can affect the program's behavior, a condition known as the probe-effect.

This dissertation explores the design and implementation of advanced online debugging techniques for actor-based programs. First, we created a taxonomy of concurrency bugs that can occur in actor-based programs, divided into two categories: lack of progress issues and message protocol violations. Using this taxonomy, we classified concurrency bugs found in the literature of actor-based programs. This systematic study drives our exploration of debugging techniques to aid the process of finding the root cause of concurrency bugs.

Second, we proposed catalogs of breakpoint types on messages and stepping operations that combine sequential and message stepping. We implemented such message-based breakpoints and stepping operations with visualizations for message causality and asynchronous stack traces in Apgar, a proof of concept debugger for SOMns programming language in IntelliJ IDE. To evaluate the proposed debugging techniques in Apgar, we conducted a user study following an experimental research design. Although we cannot generalize its results, we obtained positive assessments from participants regarding the proposed debugging techniques, not only to identify the root cause of concurrency bugs but to understand the program's behavior.

Third, we explore a novel technique to build probe-effect free debuggers called multiverse debugging. Multiverse debugging is 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 implemented Voyager, a proof of concept multiverse debugger based on a formal operational semantics of an actor-based language. Finally, 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.

Together, the advanced debugging techniques implemented for SOMns language and the proposed operational semantics of a multiverse debugger for actors presented in this dissertation provide a novel set of interactive debugging tools to help developers identify concurrency bugs in actor-based applications.

Samenvatting

Dankzij vorderingen in multicore hardware is concurrent en parallel programmeren een essentieel onderdeel geworden van softwareontwikkeling. Het actor model is een aantrekkelijke basis voor het ontwikkelen van concurrente toepassingen aangezien data races vermeden worden doordat actoren volledig geïsoleerde entiteiten vormen die geen staat delen. Actor-gebaseerde programma's zijn echter niet volledig immuun voor concurrentie bugs, bv. deadlocks kunnen nog steeds voorkomen op berichtniveau. Het identificeren van bugs in concurrente toepassingen is een uitdagende taak die veel tijd en moeite kost. Helaas maakt het niet-deterministische gedrag van concurrerende programma's het moeilijk om deze bugs na te bootsen. Bovendien kan de aanwezigheid van een debugger het gedrag van het programma beïnvloeden, een probleem dat bekend staat als het probe-effect.

Deze dissertatie onderzoekt het ontwerp en de implementatie van geavanceerde online foutopsporingstechnieken voor actorprogramma's. Allereerst hebben we een taxonomie gemaakt van concurrente bugs die kunnen voorkomen in toepassingen die actoren gebruiken, verdeeld over twee categorieën: gebrek aan voortgang, en schendingen van het berichtprotocol. Met behulp van deze taxonomie hebben we de veel voorkomende en in de literatuur gedocumenteerde types van concurrente bugs geclassificeerd. Deze systematische studie vormt de leidraad voor ons onderzoek naar debugging technieken die kunnen helpen bij het opsporen van concurrente bugs.

Verder hebben we een catalogi samengesteld van breakpoint types voor berichten en stepping-operaties die van toepassing zijn in actor-systemen. We hebben dergelijke breakpoints en stepping-operaties geïmplementeerd in Apgar, een debugger voor de SOMns programmeertaal in IntelliJ IDE, waarmee we de causaliteit van berichten en asynchrone stack-traces kunnen visualiseren. Om de voorgestelde technieken te evalueren hebben we een gebruikersstudie uitgevoerd. Alhoewel we de resultaten niet kunnen veralgemenen, kregen we positieve beoordelingen van de deelnemers over de voorgestelde foutopsporingstechnieken, niet alleen om de onderliggende oorzaak van concurrente bugs te identificeren, maar ook om het gedrag van het programma beter te vatten.

We onderzochten ook een nieuwe techniek om probe-effect vrije debuggers te bouwen, welke we multiverse debugging noemen. Multiverse debugging is een nieuwe benadering voor het debuggen van niet-deterministische programma's dat ontwikkelaars in staat stelt om alle mogelijke executiepaden van een parallel programma te observeren en het programma interactief te debuggen. We hebben Voyager geïmplementeerd, een proof-

of-concept multiversum debugger gebaseerd op een formele operationele semantiek van actor-gebaseerde talen. Tenslotte leveren we een bewijs van non-interferentie, m.a.w. we bewijzen dat het geobserveerde gedrag niet beïnvloed werd door de debugger en vice versa.

Samen bieden de geavanceerde debugging technieken die we hebben ontwikkeld alsook de voorgestelde operationele semantiek van de multiverse debugger voor actoren, gepresenteerd in dit proefschrift, een nieuwe verzameling van interactieve debugging tools om ontwikkelaars te helpen bij het identificeren van concurrente bugs in actor-gebaseerde toepassingen.

Acknowledgements

I wish to express my deepest gratitude to Elisa González Boix, the promotor of this thesis. I cannot thank her enough for the opportunity she gave me to start a Ph.D. at the Software Languages Lab. Her advice, guidance and unconditional support during all these years have been essential for completing this research. Thank you so much for the discussions for every paper, the late work hours, the revision of this dissertation, and mainly for inspiring me to work in the research field of debugging.

I am greatly indebted to Stefan Marr. From the start of this research, his insightful comments have helped me to crystallize ideas and focus on the goals of this dissertation. I truly thank him for his patience with me to get acquainted with SOMns, for reviewing each paper, and for always finding the time to answer my technical questions.

Many thanks to both! Elisa and Stefan, I am deeply grateful for your guidance and persistence for me to become a better researcher.

I also want to sincerely thanks Christophe Scholliers for his guidance and enthusiasm that helped me to materialize the multiverse debugging idea, *"...to boldly debug where no one has debugged before"*. I truly appreciate his time for helping me to get familiar with formalisms and mathematical proofs.

I would like to thank the members of my jury for the time they spent reading this thesis and their suggestions for the final version: Prof. Dr. Beat Singer, Prof. Dr. Viviane Jonckers, Prof. Dr. Kris Steenhaut, Prof. Dr. Philipp Haller, and Dr. Stéphane Ducasse.

Being part of the MetaConc project gave me the opportunity to work with researchers from different universities. I want to give a special thanks to Dominik Aumayr and Hanspeter Mössenböck from the Johannes Kepler University in Austria. To Dominik for his help to understand the technical details of SOMns tracing mechanism. To Hanspeter for his support on the initial ideas of this research. Many thanks also to Robbert Gurdeep Singh from UGent, for his collaboration and work on the Voyager frontend. And I appreciate very much the help of Clément Bera in the initial version of the asynchronous stack trace for SOMns.

Carl Sagan said that *"Science is more than a body of knowledge, is a way of thinking..."*. It has been a privilege for me to have worked at the Software Languages Lab (SOFT) that encourages scientific thinking in researchers. I would like to thank all my colleagues (professors, pre-docs and post-docs) for their feedback in every research presentation I did at the lab. Moreover, I want to express my gratitude to the members of the Distribution and Concurrency research group (DisCo): Matteo, Jim, Kevin, Scull and Isaac for their comments and support during the development of this research. A special thanks to Scull, my colleague for many years, for being a good friend that I can always count on.

I want to thanks also to Louise and Sander, two students that worked with me on the initial ideas about interrogative debugging and the first version of turns visualization, respectively.

I owe a big thanks to the participants of my user study, from the RMOD group at INRIA in France to the current and old SOFTies and guests.

My personal thanks go to my friends and family. I want to express my gratitude to my father, José Luis Torres Carbonell, for sparking my interest in science since I was a child. I want to specially thanks to my mother, Carmen Isabel López Damas, my best friend and dedicated mother who is there for me every time I need her. Without her support all these years, this thesis would not have been finished. Thank you so much, mom! (Quiero agradecer especialmente a mi madre, Carmen Isabel López Damas, mi mejor amiga y madre dedicada que está ahí para mi cada vez que la necesito. Sin su apoyo todos estos años, esta tesis no se habría terminado. ¡Muchas gracias mamá!)

There is a person without whom I would not have known Belgium or SOFT: Humberto Rodríguez Avila. I want to thank him for encouraging me to pursue a Ph.D. and for his support during my research.

Last but not least, I want to thank my son, Harrison Rodríguez Torres. Although he has only three years old, I want to thank him because he has been my strength to finish my research in the last year. For Harrison, *"Remember to look up at the stars and not down at your feet. Never give up work. Work gives you meaning and purpose and life is empty without it. If you are lucky enough to find love, remember it is there and don't throw it away" (Stephen Hawking).*

The work in this dissertation has been founded by the FWO MetaConc research project (G004816N).

Carmen Torres López
June 2021

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Multicore hardware is present everywhere. Having multiple cores in computers allows executing computing tasks in parallel, enabling programs to solve problems faster. Besides performance, it enables to logically separate independent control flows of computation [Sut05].

To take advantage of multicore processors, software developers need to learn how to program concurrent applications. This led Herb Sutter to state that concurrency was *"the next major revolution in how we write software"* in 2005 [Sut05]. However, concurrent programming is difficult, mainly because programmers need to understand and coordinate tasks that may be executed simultaneously. Also, concurrent programming is *non-deterministic*, i.e., the program output does not depend only on the input but on the scheduling of concurrent tasks. Non-determinism also makes debugging and testing difficult since reproducing the same conditions that caused the bug is hard.

Many concurrency models and abstractions have been proposed to facilitate the development of concurrent software [VRH04], e.g., declarative models such as the data-flow model, shared-state models such as the thread-based model, and message-passing models such as the actor model. Since these models are suited to different problems, software developers often use one or multiple concurrency models in the same program [TPLJ13]. Thus, 16 years after Herb Sutter's statement, we observe that software developers have a wide range of models at their hands to program concurrent software. Unfortunately, today there is still few tool support for concurrent software. However, tool support is an integral part of the software development process.

In this dissertation, we focus on debugging techniques for concurrent software written in the actor model. The actor model has been first proposed as a mathematical model in which actors are concurrent entities that communicate via messages [HBS73]. In the 1980s, Gul Agha defined the actor model as a concurrent object-oriented programming model [Agh86]. Since then, the actor model has become very important in the development of concurrent and distributed software. For example, programming languages such as Erlang, Akka and Node.js, have been used to manage WhatsApp communication services[1], manage transactions in Paypal servers[2], and improve database management access for NASA[3], respectively.

The actor model is an attractive foundation for developing concurrent applications because it can avoid some concurrency bugs by design (e.g., data races) since actors are isolated concurrent entities that do not share states. Nevertheless, actor-based programs are not immune to concurrency bugs, e.g., deadlocks and ordering issues can still happen at the message level. These bugs are caused, for example, due to incorrect implementation of the actor protocol or the message patterns, wrong variable initialization, etc.

Developers rely on debugging tools to assist the process of finding the root cause of an application failure. Debugging concurrent programs is, however, difficult due to the increased program complexity and *non-determinism*. Understanding concurrent programs require developers to reason about interactions amongst concurrent entities, i.e., actors. Non-determinism affects the order in which actors receive messages, which can be sensitive to timing and thus hard to reproduce. Furthermore, a debugging tool's mere presence can also affect the order in which concurrent entities are executed, making the reproduction of a bug even rarer. This condition, similar to the Heisenberg uncertainty principle, is known as the *probe-effect* [Gai86].

Despite the developers' efforts, debugging consumes considerable time in the software development process. A study from 2013 estimated that 50% of programming time is spent on debugging, and the global cost of debugging was estimated at 312 billion USD per year (including wages and overheads) [BJC+13]. A more recent study with professional software developers revealed that the majority of most difficult bugs were solved by developers in more than a week or even longer [PSTH16]. The most frequent root causes of their hardest bug were design errors and errors due to the parallel behavior of programs. Also, in that study, participants declared that the hardest bugs were difficult to solve due to the long distance between the unexpected behavior and the root cause, especially for those bugs caused by parallel behavior. The authors of the study concluded that developers require specialized tools to debug parallel programs.

The overall goal of this dissertation is the study of debugging techniques for concurrent programs, particularly programs written using the actor model. Debugging concurrent programs has been studied for many years [MH89, AASE+17]. Existing techniques can be classified into two main families: *offline* debugging, i.e., exploring traces after the program's execution, and *online* debugging, i.e., interactively searching for the fault in a debugging session that controls the program's execution. However, few debugging approaches have focused on actor-based programs. Most existing work for actor-based programs comes from adapting tools from sequential to concurrent programs.

[1]`https://codesync.global/media/scaling-erlang-developer-experience-at-whatsapp/`

[2]`https://www.lightbend.com/case-studies/paypal-blows-past-1-billion-transactions-per-day-using-just-8-vms-and-akka-scala-kafka-and-akka-streams`

[3]`https://openjsf.org/wp-content/uploads/sites/84/2020/02/Case_Study-Node.js-NASA.pdf`

This dissertation explores the design and implementation of advanced techniques for interactively debugging actor-based programs to aid developers in the task of finding the root cause of concurrency bugs. A secondary goal of this dissertation is to study and categorize concurrency bugs in actor-based programs. While we observe that concurrency bugs in thread-based models have been studied for many years [AHB03, PGB+05, LPSZ08, AASE+17], there is not yet a true understanding of which kind of concurrency bugs actor-based programs exhibit. It is only in the last years that the first studies of concurrency bugs for actor-based programs have appeared, e.g., for Scala and JavaScript [BFSK20, WDG+17].

1.1 Research Context

The research presented in this dissertation is mainly related to the fields of *concurrent systems*, and *debugging tool support* to identify concurrency bugs. We describe here our main concerns in each research field.

Concurrent systems research field that studies the different models and techniques to build software that consists of independent components which may perform operations at the same time (i.e., concurrently). Our focus is concurrent systems built using the actor model. In particular, we focus on concurrent systems built on a variant of the actor model called the Communicating Event-Loops (CEL). This concurrency model features non-blocking communication, serial execution, and exclusive access to the actor state. These properties are very attractive for concurrent programming because they avoid certain concurrency bugs like deadlocks and data races by design. Examples of languages that implement CEL concurrency are E [MTS05], AmbientTalk [VCGBS+14], Newspeak [Bra09], SOMns, and JavaScript [TV10].

Debugging tool support research field that studies the design and implementation of debugging techniques and tools to help developers in the task of finding and fixing application failures. We focus on studying the field of building interactive debugging techniques. We aim to enhance the means for developers to explore the program execution interactively to identify the root cause of concurrency bugs.

1.2 Problem Statement

When a program fails, programmers usually reason backward, starting from the failure to discover the cause of the incorrect behavior [Zel09]. Programmers mostly use *observation* to determine facts about parts of the program that have been executed. Common techniques for examining the program's execution include logging (e.g., print statements), offline and online debuggers, and visualization tools.

Classical logging requires adding code into the program, which is later removed by the developer after the observation. Using print statements is not effective in the context of concurrent programs first because introducing code can affect the timing of operations and thus the program's behavior (i.e., probe-effect). Also, logging statements can produce large outputs, which are often mixed with the expected output of the program. This is difficult for the developer to get a proper understanding of the program behavior [Zel09]. Besides, often developers in production insert additional logs in the program, deploy the modified program, and later examine the output. Developers often have to repeat these actions several times when searching for the root cause of a bug, which has been considered impractical for production systems (and even prohibited by change control policies) in which fast solutions are needed [Pac11].

On the one hand, offline debuggers emerged to help to reconstruct the past state after the program has been executed, i.e., from a log file or a trace. Tools for concurrent programs record the program execution as a sequence (or several parallel sequences) of events [MH89]. Interestingly, if the recording of events captures the bug, then it is possible to inspect or deterministically replay the program as many times as needed to understand the conditions leading to the bug. However, offline techniques are said to be expensive in terms of runtime and memory overhead [Eng12, PSTH16].

On the other hand, online debuggers are observation tools that allow developers to do a controlled execution of the program through the use of breakpoints and stepping operations to inspect (or even change) the program state. Thus, developers could obtain a faster result without changing the source code [Zel09]. These tools have been addressed for concurrent programs applying ideas from traditional sequential debuggers. They often behave as a collection of "sequential debuggers", one for each concurrent entity [MH89]. Their main disadvantage is that they suffer from the probe-effect, and thus, are ineffective for timing-dependent failures.

At the start of this dissertation, we observed very few attention from academia and industry in debugging tools for actor-based programs and, more generally, asynchronous code. For instance, Stanley et al. [SCM09] proposed tracking the causality of messages for the E programming language, and Gonzalez Boix et al. [GBNDM14] implemented the first catalog of breakpoints for messages and stepping operations for AmbientTalk. In mainstream languages, only Akka and JavaScript debuggers provide asynchronous stack traces [Dra13, LM18]. In the latest years, we observe more attention from academia in offline debugging techniques, e.g., record and replay approaches [SW17, AMB^{+}18], reverse debugging [MOM18, LNPV18] and interesting visualizations [AZMT18, MOM18]. Some efforts have also been made to incorporate offline features in debuggers for Node.js, e.g., to step backward in the execution of recorded trace and restoring state using checkpoints [BM14, BMM^{+}16, VBMM18]. However, advanced support for online debugging techniques remains unexplored.

We now describe the problems this dissertation tackles:

Limited understanding of concurrency bugs in actor-based programs Developers of concurrent applications do not have available a standard classification of concurrency bugs that can happen in actor-based programs. This knowledge can be useful not only to identify common bug patterns and observable behaviors for concurrency bugs but to guide the design of novel debugging techniques to identify and solve concurrency bugs.

Limited debugging support to identify the root cause of concurrency bugs in actor-based programs Today's debuggers do not provide sufficient support for identifying the root cause of complex concurrency bugs that can occur in actor-based programs, e.g., few debuggers allow setting breakpoints on different points of interest according to the actor model semantics. Moreover, we observe very few integrations between the debugging operations for concurrent code and the ones for sequential one. For example, as far as we know, there is no debugger that combines sequential stepping with message-oriented stepping [4]. Since a concurrency bug can manifest due to a combination of erroneous program states and erroneous interactions amongst actors, finding the root cause of a bug requires controlling both execution of sequential computation as well as interactions amongst actors. Besides, few approaches have studied dedicated visualization techniques that show the actors of the program and the messages they have exchanged. Finally, the combination of online and offline techniques remains unexplored, while that could give developers better tooling support for different types of concurrency bugs.

Absence of probe effect-free debuggers that help to identify concurrency bugs interactively Today's online debuggers suffer from the probe-effect, i.e., the debugger can affect the behavior of the program, which makes it difficult for developers to observe the bug again. While the interactive nature of online debuggers is very helpful to guide the exploration of a concurrent program for identifying the root cause of a bug, they do not help to manage the non-determinism of concurrent programs. The problem is that online debuggers only allow exploring the state of a single execution path (instead of all different states a concurrent program can exhibit at runtime) and may alter the manifestation of bugs.

1.3 Research Goals

In this dissertation, we study advanced debugging techniques to identify concurrency bugs in actor-based programs. This dissertation pursues the following research goals:

We investigate which kinds of concurrency bugs appear in actor-based programs Our goal here is to study bugs that have been reported for actor-based

[4]We denote as message-oriented those debugging operations (e.g., breakpoints or stepping) that work at the level of message interchanged by actors

programs and introduce terminology which can serve as common ground for understanding concurrency bugs in actor-based programs. We aim first to revise the literature of publications to categorize the concurrency bugs that have been reported. Furthermore, we are interested in investigating which features of the state of the art of techniques have been proposed to help to identify concurrency bugs in actor-based programs.

We investigate novel techniques for interactively debugging actor-based programs Based on our study of the state of the art of concurrency bugs and the disadvantages of the techniques developed to tackle them, we aim to design and implement a set of advanced debugging techniques that interactively allow developers to explore the execution of an actor-based program. A secondary goal here is to explore novel visualizations on current and past execution suitable for actor-based programs.

We investigate interactive debugging techniques which do not suffer from probe-effect We aim to design and implement online debugging techniques that help manage the non-determinism of a concurrent program and that do not affect the program behavior when debugging.

1.4 Research Approach

In this dissertation, we design and implement debugging techniques that allow developers to explore the program execution *interactively*. In particular, we explore *online* debugging techniques which are also combined with *offline* features to enable trace-based visualization features in an interactive fashion. The following main approaches will be used to achieve the aforementioned research goals.

Metaprogramming To prototype the novel online debugging techniques, we will depart from self-optimizing AST-interpreters [WWS+12]. This means that the debugger is implemented as part of the interpreter, but it is based on AST node wrapping and relies on the underlying platform for optimizations. The AST node approach will also ease the exploration of integrating sequential and concurrent debugging operations (e.g., stepping).

Formal semantics To build a probe-effect free debugging approach, we depart from an executable operational semantics of an actor-based language. The formalism will allow us to write the semantics of a debugger in a modular and mechanized way, as well as to prove properties on the debugger, e.g., probe-effect free.

1.5 Contributions

In this section, we summarize the main contributions of our research.

A taxonomy of concurrency bugs for actor-based programs Our first contribution is a taxonomy that classifies concurrency bugs reported in the literature of tool development for actor-based applications. To the best of our knowledge, it is the first taxonomy of bugs in the context of actor-based concurrent software when first published in [LMBM18][5]. Our taxonomy consists of six subcategories of bugs grouped in two main categories, i.e., lack of progress issues and message protocol violations. We analyzed the patterns and observable behaviors found in different actor-based programs. We created a catalog of 24 bugs reported in the literature and classified them according to our taxonomy (see Chapter 2).

Interactive debugging techniques for actor-based programs Our second contribution is the design and implementation of advanced debugging techniques for actor-based programs. First, we created catalogs of message-oriented breakpoints and stepping operations, which combine sequential and message stepping to interactively debug actor-based programs. Second, we combine the online techniques with offline debugging techniques, to provide developers with visualizations based on space-time diagrams for tracking message causality. Moreover, to enable better actor state inspection, we enriched the actor state information with message properties such as message type and the turn where the message was sent, also based on trace data. Third, we designed an asynchronous stack trace that shows the calling context and the send context for asynchronous messages, and shows also frames related to promise resolutions. We applied these novel features for debugging actor-based programs written in a Communicating Event-Loops language [MTS05] (see Chapter 5).

A user study to evaluate the advanced debugging techniques Our third contribution is a user study we conducted with 28 software developers to measure our novel debugging techniques for actor-based programs through a mixed methods experimental research design. We measure the time developers spend solving two debugging assignments, and at the end of the experiment, we measure participants' perception about the debugging techniques in a questionnaire (see Chapter 7).

Multiverse debugging, a novel online probe-effect free debugging technique for actor-based programs Our fourth contribution is the design of a novel debugging technique named multiverse debugging. This technique allows developers to interactive explore all possible states to execute a parallel program. We apply multiverse debugging for a language based on the Communicating Event-Loops actor model. We proved observational equivalence between the debugger and the base language semantics in a proof of non-interference, showing that the debugger is probe-effect free (see Chapter 8).

[5]The official publication was in Lecture Notes in Computer Science book series, but the first presentation of our work was in AGERE! 2016 [TLMMGB16]

1.5.1 Technical Contributions

From the mentioned contributions, we derived the following software artifacts:

Apgar is a proof of concept online message-oriented debugger integrated into an IntelliJ plugin for SOMns. SOMns is a programming language that features the Communicating Event-Loops concurrency actor model and it is build on top of the Truffle and Graal platform. Apgar implements the advanced online debugging techniques we have designed for actor-based programs (see Chapter 5 and Chapter 6).

Voyager calculus is a proof of concept multiverse debugger that was implemented in the PLT-Redex programming language for AmbientTalk [VCGBS+14]. AmbientTalk is a programming language which operational semantics features the Communicating Event-Loops concurrency actor model. AmbientTalk operational semantics is used as base language for our multiverse debugger. Voyager allows developers to observe all possible execution paths of an actor-based program and debug it interactively (see Chapter 8).

1.5.2 Supporting Publications

In what follows, we mention the publications that support this dissertation.

- **A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs** Torres Lopez, C., Marr, S., Gonzalez Boix, E., & Mossenbock, H. (2018). *In Programming with Actors - State-of-the-Art and Research Perspectives* (Vol. LNCS 10789, pp. 155-185). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 10789 LNCS). Springer. `https://doi.org/10.1007/978-3-030-00302-9_6` [LMBM18].

  This publication introduced the first taxonomy of concurrency bugs for actor-based programs. We classified concurrency bugs reported in the literature and analyzed their bug patterns and observable behaviors. We also reviewed the state of the art of techniques that have been used to find concurrency bugs in actor-based programs.

- **A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools** Marr, S., Torres Lopez, C., Gonzalez Boix, E., Aumayr, D., & Mossenbock, H. (2017). In D. Ancona (Ed.), *13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2017)* (pp. 3-14). ACM. `https://doi.org/10.1145/3133841.3133842` [MLA+17].

  This publication introduced the first catalogs of breakpoints and stepping operations at the message level we designed and implemented for actor-based programs. Also, an initial version of an actor turn visualization was proposed, which was implemented for the Kómpos debugger.

- **Multiverse Debugging: Non-deterministic Debugging for Non-deterministic Programs.** Torres Lopez, C., Gurdeep Singh, R., Marr, S., Gonzalez Boix, E., & Scholliers, C. (2019). In A. F. Donaldson (Ed.), *ECOOP 2019* (Vol. 134, pp. 27:1–27:30). [10.4230/LIPIcs.ECOOP.2019.27] (Leibniz International Proceedings in Informatics (LIPIcs); Vol. 134). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. `https://doi.org/10.4230/LIPIcs.ECOOP.2019.27` [LSM$^+$19].

  This publication introduces the idea of multiverse debugging as a new online debugging technique for interactively debugging actor-based programs without suffering from probe-effect. We introduce the Voyager calculus: a debugging configuration, and a debugging operational semantics for actor-based programs.

1.6 Dissertation Outline

This dissertation is structured as follows:

Chapter 2: Concurrency Bugs in Actor-based Programs This chapter introduces our first contribution, i.e., a taxonomy we created for concurrency bugs in actor-based from our literature review. We classify the bugs reported in the literature in the categories we proposed in our taxonomy. This chapter also discusses recent studies that derived other taxonomies of concurrency bugs from issues reported in GitHub repositories and Stack Overflow questions. Finally, we describe how the probe-effect makes it hard to reproduce bugs in concurrent programs. We conclude that actor-based programs can exhibit different issues depending on the specific actor model variant. In particular, programs based on the Communicating Event-Loops model can suffer from behavioral deadlocks, livelocks, message order violations, and bad message interleavings.

Chapter 3: State of the Art of Techniques to Handle Concurrency Bugs in Actor-based Programs This chapter describes the current state of the art of techniques that support the development of actor-based programs. In particular, we surveyed techniques from fields of debuggers, visualization, static analysis, and testing. We discussed their advantages and disadvantages with respect to our categorization of concurrency bugs. We conclude that today most of the work of static analysis and testing tools can solve some specific cases of concurrency bugs, and few debuggers support strategies such as recording the causality of messages with message-oriented breakpoints and stepping.

Chapter 4: SOMns: a Concurrent Actor-based Language This chapter describes the SOMns, the Communicating Event-Loops programming language we prototype our advance debugging techniques. It also describes the main features of the technologies in which SOMns is implemented, i.e., the Truffle framework and GraalVM.

We conclude by explaining the necessary implementation details related to asynchronous message passing to understand our work's technical contributions later.

Chapter 5: Online Debugging Techniques for Actor-based Programs This chapter introduces our second main contribution, the design of advanced online debugging techniques for actor-based programs. The first part of the chapter describes the online debugging techniques we propose for actor-based programs, i.e., breakpoints and stepping operations on the level of messages. Besides, we explain how we combine these online techniques with trace-based visualizations to provide actor state and message causality information. We also describe an asynchronous stack trace design to show the calling context and asynchronous context related to the suspension. The second part of the chapter introduces Apgar, our proof of concept debugger for the SOMNS language. We give details about its architecture, and particularly how we extended the Kómpos protocol to enable debugging support for actor-based programs. We conclude the chapter comparing the debugging techniques proposed to related work on debugging tools for actor-based programs.

Chapter 6: Implementation of Online Debugging Techniques for SOMNS This chapter describes how we implemented the advanced online debugging techniques for actor-based programs. In the first part of the chapter, we explain how we implemented the advanced techniques in Medeor, the debugging support in SOMNS interpreter. Medeor implements wrapper nodes in the AST of the target program to enable breakpoints and stepping operations on the level of messages. In the second part of the chapter, we explain the implementation of Apgar frontend as a custom language plugin in the IntelliJ IDE and its main interactions with Medeor.

Chapter 7: Evaluation of Online Debugging Techniques for Actor-based Programs This chapter describes the design and results of a user study we conducted to evaluate the online debugging techniques we have implemented for actor-based programs. Furthermore, we analyze the quantitative and qualitative results and the threats to validity for our mixed-methods experimental research design approach. Overall, we conclude that the advanced debugging techniques may be helpful in the context of actor-based programming to identify concurrency bugs.

Chapter 8: Online Debugging Techniques Probe-Effect Free This chapter introduces the novel technique of multiverse debugging for exploring all non-deterministic paths of an actor-based program when searching for the root cause of a concurrency bug. We start the chapter by defining multiverse debugging, and we show an example of how to apply it for a small language. Next, we introduce Voyager as a proof of concept debugger that uses our implementation of a formal semantics of a multiverse debugger for actors. First, we show a debugging session in Voyager through an actor-based program, and later we explain the calculus implementation. We conclude the chapter with a proof of non-interference for a multiverse debugger that shows that our debugging approach is probe-effect free.

Chapter 9: Conclusion and Future Work This chapter concludes this dissertation and explains avenues for future work.

Chapter 2

Concurrency Bugs in Actor-based Programs

This chapter analyzes concurrency bugs exhibited by actor-based programs. First, we introduce the main concepts related to the Communicating Event-Loops concurrency model, the actor variant on where we focus our research. Second, as first contribution of our research, we propose a taxonomy of concurrency bugs in actor-based programs, which is based on bugs reported in the literature and from our own experience with actor based-programs. Third, we survey and categorize the different concurrency bugs reported in the literature for different actor variants. We then compared our taxonomy of bugs with two recent field studies of concurrency bugs found in real-world repositories of Akka and Node.js programs.

2.1 The Actor Model

The actor model was first proposed as a mathematical model that represents concurrent entities as *actors* which communicate with one another by means of *messages* [HBS73]. Later, Agha defines the actor model as a concurrent object-oriented programming model [Agh86]. Each actor has a thread of execution, which perpetually processes messages stored in its mailbox. Each actor has a *behavior* associated that defines how the actor processes messages. The set of messages that an actor knows how to process denotes the *interface* of the actor's behavior. Besides, actors can store *state* which can only be accessed or mutated by the thread of execution associated to the actor. In other words, actors have exclusive access to their mutable state. In response to a message, an actor can create new actors, send messages to other actors and replace its behavior. In the actor model, sending an asynchronous message means that the sender actor will add a message in the mailbox of a receiver actor and it returns immediately, i.e., the sender actor does not block waiting for the receiver actor to compute a result for the message. Thus, access to the result happens in a *non-blocking* way.

Since the creation of the actor model, several variants have been proposed. De Koster et al. [DKVCDM16] distinguishes three variants in addition to the classic actor model [Agh86]: *active objects* [YBS86], *processes* [AVWW93], and *communicating event-loops* [MTS05].

The actor model variants most used in industry are processes and event-loops. For example, the processes variant is implemented by languages such as Erlang [AVWW93] and Akka in Scala [HS11], while communicating event-loops have been embraced by the asynchronous programming model of JavaScript and Node.js [TV10]. We will now briefly explain both variants.

2.1.1 Processes Model

The processes model represents an actor as a *process*, which as in the classic actor model, contains a mailbox and a thread of execution. The state of a process cannot be accessed by other processes. Processes communicate sending asynchronous messages, which are placed into the mailbox of the receiver actor [AVWW93, HO09].

In general, languages that implement this model has a `receive` statement, which defines the interface of the actor. Typically, this `receive` statement is blocking, i.e., when the actor finishes processing the messages in its mailbox, the process blocks, waiting to receive the next message. Besides, a process can define a flexible interface, i.e., an actor can process different types of messages at different points in time, in this case, for example, evaluating different `receive` expressions [DKVCDM16].

As a concrete example, consider Listing 2.1 showing the receive syntax in Erlang programming language [AVWW93]. The process that implements the `receive` is trying to receive a message described by one of the specified patterns. The process will be suspended until one of the messages received matches a pattern. When the match happens, the actions after the operator `->` are evaluated, and the message is removed from the mailbox of that process.

```
receive
  case msgpattern1 ->
  ... ;

  case msgpattern2 ->
  ... ;
end
```

Listing 2.1: Syntax of the `receive` statement in Erlang [AVWW93].

Actor languages typically offer dedicated syntax for asynchronous messages. For example, asynchronous messages in Akka are sent using ! on the *handler* to the actor instance, i.e., `ActorRef`. The `ActorRef` is used mainly to communicate with an actor and control the actor's life cycle [HS11]. Akka also provides a mechanism to identify unavailable actors through *dead letters*. Messages that cannot be delivered (e.g., because the receiver actor was terminated before receiving the message) will be sent to a "synthetic" actor for later inspection.

Typically, processes implementation such as Erlang do not provide futures. Akka does support futures to retrieve the result of concurrent operations synchronously or asynchronously, but they are not integrated with the actor model (as we discuss later in Section 2.3.4).

2.1.2 Communicating Event-Loops Model

The Communicating Event-Loops (CEL) is a non-blocking variant of the actor model of concurrency first introduced by the E language [MTS05]. The model was also adopted by languages such as AmbientTalk/2 [VCGBS$^+$14], and SOMNS, the language we use to prototype our debugging techniques in this dissertation [Bra09].

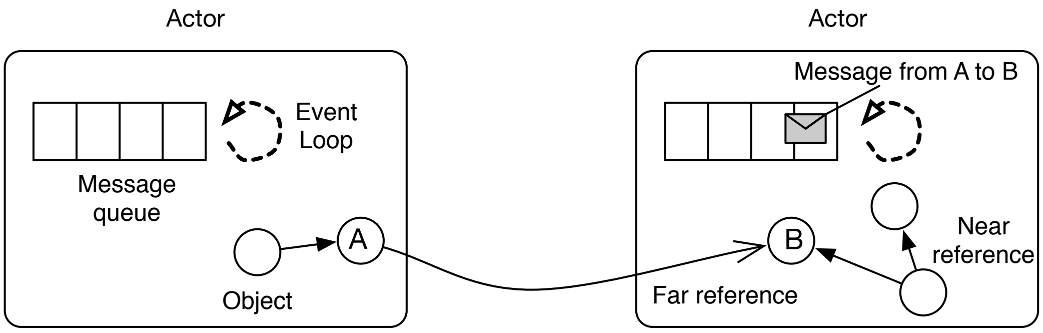


Figure 2.1: Overview of the CEL model (from [VCGBS$^+$14]).

Figure 2.1 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 messages sequentially 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*. Like in the process variant, actors have exclusive access to their mutable state. This means that each object is owned by one actor, and only the owner actor can directly access it.

In contrast to the process variant, in CEL, objects stored in an actor can be accessed by other actors. 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. A *near reference* is a direct reference between two objects within the same actor. Communication between these objects happens through synchronous messages.

To reconcile return values with asynchronous message passing, CEL model introduced the concept of non-blocking promises [MTS05]. A promise is a placeholder for the result that is to be computed by the receiver actor in response to the asynchronous message. Some actor languages refer to promises as futures [VCGBS$^+$14]. In CEL, the promise itself is an object, which can also receive asynchronous messages. Thus, a promise resolution can be dependent on another promise (returned from another asynchronous message), which is known as *promise chaining*. Messages sent to a promise are not delivered until the promise is resolved. Once a promise is resolved, the asynchronous message is forwarded in order to the result value of the computation [MTS05].

2.2 Terminology about Concurrency Bugs

Before studying different kinds of concurrency bugs for actor-based programs, we will first introduce the terminology that we will use in this dissertation. A *concurrency bug* is a failure related to the interactions among different concurrent entities of a system. Following Avizienis's terminology [ALRL04], a *failure* is an event that occurs when the services provided by a system deviate from the ones it was designed for. The discrepancy between the observed behavior and the theoretically correct behavior of a system is called an *error*. Hence, an error is an event that may lead to a failure. Finally, a *fault* is an incorrect step in a program that causes an error (e.g., the cause of a message transmission error in a distributed system may be a broken network cable). A fault is said to be *active* when it causes an error, and *dormant* when it is present in a system but has not yet manifested itself as an error. Throughout this dissertation, we use the terms concurrency bug and issue interchangeably.

This dissertation studies concurrency bugs that appear in actor-based programs used in concurrent systems. Since actors are a good fit for distributed computing, they have been adopted in languages targeted for distributed systems like Erlang, Akka and AmbientTalk. However, bugs that are only observable in distributed systems (e.g., due to network failures) are out of the scope of this dissertation.

2.3 Taxonomy of Concurrency Bugs for Actor-based Programs

While there are many studies for concurrency bugs in thread-based programs [AHB03, PGB$^+$05, LPSZ08, BFSS10, LLLG16, AASE$^+$17], only a few studies focus on message passing programs. Zhang et al. [ZWCZ15] study bug patterns, manifestation conditions, and bug fixes in three open source applications that use message passing. Also, Tu et al. [TLSZ19] study root causes and possible fixes for concurrency bugs when implementing communication channels in the Go programming language. Often, literature documents particular issues, e.g., ordering problems [LBL$^+$16], but very few efforts has focused on taxonomizing bugs. For actor-based programs, there was not established terminology at the beginning of this thesis. The first contribution of our research has been to introduce a taxonomy of concurrency bugs for the actor model derived from bugs reported in literature of actor-based programs.

This section introduces a taxonomy of concurrency bugs for the actor model derived from concrete bugs reported in the literature and from our own experience with actor-based programs. Table 2.1 introduces our proposed terminology for concurrent bugs in actor-based programs. Our overall categorization starts from the terminology for thread-based concurrency in literature (also shown in Table 2.1), which classifies bugs in two general categories: lack of progress issues and race conditions.

Depending on the guarantees provided by a specific actor model, programs may be subject to different concurrency bugs. Therefore, not all concurrency bugs are applicable to all actor variants. In the rest of the section, we define each type of bug and discuss in which variants can be present.

2.3.1 Lack of Progress Issues

Lack of progress issues in thread-based programs denotes the condition when one or more threads are in a *waiting state* because they need to obtain a resource that is held by another thread which is also waiting [AASE+17]. Actor-based programs can also exhibit this lack of progress issues, but they manifest differently, i.e., at message level instead of memory level. We have identified three kinds of conditions that can lead to a lack of progress in an actor-based program: communication deadlocks, behavioral deadlocks, and livelocks.

2.3.1.1 Communication Deadlock

A communication deadlock is a condition in a system where two or more actors are blocked forever, waiting for each other to send a message. This condition is similar to traditional deadlocks known from thread-based programs. We base the terminology on the work of [CS11b] in Erlang concurrency bugs.

Communication deadlocks can *only* occur in variants of the actor model that feature a blocking `receive` operation. This is common in variants of the actor model based on processes such as Erlang and the Scala Actors framework [HO09]. A communication deadlock manifests itself when an actor only has messages in its inbox that cannot be received with the currently active `receive` statement. Listing 2.2 shows a communication deadlock example in Erlang [CS11b]. In Line 12 the `pong` process is blocked because it is waiting for a message that is never sent by the `ping` process, i.e., its mailbox is empty. The *fault* is in Line 7, there the `ping` process returns `ok` instead of sending the message `ping_msg` to the pong process.

2.3.1.2 Behavioral Deadlock

Many variants of actors do not feature blocking constructs but they can still suffer from deadlocks at message level. We call a behavioral deadlock to the condition when two or more actors *conceptually* wait for each other because the message to complete the

Concurrency Model	Category of Concurrency Bugs		Bug Definition
Threads	Lack of Progress	Deadlock	condition in a system where two or more threads are blocked forever waiting for another thread to do something [PGR+15].
		Livelock	condition in which two or more threads while not blocked cannot make further progress [PGB+05].
	Race Condition	Data race	special case of race condition that occurs when two threads access the same data and at least one of them writes the data [AASE+17].
		Bad interleaving (also known as high-level data race [AHB03], atomicity violation [AASE+17])	occurs when the program exposes an inconsistent intermediate state due to the overlapping execution of two threads [PGR+15].
		Order violation	occurs when the expected order of execution of at least two memory accesses is not respected [AASE+17].
Actors	Lack of Progress	Communication deadlock	condition in a system where two or more actors are blocked forever waiting for each other to send a message.
		Behavioral deadlock	condition in a system when two or more actors are not blocked but wait on each other for a message to be able to progress, i.e., the message to complete the next step is never sent.
		Livelock	condition similar to a deadlock in which two or more actors are not able to make progress but they continuously change their state.
	Message Protocol Violation	Message order violation	condition in which the order of exchanging messages of two or more actors is not consistent with the intended behavior of the program.
		Bad message interleaving	occurs when a message is processed between two messages which are intended to be processed one after the other.
		Memory inconsistency	occurs when different actors have inconsistent views of shared resources. The effects of the turn that modifies a conceptually shared resource, may not be visible to other actors which also alter the same resource.

Table 2.1: Taxonomy of concurrency bugs.

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

Listing 2.2: Communication deadlock example in Erlang (from [CS11b]). Line 12 has a blocking `receive` causing the `pong` process to deadlock because the expected message is never sent.

next step in an algorithm is never sent. In this case, no actor is necessarily suspended. We call this situation a behavioral deadlock, because the mutual waiting prevents local progress. However, these actors might still process messages from their mailbox (sent by other actors). Since actors do not actually block, detecting behavioral deadlocks can be harder than detecting deadlocks in thread-based programs.

We illustrate a behavioral deadlock in an implementation of the widely known dining philosophers concurrency problem [Dij65] written in SOMNS [Bra09]. In the context of dining philosophers, a behavioral deadlock can prevent philosophers from eating when they cannot acquire two forks. In our implementation shown in Listing 2.3, the left fork has the same value as the `id` of the philosopher, but for the right fork the program computes its value (Line 12). For example, philosopher 1 will eat with fork 1 and 2 and so on. The *fault* occurs when the philosopher puts down its forks: the right fork gets a wrong value (Line 22) because the implementation swapped `numForks` and `leftForkId` variables. This causes fork 2 and 4 to be always taken. Consequently, philosopher 2 and 4 never eat, and philosopher 1 and 3 eat only once. However, since philosopher 5 can always eat, the overall application shows local progress.

In contrast to communication deadlocks, all variants of actor models can suffer from behavioral deadlocks. One cause for such deadlocks are *flexible interfaces* [DKVCDM16], because when an actor limits the set of messages it accepts, the overall system can reach a state where actors mutually wait for messages being sent, without allowing any progress. On the other hand, if an actor implements two or more interfaces, it could be that only one of them is deadlocked, allowing some progress with respect to interactions with other actors.

```
 1 class PhilosopherActor new: id rounds: rounds
 2     counter: aCounter arbitrator: arbitrator = (
 3   (* ... *)
 4   public start = (
 5     arbitrator <-: pickUpForks: self id: id.
 6   )
 7 )
 8 class ArbitratorActor new: numForks resolver: resolver = (
 9   (* ... *)
10   public pickUpForks: philosopher id: leftForkId = (
11     | rightForkId |
12     rightForkId := 1 + (leftForkId % numForks).
13     ((forks at: leftForkId) or: [forks at: rightForkId])
14       ifTrue:  [ philosopher <-: denied ]
15       ifFalse: [
16         forks at: leftForkId  put: true.
17         forks at: rightForkId put: true.
18         philosopher <-: eat ]
19   )
20   public putDownForks: leftForkId = (
21     | rightForkId |
22     rightForkId := 1 + (numForks % leftForkId).
23     forks at: leftForkId  put: false.
24     forks at: rightForkId put: false.
25   )
26 )
```

Listing 2.3: Behavioral deadlock example of a dining philosopher implementation in SOMNs. Line 22 calculates `rightForkId` incorrectly, preventing the philosophers from eating.

2.3.1.3 Livelock

A livelock occurs when two or more actors are continually changing their state but they actually do not make any progress, i.e., actors repeat the same interaction in response to the message sent by each other.

An example for a livelock is given in Listing 2.4. It shows the sleeping barber problem [Dij68] implemented in SOMNS. The waiting room, the barber, and the customers are implemented as actors. The concurrency issue in this example is caused by a *fault* in Line 7. Instead of receiving the next customer from the collection of customers `waitingCustomers`, the barber always receives the same first customer. Both actors, room and barber are not blocked. The barber asks for the next customer to the room (Line 20) and the room sends the customer to the barber to do the haircut (Line 8). But, as the customer that is sent is always the same, there is no global progress.

```
 1 class WaitingRoomActor new: capacity barber: anActor = (
 2   (* ... *)
 3   public next = (
 4     waitingCustomers size > 0
 5      ifTrue: [
 6        | customer |
 7        customer := waitingCustomers first.
 8        barber <-: enter: customer in: self ]
 9      ifFalse: [
10        barber <-: wait.
11        barberAsleep := true ]
12   )
13 )
14 class BarberActor new: resolver = (
15   (* ... *)
16   public enter: customer in: room = (
17     customer <-: start.
18     busyWait: (random next: avHaircutRate) + 10.
19     customer <-: done.
20     room <-: next
21   )
22 )
```

Listing 2.4: Livelock in a sleeping barber implementation in SOMNS. Line 7 reads always the same customer, but does not remove it from the list, preventing global progress.

2.3.2 Message Protocol Violations

The second big family of concurrency issues are related to how messages are processed by actors. Actors cannot suffer from data races like thread-based programs since they have exclusive access to their state and messages are processed serially. Nevertheless, computation in actor-based programs depends on the order how actors process messages.

We call message protocol violations to the race conditions that happen at the level of messages. We consider these race conditions to be at a *high-level* to distinguish them from the low-level memory access race conditions that occur in thread-based programs, because they occur at the level of messages rather than memory locations. Therefore, we refer to them more specifically as *message protocol violations*. We identified three types of message protocol violations, which are described in the remainder of this subsection: *message order violations*, *bad message interleavings*, and *memory inconsistencies*.

2.3.2.1 Message Order Violation

A message order violation appears when the order in which two or more actors exchange messages is not consistent with the intended *behavior* of the program. This includes messages that are received out of order or unexpected interleavings of messages. They are typically caused by actors only supporting a subset of all possible message sequences.

Message order violations are common for instance in JavaScript event-loops. In a contemporary browser, each script runs inside one single-threaded event-loop per page. After the initial parsing and interpretation of `<script>` tags, the event-loop processes incoming events related to page lifecycle events, UI events, timer events, XRS responses, etc. The order in which corresponding event handlers are executed is non-deterministic, e.g., because of user actions or I/O timing. This can give rise to an unexpected ordering of messages that is not handled correctly by the program. Listing 2.5 extracted from [RVS13] shows an example of such a message order violation. In this case the failure occurs because of an interleaving between the execution of the user action `onclick` and the HTML parsing.

The code in Listing 2.5 defines an input tag for a button in an HTML page (Line 2), and two scripts: one declaring two variables (`init` and `y`) and the behavior of function `f` which is executed when the button is clicked (Line 4–12), and a second script which updates the variables `init` and `y`. Since the parsing of the `input` tag and the execution of the scripts happen in different turns of the event-loop, a violation in the order of messages execution can occur. For example, if the button is clicked before the first script runs, the function `f` is not yet declared, causing the JavaScript interpreter to crash. As a result, the user will not see a result from the button clicked.

In the example in Listing 2.5 the message order violation only affects a *single* actor, because client-side web programs runs in a *single* JavaScript event-loop, which processes all types of events. In general, message order violations can also involve more than two actors. In the context of JavaScript, a message order violation can affect more than one event-loop if a client-side web application employs WebWorkers, or interacts with a Node.js server.

```
 1 <html><body>
 2   <input type="button" id="b1" onclick="javascript:f()">
 3    ... <!-- many elements -->
 4     <script>
 5     function f() {
 6       if (init)
 7         alert(y.g);
 8       else
 9         alert("not ready");
10     }
11     var init = false, y = null;
12     </script>
13     ...
14     <script>
15       y = { g: 42 };
16       init = true;
17     </script>
18 </body></html>
```

Listing 2.5: Message order violation within a single event-loop in JavaScript (from [RVS13]). On line 2, the `onclick` event can be triggered by the user before the function `f` is parsed and made available, causing an error.

2.3.2.2 Bad Message Interleaving

We define a *bad message interleaving* as the condition when a message is processed in between two messages which are expected to be processed one after the other, causing some misbehavior of the application or even a crash.

In the original actor model, messages are expected to be eventually delivered in the order in which the sender actor sent them (i.e., on a per-sender basis). However, messages from different senders may be interleaved in between messages from one sender. In other words, even if the actor model enforces that messages are processed in a first-in-first-out (FIFO) order per-sender, messages from different sender actors may still be processed between them. Moreover, some implementations of the actor model do not guarantee in-order delivery of the messages. This can be found in actor models used to build distributed systems, like Scala Actors framework[1] [HO09] or ActorFoundry library for Java [LDMA09] in which communication between actors is not enforced to work in a FIFO manner.

Listing 2.6 shows an example of bad message interleavings translated from a real example in Scala to the ActorFoundry programming language (extracted from [LDMA09]). The listing shows an example of bad message interleaving in a network communication between two actors, `Server` and `Client`. In line 10, the `Client` sends an asynchronous

[1]Scala Actors is currently deprecated. The Akka framework (2.6.14), is now the default actor library for Scala, and it does guarantees FIFO message delivery for some mailbox implementations, `https://doc.akka.io/docs/akka/current/typed/mailboxes.html`.

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

Listing 2.6: Bad message interleaving example in ActorFoundry (from [LDMA09]). The `Server` actor can interleave the messages `set` and `get` sent by the `Client` actor. If that is the case `v1` will get a value that differs from `v2`.

message to the `Server` to store the value 1. In line 11, the `Client` does a `call`, which waits for a result, to retrieve the value from the `Server`. Since the `Server` processes the `set` message between the two `get` messages, the values of `v1` and `v2` will be inconsistent.

In the context of JavaScript, bad message interleavings can also occur within a single event-loop if programs can receive notifications for external events, e.g., events from the network, from timers or from sensors. Such issues have been previously reported by [HPK14].

2.3.2.3 Memory Inconsistency

A memory inconsistency is a condition in which different actors have inconsistent views of shared resources. This can be caused because the effects of the turn that modifies a *conceptually shared resource* may not be visible to other actors which also alter the same resource. Previous research on Erlang has reported such kinds of problems [Huc99, HB11, DKO13].

Listing 2.7 shows a modified fragment of an Erlang program used by [DKO13] to verify the property of mutual exclusion in actors. The program (originally introduced by [Huc99]) spawns one database process and several client processes. The purpose of the program is to save information in a database, which acts as a conceptually shared resource by different client actors. The database consists of a map of key-value tuples. When a client process sends an `allocate` message to the database, the database checks if the key exists already (Line 8). If the value does not exist (Line 29) then it is saved. The `free` message in the client computes the value to be saved (Line 10) and then the client process sends the tuple to the database. If a second process does lookup before the first value is saved, the `lookup` function will fail due to the key not having been inserted yet. When the database process receives the key and value to be stored, another client that has a different value with the same key can save it. Thus, the value sent by the first process will be overwritten by the value of another client process. The *fault* is in Line 11, and to avoid the memory inconsistency it needs to be replaced by the commented `receive` statement (from Line 12 to 15). This way the value sent by the client is saved and we avoid other processes making a lookup.

```
 1 main() ->
 2     DB = spawn(fun()->dataBase(#{})end),
 3     spawnmany(fun()->client(DB) end).
 4
 5 dataBase(M) ->
 6    receive
 7        {allocate,Key,P} ->
 8            case lookup(Key,M) of
 9                fail ->
10                    P!free,
11                    dataBase(M);
12                    % receive
13                    %      {value ,Key ,V} ->
14                    %            dataBase(maps:put(Key,V, M))
15                    % end;
16                succ ->
17                    P!allocated,
18                    dataBase(M)
19            end;
20        {lookup,Key,P} ->
21            P!lookup(Key,M),
22            dataBase(M);
23        {value,Key,V} ->
24           dataBase(maps:put(Key,V, M))
25    end.
26
27 lookup(K,M) ->
28    case maps:find(K,M) of
29        error -> fail;
30        _V    -> succ
31    end.
32
```

Listing 2.7: Memory inconsistency example in Erlang (based on [Huc99, DKO13]). Line 23 shows a message pattern that allows different processes to store different values for the same key.

2.3.3 Comparison with Existing Terminology in Literature

The goal of establishing a taxonomy is to provide a common vocabulary for concurrency bugs in actor-based programs. As we mentioned before, there is no agreed terminology for bugs found in actor-based programs. In what follows, we relate terms found in literature to our taxonomy.

Bad message interleavings have been denoted as *ordering problems* by Lauterburg et al. [LDMA09] and Long et al. [LBL$^+$16] but we consider ordering problems to be too coarse-grained terminology. The term *atomicity violation* was denoted by Zheng et al. [ZBZ11], Hong et al. [HPK14] and Wang et al. [WDG$^+$17], but we decided to use the term bad message interleaving to avoid confusion with atomicity violations in thread-based programs due to low-level memory accesses errors.

Message order violations have been collected under many different names in literature: *data races* by Petrov et al. [PVSD12], *harmful races* by Raychev et al. [RVS13], *order violations* by Hong et al. [HPK14] and Wang et al. [WDG$^+$17], and *message ordering bugs* by Tasharofi et al. [TPLJ13]. We consider message order violations to be a descriptive name while avoiding confusion with low-level data races present in thread-based programs.

Memory inconsistency problems have been denoted as *race conditions* by Hughes et al. [HB11] and D'Osualdo et al. [DKO13].

Finally, the term *orphan messages* [CPS97] has been employed to refer to messages that an actor sends but that the receiver actor(s) will never handle. Rather than a kind of concurrency bug, we consider orphan messages as an observable property of an actor system which may be a symptom of a concurrency bug like communication deadlocks or message ordering violations. We use this terminology in the next section when we classify concurrency bugs reported in literature with our taxonomy. Orphan messages can for example be present in actor languages that allow flexible interfaces such as Erlang, the Scala Actors framework and the Akka library [DKVCDM16]. An actor may change the set of messages it accepts after another actor has already sent a message which can only be received by an interface which is no longer supported.

2.3.4 Issues Mixing Actor Libraries with other Concurrency Models

The taxonomy proposed in this chapter focuses only on concurrent programs written with the actor model. However, often mainstream languages offer the actor model as a library next to other concurrency models, e.g., with threads. The most representative example is the Scala language, in which the thread and actor model coexist.

Tasharofi et al. [TPLJ13] did a study of actor programs written in Scala and found that 80% of the programs mixed the actor model with another concurrency model. Their study revealed that some of the reasons for the mixing of concurrency models are that developers lack knowledge about the actor library or found it easier to implement certain protocols using a shared-memory model. Others argue that the support provided by the library, e.g., for I/O, was not efficient [TPLJ13].

In this context, when combining, for example, actors and threads, thread-based bugs such as deadlocks and data races that do not happen in actor-based programs could occur. For example, race conditions can be seen when combining, for example, actors and futures in Akka [Hal15, BFSK20]. As a concrete example of a data race between thread an actors, consider Listing 2.8. The example implemented in Akka Java shows a simple computation of adding two numbers using a `Math` actor (Line 25). The callback (Line 32) uses the `log` variable, but the actor writes on the `log` when processing the received result (Line 21). This data race can happen because the application of the `thenApply` closure in the future is not modeled as a message in the actor mailbox (as for example, how event-loops languages do). Instead, futures and actors run in different threads, and simultaneous memory access on the actor state can happen if developers are not careful. Tools to cope with these challenges when mixing concurrency models are also needed, but it is out of the scope of this dissertation.

2.4 Study of Concurrency Bugs in Actor-based Programs

In this section, we review various concurrency bugs reported in literature, and classify them according to the taxonomy introduced in Section 2.3.

The goal is twofold: (1) to classify concurrency bugs collected in prior research in the bug categories according to our taxonomy and (2) to identify bug patterns and observable behaviors that appear in programs exhibiting a particular concurrency bug. The latter is useful to design mechanisms for testing, verification, static analysis, or debugging of such concurrency issues. In the context of this dissertation, this study was used to ground our research on novel techniques for debugging actor-based programs.

Our survey gather 24 distinct concurrency bugs reported in the literature. Table 2.2 summarizes the *bug patterns* and *observable behaviors* by our bug categories (see Table 2.1). Each bug scenario is identified with an id, i.e., bug-id. We describe the bug pattern as a generalized description of the fault by identifying the actions that trigger the error. And we describe the observable behavior of the program that has the concurrency issue, i.e., the failure. The full catalog for each reported concurrency bug is shown in Table A.1. In the remainder, we highlight the identified bug patterns for each bug scenario in italic font.

```java
public class MathUserActor extends AbstractActor {
  private static ActorRef mathActor;
  private final  LoggingAdapter log =
    Logging.getLogger(getContext().getSystem(), this);

  public MathUserActor(ActorRef mathActor){
    this.mathActor = mathActor;
  }

  @Override
  public Receive createReceive() {
    return receiveBuilder()
           .match(CalculateSomething.class, this::receiveCalculateSomething)
           .match(Result.class, this::receiveResult)
           .match(CalculateSomethingFuturized.class,
                   this::receiveCalculateSomethingFuturized)
           .build();
  }

  private void receiveResult(Result message){
    log.info("sum = {}", message.result);
  }

  private void receiveCalculateSomething(CalculateSomething message){
    mathActor.tell(new Sum(1,2), getSelf());
  }

  private void receiveCalculateSomethingFuturized (CalculateSomethingFuturized
    message){
    Duration t = Duration.ofSeconds(1);
    CompletableFuture<Object> future =
      ask(mathActor, new Sum(3,2), t).toCompletableFuture();
      future.thenApply( v -> {
        Result vv = (Result) v;
        // watch out! use getContext().getSystem().log() instead
        log.info("sum = {}", vv.result);
        return vv; });
  }
}
```

Listing 2.8: Code snippet of a program written in Akka in Java that could manifest a data race (from Parallelism and Distribution course at VUB).

Bug category	Bug-id	Bug pattern	Observable behavior
message order violation	1, 11, 13, 14, 16, 17, 18, 19, 20, 23	incorrect execution order of actors (e.g., processes) or messages (e.g., user events)	crashes, exceptions, exit abnormally
bad message interleaving	12, 15, 21, 22, 24	interleaving of one message between two other messages	value inconsistencies, exceptions, errors
memory inconsistency	2, 3, 5, 6, 7, 8	actors accessing to shared resources (e.g., access to Erlang Term Storage or Mnesia database)	value inconsistencies, exceptions
communication deadlock	4, 9, 10	incorrect message implementation (e.g., orphan messages)	blocked process

Table 2.2: Summary of bugs patterns and observable behaviors found in literature.

2.4.1 Lack of Progress Issues

Most of the reported communication deadlocks so far are in the context of Erlang programs. Table A.1 shows 3 communication deadlocks we found in this context. Bug-4 is an example of a communication deadlock collected by Christakis and Sagonas [CS11b], which corresponds to the example depicted in Listing 2.2.

Christakis and Sagonas [CS11b] distinguish two causes for communication deadlocks in Erlang programs:

- *receive-statement with no messages* i.e., empty mailbox,
- *receive with the wrong kind* i.e., the messages of the mailbox are different to the ones expected by the receive statement.

We classify these conditions as bug patterns for orphan messages, which can lead to communication deadlocks in Erlang.

Christakis and Sagonas [CS11a] mention also other conditions that can cause mailbox overflows or potentially indicate logical errors. Such conditions include *no matching receive*, i.e., the process does not have any `receive` clause matching a message in its mailbox, or *receive-statement with unnecessary patterns*, i.e., the `receive` statement contains patterns that are never used.

Bug-9 was identified by Gotovos et al. [GCS11] when implementing a test program in Erlang which has a server process that receives and replies to messages inside a loop. The server process blocks indefinitely because it waits for a message that is never sent. They also identify it as problematic, *when a message is sent to an already finished process*, which is exhibited by bug-10. This can happen due to two possible situations. First, if a client process sends a message to an already finished server process, the client process will throw an exception. Second, if the server process exits without replying after the message was received, the client process will block waiting for a reply that is never sent. We categorize bug-4, bug-9, and bug-10 as communication deadlocks and the observable behaviors as orphan messages.

D'Osualdo et al. [DKO13] identified three other bug patterns leading to abnormal process termination in Erlang programs, which might cause deadlocks: *sending a message to a non-pid value*, *applying a function with the wrong arity* and *spawning a non-functional value.* These bug patterns could result in a communication deadlock or in a message order violation if the termination notification is not handled correctly.

Aronis et al. [AS17] studied built-ins operations that can cause races in Erlang programs. Because the studied built-ins can access memory that is shared by processes, races can be observed in form of different outputs. Their classification on observable interferences of Erlang/OTP built-ins can help to diagnose communication deadlocks, message order violations, and memory inconsistencies.

2.4.2 Message Protocol Violations

We found 10 message order violations, 5 bad message interleavings and 6 memory inconsistencies reported in the literature of actor-based programs. In what follows, we discuss the bug patterns and observable behaviors of those message protocol violations.

2.4.2.1 Message Order Violation

In Erlang, updating certain resources such as the global name registry requires careful coordination to avoid concurrency issues. Bug-1 is an example of a race on the global process registry [CS10]. The bug is caused because two processes try to register processes for the same global name more than once, which is done with non-atomic operations. For correctness, these processes would need to coordinate with each other.

Bug-11 reported by Christakis et al. [CGS13b] is another example of a message order violation exhibited when a *spawned process terminates before the parent process registers its process id.* The application expects the parent process to register the id of the spawned process before the spawned process is finalized, but as the execution of `spawn` and `register` functions are not atomic, an unexpected termination can cause a message order violation.

Zheng et al. [ZBZ11] identified issues when two events are executed but the application cannot return the responses in time, e.g., *the second message is executed with the value of the first message.* It is included in our table as bug-14. They argue that the cause of this issue can be the network latency and the delay in managing the responses by the JavaScript engine. If the events operate on the same data, it can lead to inconsistencies e.g., deleting an object of a previous event. We consider this a case of a message order violation, because the order of the messages is not consistent with the expected protocol or behavior of the web application.

In the context of JavaScript, Petrov et al. [PVSD12] identified 4 different message order violations. An *interleaving between the execution of a script and the event for rendering an input text box* is shown in bug-17, which can lead to inconsistencies when saving the text a user entered. Also problematic is the potential *interleaving of creating an HTML element and executing a script that uses the element* shown in bug-18. If the HTML element has not yet been created, it will cause an exception. Moreover, bug-19 corresponds to an issue that manifests when *executing a function can race with its definition.* This can happen when the function is invoked first because the HTML loads faster, and the script where it is declared is only loaded later. Another example is shown in bug-20, *the `onload` event of an HTML element is triggered before the code is loaded*, which causes the event handler to never run correctly.

Raychev et al. [RVS13] detected similar race conditions to the ones reported in [PVSD12], which we categorize as message order violations. Listing 2.5 corresponds to their bug example which we identify as bug-16. Hong et al. [HPK14] also collected message ordering violations in three different existing websites. One of its examples shows a scenario where *a user input invokes a function before it is defined.* This last example is detailed in bug-23. From all these collected bugs, we conclude that a common issue in JavaScript programs is the *bad interleaving of two events in an unexpected order.*

Tasharofi et al. [TPLJ13] identified twelve bugs in five Scala projects using the Akka actor library, which we categorize as message ordering problems. Bug-13 gives details of one of these bugs. The study found two bug patterns in Scala and Akka programs that can cause concurrency bugs in actors. First, *when changing the order of two receives in a single actor (consecutive or not)*, which can provoke a message order violation. Second, *when an actor sends a message to another actor which does not have the suitable receive for that message.* This last issue manifests as an orphan message, and can also lead to other misbehaviors such as communication deadlocks.

2.4.2.2 Bad Message Interleaving

Most of the bad message interleavings we have found are reported in JavaScript. Zheng et al. [ZBZ11] also identified bad message interleavings such as the one exhibited in bug-15. The bug pattern corresponds to the *use of a variable not initialized by other methods before it was defined.* This delay of receiving a response can be caused by a busy network and leads to an exception in the application. Hong et al. [HPK14] also observed bad message interleavings in JavaScript programs. Bug-21 shows a pattern in which a variable is undefined because *after a user has uploaded a file to a workspace, the user changes the workspace before the file has been completely uploaded.* In the case of bug-22, a variable is `null` because an *event handler updates the DOM between two inputs events that manipulate the same DOM element.*

Chang et al. [CDG$^+$19] reported a bad message interleaving from a Node.js application. The code snippet corresponds to a callback chain of three callbacks, which should be executed one after the other. However, because Node.js does not guarantee the atomicity of the callback chain, message interleavings can occur. Two cases can be observed due to the bad interleaving, a message is set to undefined, and a message is overwritten.

Bug-12 corresponds to the example of bad message interleaving collected by Lauterburg et al. [LDMA09] from Scala and translated to ActorFoundry, which was shown in Listing 2.6. The bug pattern occurs when *an actor executes a third message between two consecutive messages due to the actor model implementation being not FIFO.*

2.4.2.3 Memory Inconsistency

To the best of our knowledge, memory inconsistency issues have only been reported in the context of Erlang programs. Christakis et al. [CS10] shows an example of high-level races between processes using the Erlang Term Storage in bug-2. In this case the error is due to *inserting and lookup in tables that have public access*, thus it is possible that two or more processes try to read and write from them simultaneously. A second example detailed in bug-3, shows a similar issue that can happen when accessing tables of the Mnesia database. The cause is due to the *use of reading and writing operations that can cause race conditions.* We categorize both issues as memory inconsistency problems.

Hughes et al. [HB11] detected four bugs corresponding to memory inconsistencies in *dets*, the disk storage back end used in the Erlang database Mnesia. Bug-5 refers to *insert operations that run in parallel* instead of being queued in a single queue. They can cause inconsistent return values or even exceptions. The observable behavior of bug-6 corresponds to an inconsistency of visualizing the dets content. This issue can occur when *reopening a file that is already open and executing insert and get_ contents operations in parallel.* Bug-7 and bug-8 are caused due to failure on integrity checks. Bug-7 is reproduced only in one specific scenario when *running three processes in parallel*, and bug-8 can occur only in those languages implementations that similar to Erlang with Mnesia *can keep new and old versions of the server state.*

Huch et al. [Huc99] and D'Osualdo et al. [DKO13] conducted studies to verify mutual exclusion in Erlang programs. Listing 2.7 shows an example. The bug pattern identified corresponds to the *wrong definition of the behavior of the actor*, and the observable behavior is that two actors can store different values for the same key which leads to inconsistencies.

2.4.3 Concurrency Bugs by Actor Variants

We now discuss which concurrency bugs can occur according to the two variants of the actor model for which we have found bugs in literature, i.e., processes and CEL. Furthermore, we identify the patterns that can cause a concurrency bug and the behavior that can be observed in the programs that have these bugs.

2.4.3.1 Lack of Progress Issues

In languages that implement the *processes* variant of the actor model, e.g., Erlang and Scala, programs can exhibit communication deadlocks because the actor implementation features blocking operations. A common observable behavior of this concurrency bug is orphan messages. This means an actor with this issue is blocked, i.e., the process is in a waiting state.

Languages that use the *communicating event-loops* model, e.g., JavaScript and SOMns, do not provide blocking primitives, and thus, do not suffer from communication deadlocks. However, the rest of lack of progress issues such as behavioral deadlocks and livelocks can occur. Bug patterns for a behavioral deadlock or a livelock are typically mistakes in the code when processing a message, or a message that was sent to the wrong actor at the wrong time. The resulting observable behavior can be an incorrect program output in which one or more actors do not progress with their computation. As mentioned before, those bugs are hard to diagnose, because some actors are not blocked, but the overall program is not progressing.

2.4.3.2 Message Protocol Violations

Both variants of the actor model can suffer from message protocol violations. In the case of *processes* based programs, literature has reported message order violations and memory inconsistencies. For message order violations, possible bug patterns are the delays in managing responses or the unsupported interleaving of messages, i.e., the actor protocol does not correspond to the executed message interleavings. These can result in a program crash or inconsistent computational results. Memory inconsistencies are typically caused by a wrong message order when accessing shared resources.

Similar to the processes actor variant, we found in *event-loops* based programs message order violations and bad message interleavings. Our survey shows that JavaScript has more occurrences in these bug categories. Bug patterns are typically caused by high-level races between HTML parsing and events execution (e.g., user actions). A common observable behavior is exceptions or application crashes.

2.5 Related Studies of Concurrency Bugs in Actor-based Programs

As mentioned before, at the start of this work, few research has focused on categorizing bugs for actor-based programs. Recently, several field studies have been conducted on how bugs appear in actor-based programs. In particular, we have found two studies for Akka programs [HZ18, BFSK20] and one study about bugs in Node.js programs [WDG$^+$17]. Here we describe them and compare their bug categorization with ours.

2.5.1 Field Studies in Akka programs

Hedden et al. [HZ18] conducted a study categorizing 126 bugs in 12 real-world Akka systems in the context of distributed applications. They have used the ScalaIndex[2] website to search for the top 20 communities rated programs using Akka written in Scala language. Authors checked that each community had active issue tracking (i.e., GitHub) and contained issues with the labels “bug" and “closed," and that at least one git commit was done when closing it. They checked that the bug was resolved and the track issue closed with fixes to the code and a non-trivial solution, e.g., not a misspelling, missing documentation, or a mislabeled bug. They give priority to bugs related to erroneous behavior rather than to program development. They selected 12 systems with the mentioned requirements.

Authors classified the bugs found in three categories, i.e., *communication* (20,5%), *coordination* (22%) and *logic* (57,5%). Communication and coordination are bugs related to concurrency. Logic category refers to bugs that can be seen in any system, e.g., null pointer errors, optimization issues, etc. In Table 2.3 we show the subcategories they proposed for communication (from 1 to 4) and coordination (from 5 to 10) categories.

We observe in Hedden et al.'s categorization that most of their categories are related to issues in the context of distributed systems. Because our taxonomy is focused on bugs that can be seen in concurrent programs (i.e., without considering distribution), we only found one overlap in our work, i.e., our message order violation category is very similar to Hedden's subcategory of *message order*.

More recently, Bagherzadeh et al. [BFSK20] did an extensive study of 186 Akka concurrency bugs that have been reported in GitHub and Stack Overflow repositories. The authors collected 130 bugs from Stack Overflow, manually inspecting the questions with the tags “Akka" and “Actor." They consider the question related to a bug if the developer asking explains the error of their code (i.e., the expected and unexpected behavior). The authors consider that the bug was solved if the answer identifies a solution for the mentioned error. The remaining 56 bugs were collected from Scala and Java projects using Akka in GitHub. The repositories selected were rated with five stars by contributors and including the statement `import akka.actor.*`, to indicate that they used actors. In later step, the authors stem the words in the message commits and filter messages that contain the keywords “error", “bug",“fix", “issue", “mistake", “incorrect", “fault", “defect" and “flaw". Finally, the authors manually analyze each commit obtained to check if the bug and fix reported correspond truly to Akka actors. The authors classified the bugs found according to 10 bug patterns (or root causes). We summarize them in Table 2.4.

Now we compare the *bug patterns* we found in the literature (see Table 2.2) with Bagherzadeh et al. root causes, and the *observable behaviors* we found in the literature with Bagherzadeh et al. symptoms.

[2]https://index.scala-lang.org

Bug subcategory	Description
1. response	refers to issues that occur when an actor receives improper responses to different communication based operations, e.g., when try to connect to a not running database.
2. connection	refers to issues not related to sending messages to another actor, but to streaming services or file requests.
3. error handling	refers to issues that occur when connection and message errors are not handled properly, e.g., defining unlimited number of reconnections attempts can lead to a system deadlock.
4. message order	refers to issues due to out of order messages, i.e., actors receive messages in the wrong order.
5. cooperation	refers to issues that emerge when actors are executing simultaneously, although in its example authors mention an issue due to mixing actors with semaphores.
6. shutdown	refers to issues involving problematic shutdown process e.g., processes that remain with open connections at system termination.
7. recovery	refers to issues when actors recover or recreate after a failure, e.g., system did not recognize a worker after the worker dies and recover because an unsuccessful registration of the worker in the system.
8. workload	refers to issues caused by long or demanding computations, which results for example in “deadlocked" actors (i.e., while busy due to configured time out of windows), or getting multiple duplicated messages.
9. operation order	refers to issues that occur due to wrong order of operations not related to messages sent and received by actors, i.e., other operations carried out by the system such as when sending messages to uninitialized actors.
10. creation	refers to issues that occur when actors are incorrectly created, e.g., null references to actors when running multiple actor systems within a computer cluster.

Table 2.3: Categories for concurrency bugs proposed by [HZ18] in Akka programs.

Bug pattern	Description
1. logic	refers to issues caused by incorrect implementation of the logic of the program, e.g., operations performed over wrong variables causing undesired results.
2. race	refers to issues that occur when two conflicting computations have different execution and program orders.
3. API confusion	refers to issues related to incorrect use of Akka API classes.
4. explicit life cycle	refers to issues caused by incorrect implementation of actors lifecycle, e.g., an actor lifecycle in Akka consists of creation, initialization, lookup, monitoring, termination and restart.
5. programming	refers to incorrect implementation of syntatic and semantic elements of the programming language, e.g., incorrect dependencies and imports.
6. messaging patterns	refers to issues caused by incorrect use of Akka message patterns, e.g., request-response, forward, and route.
7. model confusion	refers to issues caused by incorrect implementation of the semantics of programs based on the actor model, because developers confuse its semantics with threads, e.g., exception handling for Scala actors and Akka actors is different.
8. misnaming	refers to issues caused by incorrect actor naming.
9. misconfiguration	refers to issues caused by incorrect use of actors configuration parameters in Akka, e.g., dispatching, shutdown, mailbox, etc.
10. untyped communication	refers to issues caused by incorrect discovering of message types in the implementation of the program, i.e., because actors do not know the type of message they may receive.

Table 2.4: Bug patterns proposed by [BFSK20] for concurrency bugs in Akka repositories.

Observable behavior	Description
1. error	an actor or the application throws an error (e.g., timeout, out-of-memory and actor name errors) or exception during its execution or compilation.
2. unexpected behavior	an actor or its enclosing application does not behave in a way that the developer expects from its implementation, e.g., misbehaving schedulers, dispatchers, supervisor actors, loggers, etc.
3. incorrect messaging	an expected message is not sent by an intended sender of the message or not received, stashed or processed by its intended receiver.
4. incorrect termination	an actor or its enclosing application does not terminate, terminates prematurely, terminates and restarts infinitely, or hangs.
5. incorrect exceptions	an intended actor does not throw, catch, or properly handle an expected exception.

Table 2.5: Observable behaviors proposed by [BFSK20] for concurrency bugs in Akka repositories.

In Table 2.4 we consider the majority of bug patterns (i.e., 1, 3, 4, 5, 7, 8, 9) not to be related to actor-based programming. Instead, those bug patterns are related to programming with Akka and incorrect implementation of the program, which, although could lead to bugs, are not directly related to the interaction of the concurrent entities, i.e., actors and messages. Bug pattern 2, 6 and 10 are related to actor programming and we now relate them to our taxonomy. First, *bug pattern 2* is similar to our category of message order violation. However, as we mentioned before Section 2.3.4, there are special cases in Scala in which concurrency bugs from thread-based model can occur, e.g., data races when using futures and actors. Although there is not a direct match of *bug pattern 6* in our taxonomy, we think this is similar to the bug patterns we found in our communication deadlocks examples in Erlang programs. Also, we think that the *bug pattern 10* can be manifested by orphan messages and lead to communication deadlocks in Akka.

In their work, Bagherzadeh et al. also classified their bugs considering 5 observable behaviors (or symptoms). We summarize them in Table 2.5. Comparing the observable behaviors in Table 2.5 with the observable behaviors we have found in literature, Table 2.2), we see that our taxonomy is aligned with the findings of [BFSK20]. In particular, we have seen in literature errors and exceptions (*observable behavior 1*), incorrect termination (*observable behavior 4*) such as crashes and exit abnormally of the applications, e.g., for the message order violation and bad message interleavings reported in JavaScript. Also, in literature have been reported misbehaviors such as incorrect messaging (*observable behavior 3*), e.g., in communication deadlocks in Erlang programs. However, we consider orphan messages more like a bug pattern than an observable behavior for the category of communication deadlocks. We consider observable behaviors 2 and 5 specific for Akka actors.

2.5.2 Field Study in Node.js programs

In the context of event-loops programs, Wang et al. [WDG$^+$17] studied 57 concurrency bugs in Node.js programs from GitHub repositories. Authors used keywords to search potential concurrency bugs, such as “concurrent", “race", “synchronization", “atomic", “mutex", “transaction", “deadlock", “compete" and “starve". They also used GitHub search filters labeled with “bug" tag and in a closed state. They also filter by the keywords “submit" and “fix". In a second step, the authors manually checked 1583 bug reports to exclude unrelated reports for concurrency bugs and obtained 57 complete reports about the bugs.

Wang et al. classified bugs found into three categories, i.e., *order violation, atomicity violation and starvation* (see Table 2.6). As mentioned in Section 2.3.3 we found that their order violation category is equivalent to our category of a message order violation. And, the atomicity violation category corresponds to what we call bad message interleavings. We consider what they call starvation a case of behavioral deadlock.

Bug category	Description
order violation	violation order between two or more events (i.e., asynchronous operations or callbacks). The events access the same shared resources (e.g., variables, files) and are expected to be processed in a certain order.
atomicity violation	violation among callbacks or asynchronous operations that should be executed atomically without interruption.
starvation	happen when some tasks take a long time and prevent other events from processing. Usually, callbacks registered by higher priority can starve the lower ones.

Table 2.6: Bug categories proposed by [WDG+17] for concurrency bugs in Node.js repositories.

Bug pattern	Description
1. non-deterministic event triggering	refer to issues that emerge when two asynchronous operations are scheduled by the worker pool their completion order is unknown.
2. non-deterministic event handling	the schedule of one callback execution from multiple available callbacks is non-deterministic, i.e., they may not be processed in the expected order and cause issues.
3. non-deterministic execution of asynchronous operations	refer to issues that emerge when multiple asynchronous tasks are delegated to the worker pool simultaneously, their processing order is unknown.
4. non-determinism among multiple processes	Node.js is single-thread but it can create many processes to distribute the workload. These processes may have conflicting accesses to the same resources, e.g., copy and delete actions of a directory from different processes.
5. scheduling and protocol APIs	refer to issues that originates due to incorrect usage of Node.js asynchronous APIs, i.e., schedule APIs and event-driven specification protocol.

Table 2.7: Bug patterns proposed by [WDG+17] for concurrency bugs in Node.js repositories.

Similar to Bagherzadeh et al. [BFSK20], Wang et al. identified bug patterns (i.e., faults or root causes) shown in Table 2.7, and observable behaviors (i.e., failures) shown in Table 2.8 for the concurrency bugs.

The authors reported having found bug patterns 1, 3, 4, 5 in the categories of order violation and atomicity. Bug pattern 2 was seen in a lower number of bugs in the category of starvation. From the five bug patterns of Table 2.7 all are similar to our taxonomy except *bug pattern 5*, which is specific to Node.js APIs. We consider *bug pattern 1, 2, and 3* characteristics bug patterns that can cause message order violations and bad message interleaving. *Bug pattern 4* is related to our memory inconsistency category.

From the classification of observable behaviors in Table 2.8, we found similarities in 3 of them, i.e., in the bugs reported in the literature, authors also mentioned *behaviors 1, 2 and 4*. However, we do not have in our catalog concrete examples for behaviors 3 and 5, which certainly can occur in actor-based programs.

Observable behavior	Description
1. crashes/ exceptions	e.g., null pointer exceptions, or crashes due to out of memory.
2. incorrect states	e.g., incorrect persistence data in database.
3. wrong outputs	refers to wrong results of the program which are shown to users.
4. hangs/ no response	e.g., when a listener is removed from an event before it is triggered, the event cannot be processed correctly.
5. operation failures	refers to unexpected behaviors, e.g., jobs getting processed incompletely or rejected, I/O starvation issues under heavy load.

Table 2.8: Observable behaviors proposed by [WDG$^+$17] for concurrency bugs in Node.js repositories.

2.5.3 Conclusion from the Related Field Studies

We draw two conclusions from the recent field studies of issues in repositories of Akka and Node.js with respect to our taxonomy.

First, the bug patterns we have found in the literature for our three categories of message protocol violations (see Table 2.2), have also been reported in repositories of Akka and Node.js programs in GitHub. Bug patterns that cause memory inconsistencies problems are particular to Erlang programs but can be compared with some Akka and Node.js races.

Second, the observable behaviors we have seen in literature have also been reported for Akka and Node.js programs (see Table 2.2).

All in all, the categorization introduced by our work provides thus a solid basis to categorize concurrency bugs in real-world applications. We aim to keep our catalog of bugs up to date as more field studies are conducted.

2.6 Heisenbugs and Probe-Effect

So far, we have focused on bugs manifested by an actor-based program. However, some bugs may only manifest when the program is being externally controlled, e.g., by a debugging tool. Those kind of bugs are typically known as *heisenbugs* [sig83, Gra86]. A heisenbug has been referred to as a concurrency bug that “disappears" in the next program's execution [sig83, Gra86]. In other words, are bugs that, due to timing issues originated by external tools such as debuggers, can be seen in one run of the program (i.e., in one path of the program execution), and later is very difficult to see it again because the output or behavior of the program where the bug was observed cannot be reproduced again.

The fact that observing the program execution affects the program behavior is known as the *probe-effect*. Gait initially defined the probe-effect as: *“a characteristic behaviour of the execution trace of an incorrectly synchronized concurrent program when extrane-*

ous delays are introduced" [Gai86]. Gait summarized two causes for the probe-effect. First, the interactions between application programs and multiprocessor operating systems, and second, the effect debuggers may have in the program's execution by altering synchronisation actions [Gai86]. This last means that the debugger's mere presence (e.g., adding print statements or executing debugging commands) may affect the order in which concurrent entities are executed, making the reproduction of a concurrent bug even more difficult [MH89]. This idea that measuring something can change the measurement itself recalls the Heisenberg Uncertainty principle [Hei27][3].

2.7 Conclusion

In this chapter, we explained the main concepts of the Communicating Event-Loops concurrency model, the actor model variant to which we address our research. We proposed a taxonomy of concurrency bugs for actor-based programs to enable research on debugging support for actor-based programs. Although the actor model avoids data races and deadlocks by design, it is still possible to have lack of progress issues and message-level race conditions in actor-based programs.

Our literature review shows that actor-based programs exhibit a range of different issues depending on the specific actor model variant. In languages that feature the processes actor variant like Akka in Scala and Erlang, programs can suffer from communication deadlocks because the actor implementation uses blocking operations. In languages that implement the event-loops concurrency model, this issue cannot occur. However, they can suffer from other lack of progress issues such as behavioral deadlocks and livelocks. Behavioral deadlocks and livelocks are really hard to identify because actors are not blocked. Both lack of progress issues can be seen in all variants of the actor model. Message order violations, bad message interleaving, and memory inconsistencies are message protocol violations that can also happen in programs that implement any variants of the actor model.

We also discussed three recent related field studies that proposed other taxonomies for actor-based programs from repositories of mainstream languages such as Akka and Node.js. We found some bug patterns and observable behaviors in the studies, which as our taxonomy are focused in concurrency and have overlaps in our classification. Other bug categories are rather specific of an actor language or focus on issues that can happen in distributed environments.

In the next chapter, we will review the state-of-the-art techniques for identifying and handling concurrency bugs in actor-based programs and how they are related to the concurrency bugs we have studied in this chapter.

[3]The Heisenberg principle for subatomic particles says that *"that the position and the velocity of an object cannot both be measured exactly, at the same time, even in theory"* [Bri20].

Chapter 3

State of the Art of Techniques to Handle Concurrency Bugs in Actor-based Programs

This chapter surveys the current state of the art of techniques to identify and solve concurrency bugs in actor-based programs. Furthermore, we analyzed how the techniques relate to the bug categories of our taxonomy to identify open issues. Specifically, we survey techniques for static analysis, testing tools, visualization, and debuggers.

3.1 Identifying and Solving Concurrency Bugs

In the tasks of identifying and solving concurrency bugs, we can find tools in two categories, static and dynamic [MH89, Zel09]. On the one hand, static techniques, such as *model checking* identify concurrency issues without executing the program [CHVB18]. On the other hand, dynamic analysis techniques such as testing and debugging "evaluate a system or a component based on its behavior during execution" [glo90]. Specifically, *testing tools* require the implementation (or generation) of code that tests the main functionalities of the program [BMP18]. Alternatively, *debuggers* do not require additional code, but they run the program in a controlled way. Debugging techniques are applied directly to the execution of the target program. Debuggers have the benefit of starting the program inspection fast without changing the source code, which allows developers to have immediate results [Zel09].

Here we cite definitions of the three mentioned techniques from the standard glossary of IEEE [glo90]:

Static analysis "the process of evaluating a system or a component based on its form, structure, content, or documentation".

Testing "the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component". Also, "the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items".

Debug "to detect, locate, and correct faults in a computer program. Techniques include use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operation and traces".

In our research, we focus on debugging techniques that allow developers to explore the program execution interactively to find the root cause of a bug when they do not have a clear idea of what the fault is.

3.2 Debugging Techniques

For more than three decades, researchers have emphasized that the main problems associated with debugging concurrent programs are the increased program complexity, *non-determinism* and *non-repeatability* [MH89]. Understanding concurrent programs is complex because it requires understanding interactions amongst concurrent entities. On the other hand, the non-deterministic behavior of concurrent programs makes that the output of a program does not only depend on the input but also on the scheduling of concurrent entities. Moreover, communication or synchronization between concurrent entities can also be sensitive to timing, affecting the order in which concurrent entities execute operations. Finally, the problem with non-repeatability is that running a program several times with the same input may not reproduce the bug as it may only manifest on weird schedulings of concurrent operations. Non-repeatability occurs mostly due to races between the concurrent entities [MH89]. Control over sources of non-determinism is thus important for achieving the reproducibility of bugs. But, as explained in Section 2.6, the debugging tool may introduce more non-determinism, e.g., timing differences, which makes that the bug does not manifest, requiring many debugging cycles before being able to reproduce the bug.

Debugging tools for parallel and concurrent programs can be categorized into two main families [MH89]:

- *Event-based debuggers* also known as log-based, offline, or postmortem debuggers.
- *Breakpoint-based debuggers* also known as online, cyclic 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 and resume program execution at well-defined points, inspect program state, and perform step-by-step execution. In the remainder of this section, we analyze the features of each family and how they help to handle concurrency bugs. We refer to event-based approaches as *offline debugging* techniques and breakpoint-based approaches as *online debugging* techniques.

3.2.1 Online Debugging Techniques

Online debuggers are software tools that allow developers to inspect the program state interactively, i.e., pausing and resuming the program execution at one (or more) point of interest (typically called *breakpoints*). Breakpoints is a well-known debugging feature by today's developers, which was actually invented for the ENIAC computer more than 70 years ago: *"literally the removal of a wire to break the flow of program pulses...to halt calculations at particular points so that values stored in memory could be checked"* [HPR16]. With the hardware evolution during the years, debuggers have integrated different types of breakpoints, such as line and conditional breakpoints, as well as other online features such as variables state inspection, and stack traces.

In particular, online debuggers for actor-based programs have adapted the aforementioned techniques from sequential debugging to the concepts of concurrent programming. We summarize three online debugging techniques that have been applied in the context of actors, i.e., *actor state inspection*, *breakpoints* (line, conditional and message-breakpoints) and *stepping operations* (sequential and message-oriented), and *asynchronous stack traces.* Here we detail each of them:

Actor state inspection in the context of actor-based programs refers to inspecting the state of the objects inside the actor and the messages enqueued in its mailbox. Examples of debuggers that provide this feature are REME-D for the AmbientTalk language [GBNDM14], and IDeA for the Akka framework in Scala [MOM18].

Breakpoints and stepping operations in the context of actor-based programs, there are interesting locations in which developers can inspect the programs' state using breakpoints. We can make use of line breakpoints and conditional breakpoints, which are often used in sequential debugging. However, pausing the program on the level of messages, promises or turns can be of great benefit. For example, REME-D debugger employed the concept of message-oriented breakpoints based on the work of [Wis97]. Furthermore, stepping operations that cross the boundaries of turns will allow to inspect an actor state, for example, at the end of the current turn, in the next turn, or return to the turn where a promise is resolved.

Asynchronous stack traces in the context of actor-based programs, an asynchronous stack provides the concept of a call stack from the sequential world into the concurrent one. They show the sequence of asynchronous messages that were processed until the point of the suspension. For example, the Scala debugger provides an asynchronous stack for programs written with Akka [Dra13] and Chrome debugger [Not17] for JavaScript programs.

We now discuss how those online techniques could help to handle concurrency bugs. For example, *inspecting the actor state* could help identify behavioral deadlocks, e.g., inspecting the promise object state if it is successfully resolved or not may help to detect a deadlock due to an unresolved promise. Besides, inspecting variables state may help identify value inconsistencies that are often produced by message order violations or bad message interleavings.

Breakpoints combined with actor state inspection are useful for observing the state of one or multiple actors in a program with a failure. For example, pausing the program before a message is sent can help developers to identify a wrong value that is about to be sent in the argument of the message, which can cause a behavioral deadlock or a message order violation.

Finally, we consider that *asynchronous stack traces* can be especially helpful for message order violations and bad message interleavings because it can help to detect incorrectly (or missing) asynchronous messages sent.

3.2.2 Offline Debugging Techniques

Offline debugging techniques allow developers to explore the program execution based on a trace that contains all the events of the program after it has been executed. Examples of offline debugging techniques are *record and replay*, *reverse debugging*, *actor state inspection* and *visualization*. In the following, we describe each of the four offline debugging techniques which have been applied in the context of actors:

Record and replay this technique consists of recording (all or fragments) events of a program execution in a trace and then re-execute the events again from the trace. In actor-based programs, events can be the creation of an actor, messages sent, messages processed, etc. [AMB$^+$18]. This technique eases the problem of non-repeatability due to non-determinism. Because replaying the trace events deterministically, i.e., replaying the same path of execution in which the bug was observed, will allow developers to observe the same program misbehavior immediately. Examples of debuggers that apply this technique are Jardis, a debugger for JavaScript programs [BMM$^+$16] and IDeA debugger for Akka [MOM18].

Reverse debugging also known as back-in-time debugging or omniscient debugging, it is a debugging technique that allows to go forward and backward into the history of the program execution to inspect the program state [Eng12]. More concretely, the debugger reads the events from a trace, i.e., the state is restored from a full log or snapshots (or checkpoints) previously recorded. Different from the record and replay technique, developers can step backward until a breakpoint hits, i.e., observing a past state in the program interactively. The debuggers Jardis and IDeA also apply reverse debugging together with record and replay.

Actor state inspection similar to online debuggers, we can find offline debuggers that allow the state inspection of the actor objects and the actor mailbox. The difference is that in offline approaches, the state is previously recorded, and setting breakpoints and triggering stepping operations is done over the recorded trace. For example, the IDeA debugger allows inspecting the actor local variables and its mailbox.

Visualization offline debuggers typically provide views to visualize messages exchanged between the actors of the application. Data is obtained from a previously recorded trace. For example, the Causeway debugger [SCM09] for the E language introduced views in which messages are shown in chronological order by each process in a tree, which is built based on program events, e.g., asynchronous message sends, recorded during the program execution.

We now discuss how those offline techniques could help handle concurrency bugs. *Recording and replaying* the program execution can be effective in the context of actors, for example, for identifying heisenbugs, because if the bug was recorded, then it is possible to reproduce the program execution deterministically. In contrast, in a cyclic debugging approach, it can take more than one run of the program to see it again. Thus, it will be possible to reproduce as well lack of progress issues and message protocol violations.

Reverse debugging techniques could be helpful to find the root cause of concurrent bugs because literature has reported that the distance between the failure and the root cause is long in concurrent programs [PSTH16].

The technique of *actor state inspection*, like for online debuggers, allow inspecting variables state that could help discover root causes for lack of progress issues or message protocol violations.

Visualization of messages exchanged between actors can be useful to unveil complex paths of message sends that could aid in solving message protocol violations. Unfortunately, offline visualization on traces often does not capture enough information to find the root cause of a bug.

3.3 State of the Art Techniques to Handle Concurrency Bugs

This section reviews state of the art for identifying and solving concurrency bugs in actor-based programs. We distinguish techniques between online debuggers and offline debuggers. We also survey visualization techniques and interesting works in the fields of static analysis and testing. We finish this section by discussing the advantages and disadvantages of each technique, and we summarize the approaches by our taxonomy of concurrency bugs.

3.3.1 Online Debuggers

In this section, we will analyze the existing online debuggers for research actor-based languages (e.g., E, AmbientTalk) or for mainstream languages (e.g., JavaScript, Erlang, Scala).

REME-D [GBNDM14] is an online debugger for distributed communicating event-loops programs written in AmbientTalk [VMG^{+}07]. REME-D provides message-oriented debugging techniques such as *actor state inspection*, in which the developer can inspect an actor's mailbox and objects while the actor is suspended. It also supports a catalog of breakpoints, which can be set on asynchronous and promise messages (or future messages in AmbientTalk) sent between actors. REME-D allows inspecting the history of messages sent and received when an actor is suspended, also known as *causal link browsing* [GBNDM14].

In the context of JavaScript, the Chrome DevTools online debugger supports Web Workers [Not17] which are actors that communicate with the main actor through message passing. The Chrome debugger allows pausing workers. In the case of shared workers, it also provides mechanisms to inspect, terminate, and set breakpoints [Blo12]. Chrome also supports *asynchronous stack traces*. This means it shows the stack at the point a callback was scheduled on the event-loop. Since this works transitively, it allows inferring the point and context of how a callback got executed.

Erlang has an online debugger [AB20] that supports *line, conditional, and function breakpoints*. The Erlang processes can be inspected from a list, and for each process, a view with its current state, as well as its current location in the code, can be opened, which allows one to inspect and interact with each process independently. It also supports stepping through processes and inspecting their state.

The ScalaIDE offers a classic online debugger with support for stepping, line, and conditional breakpoints [fED]. Furthermore, one can follow a message send and *stop in the receiving actor*. Additionally, the debugger supports asynchronous stack traces similar to the Chrome debugger. In particular, Dragos [Dra13] proposes two interesting points to save asynchronous information, i.e., when a *promise is created* and when an *actor message is sent*. In their approach, the stack trace data relies on static type information at a breakpoint source location. Besides, it shows the state at the moment when the message was sent.

All in all, online techniques aid in identifying the root cause of bugs by allowing developers to interactively control the execution of program and inspect the program state at interesting points, e.g., at the receiving actor before processing a message. However, they suffer from the probe effect since the mere presence of the debugger may affect the program execution, making that bugs do not manifest during the debugging session. So far, most of the efforts in online debugging techniques have focused on adding some support for debugging messages (e.g., message-oriented breakpoints) and asynchronous stack traces. However, we observe very few integrations between the debugging operations for concurrent code and the ones for sequential one. Since a concurrency bug can manifest due to a combination of erroneous program states and erroneous interactions amongst actors, finding the root cause of a bug requires controlling both executions of sequential computation within a turn as well as interactions amongst actors. Integrating online techniques such as breakpoints and stepping operations from the sequential and concurrent world is thus needed for handling concurrency bugs.

3.3.2 Offline Debuggers

In this section, we describe different debugging tools for actor-based programs that use offline debugging techniques. Causeway is a postmortem debugger for E programming language that pioneered a message-oriented approach to debugging CEL programs which followed the flow of messages across actor boundaries based on the *happened-before relationship* [SCM09]. It focuses on displaying the *causal relation of messages* to enable developers to determine the cause of a bug. Causality is modeled as the partial order of events based on Lamport's happened-before relationship [Lam78].

Actoverse debugger [SW17] enables *reverse debugging* of Akka programs written in Scala. It uses snapshots for restoring the state of actors and enables back-in-time debugging in a postmortem mode. Furthermore, Actoverse provides message-oriented breakpoints and a message timeline that visualizes the messages exchanged by actors, similar to a sequence diagram. The authors aim to ease finding the cause of message protocol violations in Akka programs.

The recently proposed IDeA debugger [MOM18] provides a 3D interface in virtual reality for developers to debug Akka programs using traces. It is a postmortem approach because it *records traces for later replay* the program's execution deterministically. Using the trace allows developers to interact with the program, setting breakpoints and querying the actor state. Besides, it provides a list of steps of the program execution in the past, allowing developers to select them and *inspect the actor state* at that point in time. Furthermore, it provides a visualization to track the causal relationship of messages sent.

For the JavaScript/Node.js languages, there are debugging approaches focused on enabling time-traveling debugging features based on recording and replaying snapshots or checkpoints of the program state [BM14, BMM^{+}16, VBMM18]. The most recent implementation is McFly, a novel time-traveling debugger for web applications [VBMM18]. It allows to *step forwards* using a log to replay the program's execution deterministically. For *stepping backward* it uses two monitors to determine the statement and the time the developer wants to inspect in the past. The first monitor contains the last branch instruction taken by each function in the call stack and a second monitor that includes the timestamp of each function in the call stack. Furthermore, McFly supports visual state by *checkpointing and logging* changes to a high-level representation of the layout engine's visual state. Checkpoints are object graphs representing the application program state from the JavaScript engine and a visual state from the layout engine. Also, it ensures the time-traveling is deterministic through logs of I/O operations.

All in all, offline debugging techniques can ease the problem of non-repeatability since once the bug is captured, the program can be deterministically replayed. The record and replay technique, for example, has been considered an intermediate form between cyclic and reverse debugging [Eng12] because it can trigger breakpoints and replay a program execution from a trace. However, offline techniques usually incur a *high overhead*. Besides, the information captured is often not enough to identify the root cause of a concurrency bug [PSTH16]. Often developers cannot inspect past states without replaying again the program trace [Eng12]. Reverse debuggers can alleviate these issues by offering backward stepping operations. But they are not widely used in practice yet, mainly due to the overhead of tracing strategies, e.g., access to long traces tends to be slow [BJC$^+$13, PSTH16]. In this context, techniques that allow the efficient exploration of the program execution in both directions, i.e., forward and backward, are needed.

3.3.3 Visualization Techniques

This section discusses mechanisms and approaches to visualize actor-based systems. We group each approach based on either it is implemented on a debugger or it is implemented as a standalone tool.

3.3.3.1 Visualization in Non-debugging Tools

Miriyala et al. [MAS92] proposed the use of *predicate transition nets* for visualizing actors execution. Based on the classic model of actors, the approach focuses on the representation of the actor's behavior and sent messages. The activation of each transition in the Petri net corresponds to a behavior execution. The main idea is that the user interacts with a visual editor for building the execution of an actor system in the Petri net. The authors emphasize that the order of net transitions should be represented in the same order as the execution of messages of the actor system.

Coscas et al. [CFL95] presented a similar approach in which the predicate transition nets are used to simulate actors execution in a step by step mode. When a user fires a specific transition, he or she only observes a small part of the whole net. The approach also verifies messages that do not match with the ones expected by the actor, i.e., messages that do not match the actor's interface.

An interesting visualization approach based on dynamic program analysis has been proposed for promises in asynchronous JavaScript programs [AZMT18]. The authors argue that the use of *visual graphs* could help developers to understand the flow of execution of promises and also identify anti-patterns, i.e., incorrect usages of promises such as unresolved promises and missing reactions for broken (or ruined) promises. Specifically, they use directed graphs to show control and data flow dependencies. Their dynamic graphs represent promises that are created, resolved, and unresolved during one program execution. Figure 3.1 shows a promise graph with an example of a promise exception not being handled.

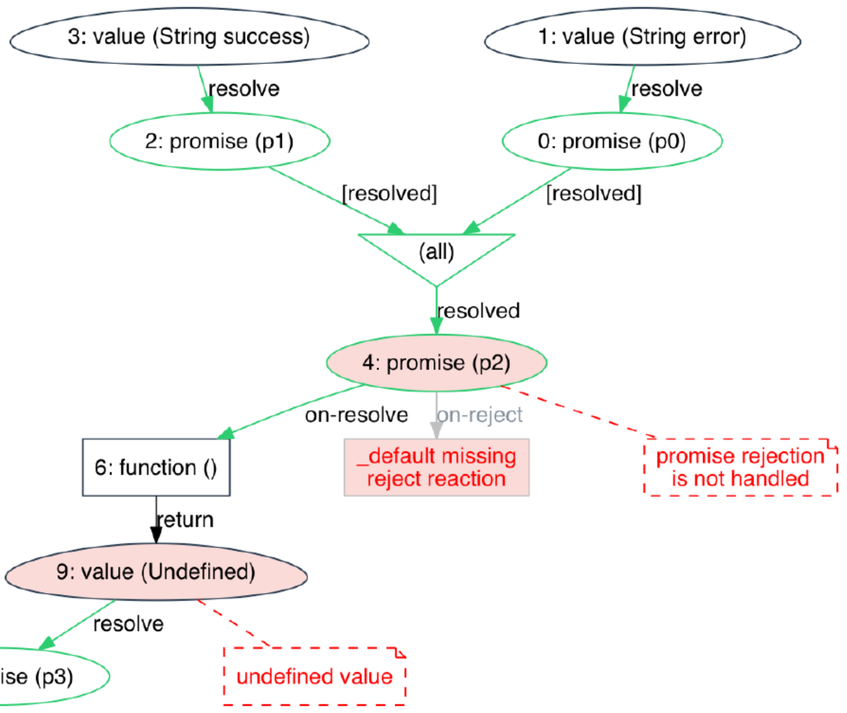
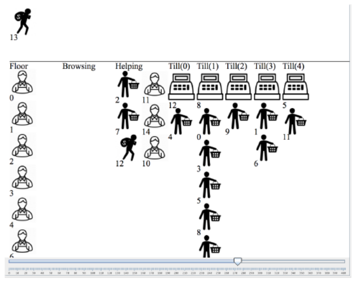


Figure 3.1: A visualized promise graph, extracted from [AZMT18].

Sun et al. [SBSB19] proposed *asynchronous graphs* to visualize the asynchronous flow of execution of a Node.js application. Their tool uses instrumentation techniques to detect bugs automatically caused by the incorrect usage of a combination of different event-based APIs, such as the ones related to callback scheduling of promises and emitters. Each node in the graph belongs to a single execution of an event-loop and can be of different types, e.g., callback registrations, callbacks execution, and object binding. Unlike the previous approach, [AZMT18], asynchronous graphs aim to capture not only promises but also all sources of asynchronous execution, e.g., callbacks executed due to self-scheduling (APIs including promises) and external scheduling (I/O or timing events happening in the operative system).

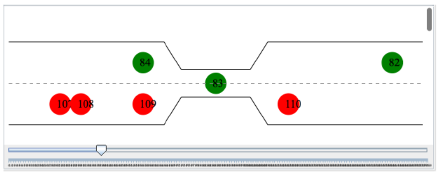
Clark et al. [CBKB19] proposed a visualization based on the semantics of event histories of an actor language. Specifically, they proposed a *filmstrips pattern* visualization and implemented it for ESL, an actor language that follows the semantics for the classic model of actors [NA96]. A filmstrip is denoted as a sequence of snapshots of objects and relationships of the program described in state transitions derived from system operation calls. Their approach translates messages into state transitions, state transitions into semantic values, and semantic values are represented as sequences of displays. A display consists of a 2D board containing a collection of trees with boxes, shapes, and images. Although their research context is multi-agent-based systems, the authors argue that a filmstrip gives a visual semantic description of the system that can help developers identify the application issues. Figure 3.2 shows filmstrip examples.



(a) Shop Filmstrip

```
type Cid          = Int;
type Aid          = Int;
type Tid          = Int;
data ShopE        = NotInShop(Cid)
                  | Browsing(Cid)
                  | Queueing(Cid,Tid)
                  | SeekingHelp(Cid)
                  | GettingHelp(Cid,Aid)
                  | OnFloor(Aid)
                  | AtTill(Aid,Tid);
data Helping      = Help(Cid,Possibly[AId]);
data Possibly[T] = Just(T) | Nothing;
data Till         = Till(Tid,Possibly[Aid],[Cid]);
data ShopS        = Shop([Cid],[Aid],[Cid],[Helping],[Till]);
```

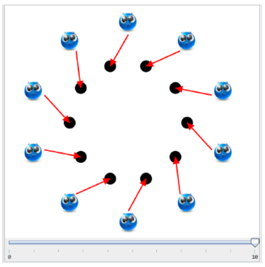
(b) Shop Event and Semantic Domains



(c) Traffic Filmstrip

```
type Vid          = Int;
data TrafficLight = Left | Right;
data Colour       = Red | Amber | Green;
data RoadE        = QueueLeft(Vid)
                  | QueueRight(Vid)
                  | Advance(Vid)
                  | LeaveLeft(Vid)
                  | LeaveRight(Vid)
                  | Change(TrafficLight,Color);
data Road         = Road([Vid],[Vid]);
data RoadS = Road(Colour,Colour,Road,Possibly[Vid],Road);
```

(d) Traffic Event and Semantic Domains



(e) Dining Philosophers Filmstrip

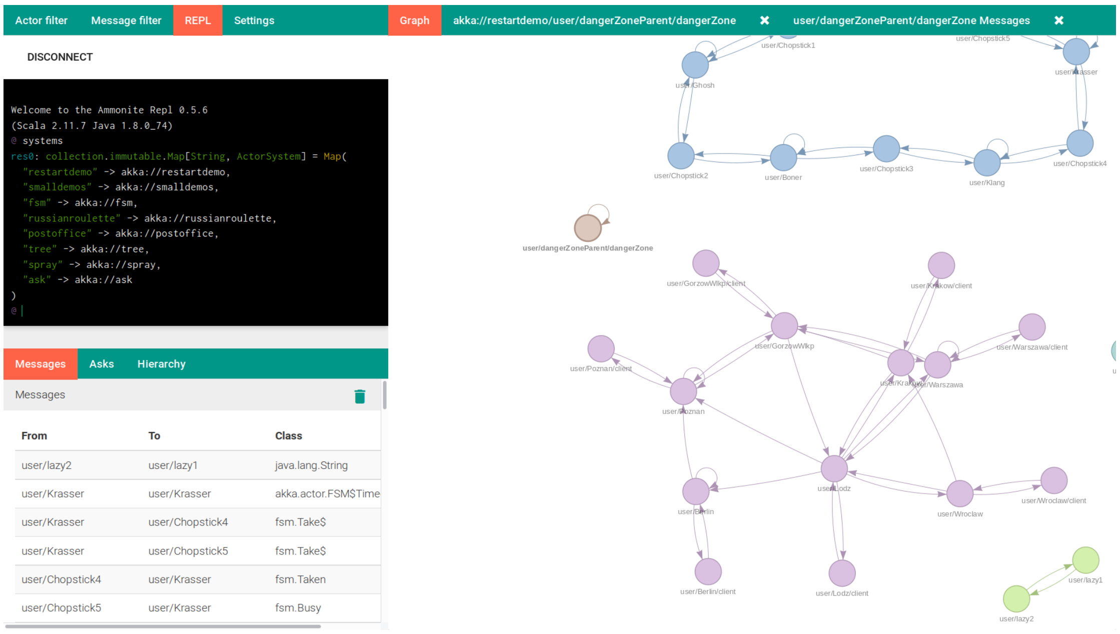
```
type Pid          = Int;
type Fid          = Int;
type Forks        = [Fid];
data Side         = Left | Right;
data Philosopher  = Phil(Pid,Possibly[Fid],Possibly[Fid]);
type Philosophers = [Philosopher]
data DinerE       = Pickup(Philosopher,Fid,Side)
                  | Eat(Pid)
                  | DropFork(Philosopher,Fid,Side);
data DinerS       = Dining(Forks,Philosophers);
```

(f) Dining Philosopher Event and Semantic Domains

Figure 3.2: Filmstrip examples, extracted from [CBKB19].

An interesting approach has been developed to represent Akka actors through *animated flow visualizations* [Lig21]. The actor system view shows inter-actor communication and actor-node communication, and it allows developers to identify easily dead letters messages, i.e., messages sent to an actor that is terminated before receiving it (see Section 2.1.1). On the other hand, it has the disadvantage that the UI does not represent an accurate representation of the message flow.

Another proposal for Akka programs is Akka-viz [Av16], an experimental tool that visualizes on a *graph* messages exchanged between actors. It allows filtering messages by class, monitor actor creation and display information about exceptions in actors. A similar graph representation was developed by [web16]. In contrast to Akka-viz, it does not show the actor's state, but only its mailbox size. Figure 3.3 shows a screenshot of the Akka-viz tool.



```
Welcome to the Ammonite Repl 0.5.6
(Scala 2.11.7 Java 1.8.0_74)
@ systems
res0: collection.immutable.Map[String, ActorSystem] = Map(
  "restartdemo" -> akka://restartdemo,
  "smalldemos" -> akka://smalldemos,
  "fsm" -> akka://fsm,
  "russianroulette" -> akka://russianroulette,
  "postoffice" -> akka://postoffice,
  "tree" -> akka://tree,
  "spray" -> akka://spray,
  "ask" -> akka://ask
)
@ |
```

Messages

From	To	Class
user/lazy2	user/lazy1	java.lang.String
user/Krasser	user/Krasser	akka.actor.FSM$Time
user/Krasser	user/Chopstick4	fsm.Take$
user/Krasser	user/Chopstick5	fsm.Take$
user/Chopstick4	user/Krasser	fsm.Taken
user/Chopstick5	user/Krasser	fsm.Busy

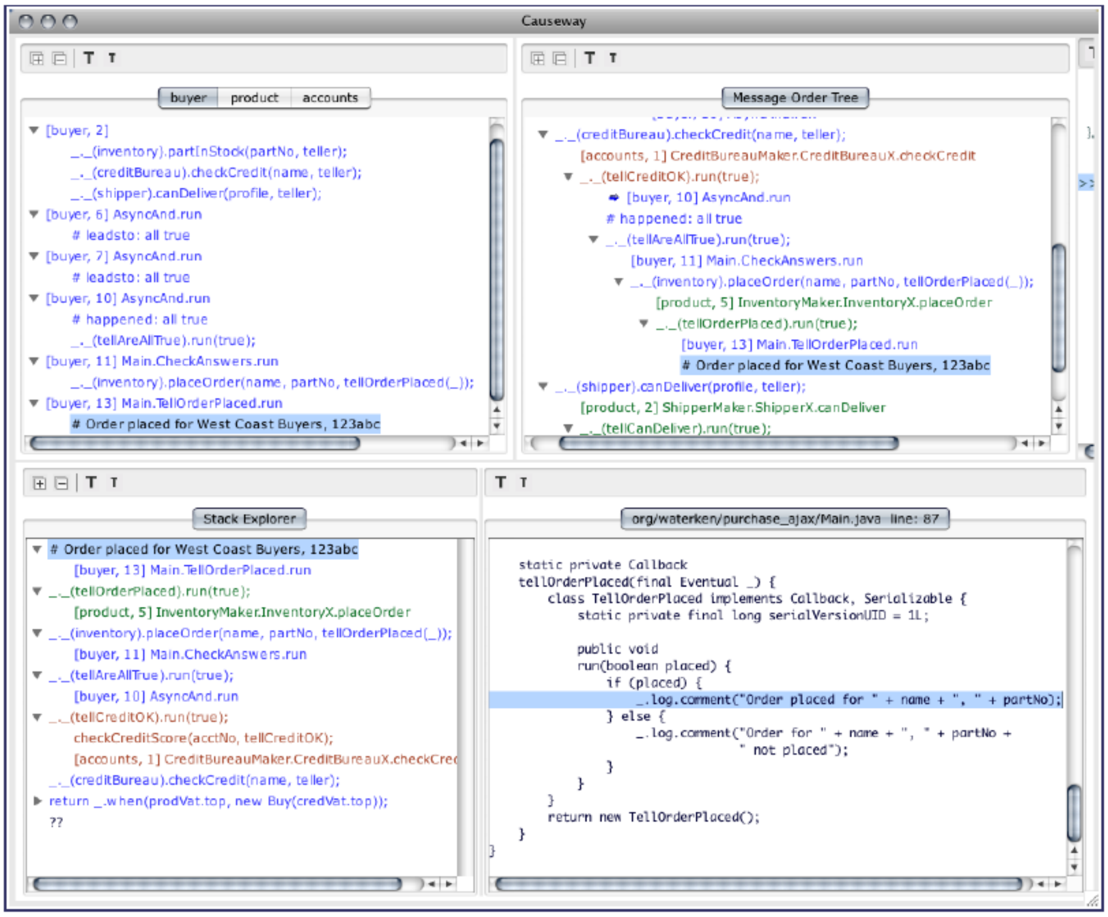
Figure 3.3: Screenshot of Akka-viz visualization for a dining philosophers program, extracted from [Av16].

Colak et al. [CC19] recently created AkkaVisual, a web-based tool, which consists of an actor model visualization for teaching programming with actors to computer science students. Their proposal shows information about actors and messages, e.g., sender, receiver, message content, and type of message sent by an actor. The authors argue that a *timeline view* is better than a graph view to show the order in which each message was sent. They did a user study with a group of 19 students to evaluate their tool. An interesting result is that participants said they would like to have a visualization that is possible to record and replay.

3.3.3.2 Visualization as Part of a Debugger

The Causeway debugger visualizes the program's execution based on views for *process order, message order, stack, and source code view* (see Figure 3.4) [SCM09]. The *process order* view shows all messages executed for each actor in chronological order, e.g., a parent item with asynchronous message sends. The *message order* view shows the causal messages for a message sent, i.e., other messages that have been executed before the message was sent and provoked the sent of the message we want to debug. In this view, it is also possible to distinguish processes by color, which helps users to visualize when a message flow (known as activation order) corresponds to a different process.

The *stack view* shows the activation order for a message sent, i.e., asynchronous sends and immediate calls that originated the message selected in the process order. The *source code view* shows the code where the message was sent in the code. Thanks to the synchronization achieved between all the views, it is possible to transit through the messages related to the execution of the actor's behavior that led to the bug.



```
▼ [buyer, 2]
      _._(inventory).partInStock(partNo, teller);
      _._(creditBureau).checkCredit(name, teller);
      _._(shipper).canDeliver(profile, teller);
▼ [buyer, 6] AsyncAnd.run
      # leadsto: all true
▼ [buyer, 7] AsyncAnd.run
      # leadsto: all true
▼ [buyer, 10] AsyncAnd.run
      # happened: all true
      _._(tellAreAllTrue).run(true);
▼ [buyer, 11] Main.CheckAnswers.run
      _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
▼ [buyer, 13] Main.TellOrderPlaced.run
      # Order placed for West Coast Buyers, 123abc
```

```
▼ _._(creditBureau).checkCredit(name, teller);
    [accounts, 1] CreditBureauMaker.CreditBureauX.checkCredit
  ▼ _._(tellCreditOK).run(true);
        [buyer, 10] AsyncAnd.run
        # happened: all true
      ▼ _._(tellAreAllTrue).run(true);
          [buyer, 11] Main.CheckAnswers.run
        ▼ _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
            [product, 5] InventoryMaker.InventoryX.placeOrder
          ▼ _._(tellOrderPlaced).run(true);
              [buyer, 13] Main.TellOrderPlaced.run
              # Order placed for West Coast Buyers, 123abc
▼ _._(shipper).canDeliver(profile, teller);
    [product, 2] ShipperMaker.ShipperX.canDeliver
  ▼ _._(tellCanDeliver).run(true);
```

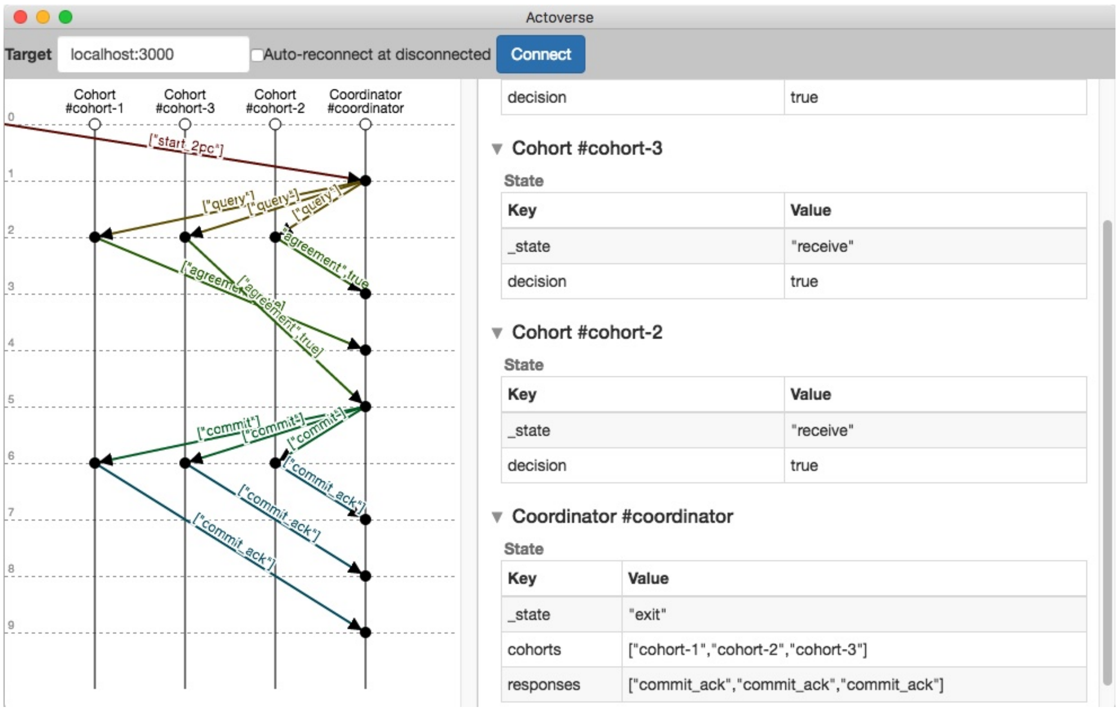
```
▼ # Order placed for West Coast Buyers, 123abc
      [buyer, 13] Main.TellOrderPlaced.run
▼ _._(tellOrderPlaced).run(true);
      [product, 5] InventoryMaker.InventoryX.placeOrder
▼ _._(inventory).placeOrder(name, partNo, tellOrderPlaced(_));
      [buyer, 11] Main.CheckAnswers.run
▼ _._(tellAreAllTrue).run(true);
      [buyer, 10] AsyncAnd.run
▼ _._(tellCreditOK).run(true);
      checkCreditScore(acctNo, tellCreditOK);
      [accounts, 1] CreditBureauMaker.CreditBureauX.checkCred
  _._(creditBureau).checkCredit(name, teller);
▶ return _.when(prodVat.top, new Buy(credVat.top));
  ??
```

```
    static private Callback
    tellOrderPlaced(final Eventual _) {
        class TellOrderPlaced implements Callback, Serializable {
            static private final long serialVersionUID = 1L;

            public void
            run(boolean placed) {
                if (placed) {
                    _.log.comment("Order placed for " + name + ", " + partNo);
                } else {
                    _.log.comment("Order for " + name + ", " + partNo +
                                  " not placed");
                }
            }
        }
        return new TellOrderPlaced();
    }
}
```

Figure 3.4: Causeway GUI. It shows the process order (left) and message order (right) views on the top, and on the bottom it shows the stack view (left) and source code view (right). Extracted from [SCM09].

As we mentioned in Section 3.3.2, Shibanai et al. [SW17] proposed a debugger for Akka programs. Their tool uses *sequence diagrams* to visualize messages sent in the program and provide developers a visual means for tracking causality. A vertical line represents each actor, and an arrow represents a message. The event in which an actor receives a message is represented as a point in the actor line. Clicking that point on the diagram will restore a past state of actors, and messages after that point are not persisted to avoid memory issues. The order of messages is computed using Lamport timestamps. Figure 3.5 shows a screenshot of the Actoverse debugger.



localhost:3000

decision	true

▼ Cohort #cohort-3

State

Key	Value
_state	"receive"
decision	true

▼ Cohort #cohort-2

State

Key	Value
_state	"receive"
decision	true

▼ Coordinator #coordinator

State

Key	Value
_state	"exit"
cohorts	["cohort-1","cohort-2","cohort-3"]
responses	["commit_ack","commit_ack","commit_ack"]

Figure 3.5: Screenshot of Actoverse debugger, extracted from [SW17].

As we mentioned in the previous section, the IDeA debugger [MOM18] uses *virtual reality* for debugging an Akka program at the message level. IDeA represents actors as geometric entities in 3D space and messages as arrows between the entities, which fade over time. The animation consists of traces that can be manipulated, e.g., displaying the sequence of messages exchanged between the actors, go back to a previous step in the execution, and enforce a selected actor to receive its next message. As visualization helpers, they allow to position actors on the workspace, as well as tracking the causality of messages (see Figure 3.6). They also mentioned how to suppress information, for example as selecting actors and discard messages in untracked actors. Moreover, the debugger provides a focus area to allow the developer to move a set of selected actors. The authors argue that an immersive visualization in 3D requires less effort than a 2D environment because the first involves only moving the eye-head-bearing, whereas the second environment requires an explicit interaction.

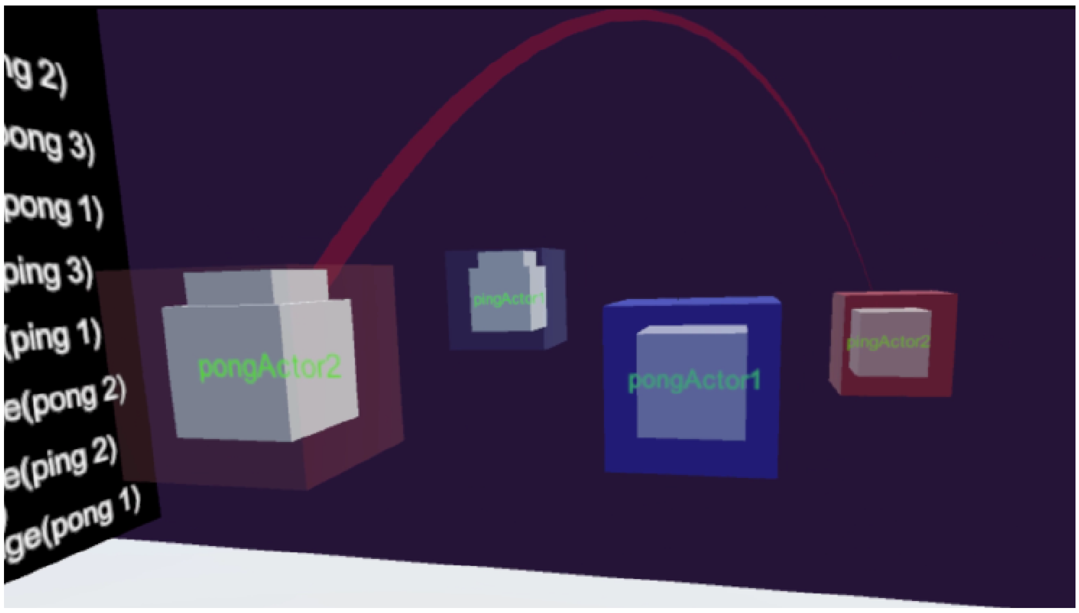


Figure 3.6: Tracking causality visualization in IDeA. Actor `pingActor2` sends a message to `pongActor2` which is marked in the same color (i.e., red). Extracted from [MOM18].

3.3.4 Static Analysis

The static approaches surveyed in this section focuses on approaches that identify concurrency issues without executing a program. The approaches include techniques based on typing, abstract interpretation, model checking, and others.

3.3.4.1 Type Systems

In the field of actor languages, the Erlang programming language has been subject to extensive studies. Dialyzer is a static analysis tool that uses *type inference in addition to type annotations* to analyze Erlang code [Sag05]. The static analysis uses information on control flow and data flow to identify problematic usage of Erlang built-in functions that can cause concurrency issues. Dialyzer also has support for detecting message order violations as well as memory inconsistencies [Sag10, CS10]. Christakis et al. [CS11b] extended Dialyzer to also detect communication deadlocks in Erlang using a technique based on *communication graphs*.

Another branch of work uses *type systems* to prevent concurrency issues. For actor languages, this includes, for instance, the work of Colaco et al. [CPS97]. Based on a type system for a primitive actor calculus, they can give compilation errors for some situations leading to orphan messages. However, static analysis cannot detect all possible orphan messages. Therefore, the approach relies on *dynamic type checks* to detect the remaining cases. Similar work was done for Erlang, where orphan messages are also detected based on a type system [DP02].

3.3.4.2 Abstract Interpretation Techniques

Abstract interpretation of programs is a static technique that describes computations in a universe of abstract objects. Executing the abstract representation of objects will give results about the actual computations [CC77].

Stievenart et al. [SNDMDR17] used *abstract interpretation* techniques to statically verify the absence of errors in actor-based programs and upper bounds of actor mailboxes. As mentioned before, the verification of mailbox bounds can avoid the presence of orphan messages. The proposed technique is based on different mailbox abstractions, which allows to preserve the order and multiplicity of the messages. Thus, this verification technique can be useful to avoid not only lack of progress issues but also message order violations.

Garoche et al. [GPT06] verified safety properties statically for an actor calculus by using *abstract interpretation.* Their work focuses on orphan messages and specific message order violations. Their technique is especially suited for detecting unreadable behavior, detecting unboundedness of resources, and determining whether linearity constraints hold.

3.3.4.3 Model Checking Techniques

Model checking is a static technique for automatically verifying correctness properties of programs given a specification of the property [CHVB18]. Model checking has been explored in the context of actor-based languages to verify properties like boundedness of actor mailboxes and incorrect interleavings of messages. Dam et al. [DF98] proposed an approach using static analysis to verify properties such as the boundedness of mailboxes. The verification of this property can avoid the presence of orphan messages in a program. Their technique applies *local model checking in combination with temporal logic and extensions to the μ-calculus* for basic Erlang systems. In the context of Erlang programs, Fredlund et al. [FGN$^+$03] proposed a model checker that verifies boundedness of their mailboxes and process spawning.

Huch [Huc99] focused on verifying the property of mutual exclusion or the absence of deadlocks and livelocks in Erlang programs, using model checking techniques and abstract interpretation. Similarly, [DKO13] also worked on Erlang and used static analysis and *infinite-state model checking.* Their goal is to check specific properties for programs that are expressed with annotations in the code. With this approach, they are able to verify, for instance, correct mutual exclusion semantics modeled with messages. However, their current approach cannot model arbitrary message order violations because the used analysis abstracts too coarsely from messages.

3.3.4.4 Static Techniques to Detect Races

Zheng et al. [ZBZ11] developed a static analysis for JavaScript relying on *call graphs and points-to sets*. The analysis detects bad message interleavings and message order violations by analyzing the JavaScript event-loop with respect to its reaction to incoming messages. WebRacer [PVSD12] is a tool that uses a *memory access model and a notion of happened-before relations* for detecting races at the level of the DOM tree nodes. The detected bugs correspond to bad message interleavings and message order violations in our taxonomy. EventRacer [RVS13] is another tool that aims at finding bad message interleavings or message order violations in JavaScript applications. In this case, the authors proposed a race detection algorithm based on *vector clocks*.

3.3.5 Testing

This section describes techniques that have been used on testing actor based-programs to identify concurrency bugs. Several techniques make use of static analysis techniques such as state model checking and variants of symbolic execution, e.g., concolic testing. Hence, we group the different approaches into three groups, the ones that test actor-based programs using symbolic execution, model checking, and other techniques.

3.3.5.1 Testing based on Symbolic Execution

Symbolic execution is a static technique to test whether certain predefined properties can be violated by a program [BCD^{+}18]. A key idea in symbolic execution is to explore programs taking as input *symbolic* values rather than concrete ones.

In the context of actors, much work has focused on *concolic testing*, which is a mix of concrete and symbolic execution. In particular, the authors used symbolic execution for generating data inputs that may lead to alternate behaviors, while they used concrete execution to guide the symbolic execution along a distinct execution path. Sen et al. [SA06] proposed a testing algorithm to detect communication deadlocks in a language closely related to actor semantics. They use a concolic testing approach that combines symbolic execution for input data generation with concrete execution to determine branch coverage. The key aspect of their technique is to minimize the number of execution paths that need to be explored while maintaining full coverage.

Albert et al. [AAGZ15] developed a test case generation framework which *avoids redundant computations when exploring the order of several tasks*. Their approach based on symbolic execution focus in implementing a test generation framework based on constraint logic programming. More recently, Albert et al. [AAGZ18] 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 message selection.

Recently, Li et al. proposed a target test generation technique that consists of building a message flow graph to later apply a *backward symbolic execution* in an actor system [LHA18]. Their approach explores the state space using symbolic execution based on heuristics that consider paths where only interact with a small number of actors. In their evaluation, their tool could identify message order violation bugs in Akka programs. Other bugs were related to orphan messages.

3.3.5.2 Testing based on Model Checking

In the field of testing, researchers have also applied static techniques like model checking.

Concuerror [CGS13b] is a systematic testing tool for Erlang that can detect abnormal process termination as well as blocked processes, which might indicate a communication deadlock. To identify these issues, Concuerror *records process interleavings* for test executions and implements a *stateless search strategy* to explore all interleavings.

Basset [LKMA10] is an automated testing tool based on Java PathFinder, a *state model checker*, that can discover bad message interleavings in Scala and ActorFoundry programs.

Tasharofi et al. [TKL$^+$12] improved Basset with a *partial order reduction technique to reduce schedules* to be explored, which improves the performance of Basset. Their key insight is to exploit the transitivity of message send dependencies to prune the search space for relevant execution schedules. For the Scala-Akka programs there is another testing tool called Bita, which can also detect message order violations [TPLJ13]. Their proposal is based on a technique called *schedule coverage*, which analyzes the order of the receive events of an actor.

3.3.5.3 Techniques for Test Case Generation

Claessen et al. [CPS$^+$09] use a test case generation approach based on *QuickCheck*[1] *in combination with a custom user-level scheduler* to identify race conditions. The focus is specifically on bad message interleavings and process termination issues. To make their approach intuitive for developers, they visualize problematic traces. Hughes et al. [HB11] use the same approach and apply it to a key component of the Mnesia database for Erlang. They demonstrate that the system is able to find race conditions at the message level that can occur when interacting with the shared memory primitives used by Mnesia.

Hong et al. [HPK14] proposed a JavaScript testing framework called WAVE for the same classes of issues mentioned by Petrov et al. [PVSD12] and Raychev et al. [RVS13]. The framework generates test cases based on *operation sequences.* In case of a concurrency bug, they can observe different results for the generated test cases.

[1]QuickCheck is a testing tool for Erlang available at `http://www.quviq.com/products/erlang-quickcheck/`.

The Setac framework [TGMJ11] for the Scala Actors framework enables testing for race conditions on actor messages, specifically message order violations. A test case defines *constraints on schedules and assertions* to be verified, while the framework identifies and executes all relevant schedules on the granularity of message processing.

Recently, Chang et al. [CDG+19] proposed NodeAV, a tool for detecting bad message interleavings in Node.js applications. Their approach first instruments the source code and then *executes test cases on the instrumented version* to collect the execution trace. Later they build a happened-before graph with the partial order among events of the execution trace. Finally, they infer atomic event pairs based on happened-before graphs and use predesigned bad message interleavings patterns to detect the incorrect interleavings.

3.3.6 Discussion based on our Taxonomy of Concurrency Bugs

This section summarizes the main techniques for each field of study we have seen in this chapter, and we discuss their advantages and disadvantages. Table 3.1, Table 3.2 and Table 3.3 show the techniques we surveyed of the fields static analysis, testing and debugging, respectively. A 'p' indicates that a bug category is addressed only partially. Typically, the approaches are limited by, for instance, a too coarse abstraction or a description language not expressive enough to capture all bugs in a category.

From Table 3.1 we observe that existing static analysis tools address mostly message order violations and communication deadlocks. On the one hand, approaches based on abstract interpretation have been applied to identify, for example, safety properties for actor messages. On the other hand, model checking and symbolic execution techniques have been explored to identify communication deadlocks. Overall, static techniques require programmers to know in advance exactly the type of bug is looking for or which specific property is violated in a program.

From Table 3.2 we observe that testing tools have addressed mostly message protocol violations. Although there are approaches that automatically generate test cases [BMP18], other approaches such as [CDG+19] require to have test cases available to use a testing tool.

Table 3.3 shows our categorization of bugs that we think are addressed in the debugging tools we surveyed from literature.

3.4 Conclusion

In this chapter, we have studied techniques of static analysis, testing, debuggers, and visualization that assist developers in finding concurrency bugs in actor-based programs. Much work on identifying concurrency bugs is done in the fields of static analysis and testing. However, since these techniques work on approximations of programs, it is difficult to use them when the bug is not known in advance. In this dissertation, we

	Communi. Deadlock	Behav. Deadlock	Live-Lock	Message Or. Violation	Bad Msg. Inter.	Mem. Incon.
Static Analysis						
type inference and communication graphs [CS11b]	X					
type inference [CS10]				X		X
type system [CPS97]	p					
type system [DP02]	p					
local model checking and temporal logic [DF98]	p					
model checking [FGN$^+$03]	p					
abstract interpretation [SNDMDR17]	p			p		
model checking [Huc99]	p			p		p
infinite state model checking [DKO13]	p			p		p
abstract interpretation [GPT06]	p			p		
call graphs and points to sets [ZBZ11]				p	p	
memory access model and happened-before relationship [PVSD12]				X	X	
race detection algorithm based on vector clocks [RVS13]				X		

Table 3.1: Overview of the bug categories addressed in the literature of static analysis techniques. A 'p' indicates that the bug has been addressed only partially. A 'X' indicates that the technique can address the bugs in that category.

	Communi. Deadlock	Behav. Deadlock	Live-Lock	Message Or. Violation	Bad Msg. Inter.	Mem. Incon.
Testing Tools						
concolic testing [SA06]	X					
test case generation with custom scheduler [CPS$^+$09]					X	
stateless search strategy [CGS13b]	X					
state model checker [LKMA10]					X	
coverage-guided schedule generation [TPLJ13]					X	
specify constraints on schedules and assertions [TGMJ11]				p	p	
reduce schedules with partial order reduction [TKL$^+$12]				p	X	
test case generation with randomizing scheduler [HB11]				p		X
test cases based on operation sequences [HPK14]				X	X	
test instrumented source code and build happened-before graphs [CDG$^+$19]					X	
test generation based on symbolic execution [AAGZ15, AAGZ18]	p			p	p	
targeted test generation based on backwards symbolic execution [LHA18]	X			p	p	

Table 3.2: Overview of the bug categories addressed in the literature of testing techniques. A 'p' indicates that the bug has been addressed only partially. A 'X' indicates that the technique can address the bugs in that category.

	Communi. Deadlock	Behav. Deadlock	Live-Lock	Message Or. Violation	Bad Msg. Inter.	Mem. Incon.
Online Debuggers						
REME-D [GBNDM14]	N/A	X	X	p	p	
Chrome DevTools [Not17]	N/A	p	p	p	p	
Erlang debugger [AB20]	p	p	p	p	p	p
ScalaIDE debugger [fED]	p	p	p	X	X	
Offline Debuggers						
Causeway [SCM09]	N/A	p	p	X	X	
Actoverse [SW17]	p	p	p	X	X	
IDeA [MOM18]	p	p	p	X	X	
McFly [VBMM18]	N/A	p	p	p	p	

Table 3.3: Overview of the bug categories that we consider that are addressed by debuggers of state of the art. A 'p' indicates that the bug has been addressed only partially. A 'X' indicates that the technique can address the bugs in that category. A 'N/A' indicates that is not applicable for those debuggers for languages which cannot suffer from communication deadlocks.

focus on online debugging techniques which allows interactive exploration of actor-based programs to find the root cause of concurrency bugs.

From the related work we studied in Section 3.3.1 and Section 3.3.2, there is a need for debuggers that *combine strategies such as visualizing the causality of messages with message-oriented breakpoints and rich stepping.* We argue that a debugger that does not only allow us to inspect the program state within a turn but also combines message-oriented features with sequential ones will help to identify the root cause of concurrency bugs as many times erroneous steps in sequential code and interactions between actors lead to application failures. So far, most of the visualization techniques focus on showing actor state and messages, but some of them do not show this information based on a happened-before relationship amongst messages. Besides, other visualization approaches are often not combined with breakpoints or stepping operations and thus developers cannot inspect actor state at certain points in the program execution. In Chapter 5 we propose interactive debugging techniques based on message-oriented breakpoints and stepping, trace-based visualizations, and an asynchronous stack trace to help developers identifying concurrency bugs present in actor-based programs.

Moreover, we have not found a debugging approach that enables interactive debugging of programs while minimizing the effects of the probe-effect. As mentioned by McDowell et al. [MH89] offline debuggers use deterministic replay to avoid the effects of the probe-effect, but they are only effective if the bug manifested in the execution that got recorded. Unfortunately, due to non-determinism, many cycles may be required to record the bug. In Chapter 8 we propose a new technique to allow developers to explore not one but all possible non-deterministic execution paths of an actor-based program interactively while avoiding the probe-effect.

Before explaining the details of the debugging techniques we propose in this dissertation, we will introduce in the next chapter SOMns, the programming language we will use in our research.

Chapter 4

SOMNs: a Concurrent Actor-based Language

The work described in this dissertation employs the SOMNs programming language as the research vehicle in which the novel debugging techniques will be implemented. This chapter describes the main characteristics of SOMNs, which are required to understand the code snippets and technical contributions presented in the following chapters.

4.1 The SOMNs Programming Language

The SOMNs programming language is an implementation of the Newspeak programming language [Bra09] building on the Simple Object Machine class libraries[1] (SOM). SOMNs has been designed as a research language in the context of concurrency models with good performance. SOMNs implements shared-memory models such as threads and software transactional memory as well as message-passing models such as communicating sequential processes and actors. In this work, we use SOMNs concurrency implementation for actors, i.e., its implementation of the Communicating Event-Loops (CEL) actor model (see Section 2.1.2), as a foundation for implementing advanced debugging techniques for actor-based programs.

We first detail the main language concepts including its object-oriented and concurrent programming features. Subsequently, we describe the implementation strategy on which the SOMNs language is built. In particular, we explain the implementation of asynchronous message passing in the SOMNs interpreter as it will be crucial to understand how we extended the SOMNs interpreter to add our novel debugging support for concurrency. We will illustrate SOMNs's features by means of code snippets from three sample applications: an instant messenger application (i.e., a chat application), a Pythagoras calculator application and an application that identifies prime numbers. The full description and implementation of those examples can be found in Appendix C.

[1]`http://som-st.github.io/`

The source code of all sample programs can be also found online in a fork of SOMns [2]. Appendix B includes the SOMns cheat sheet for fast reference with a summary of the language constructs which was also employed in the user study we conducted to validate our debugging support (see Chapter 7).

4.2 Object-oriented Programming in SOMns

SOMns is an object-oriented programming language based on the Smalltalk tradition [KG76]: everything is an object and computation happens in terms of objects sending messages. In this section, we explain the object-oriented programming model by means of the prime numbers application and instant messenger application. The full implementation is included in Appendix C.1 and Appendix C.2, respectively.

4.2.1 Classes

SOMns follows the traditional principles of the class declaration of object-oriented programming in which a class defines instances of objects with identical behavior, i.e, "*all instances of a class respond to the same set of messages*" [Bra09].

A class declaration in SOMns consists of a constructor and a body. The constructor defines the structure of the class through instance variables (called *slots* in SOMns terminology). The body consists of nested classes and methods. A method provides an implementation of a message. Listing 4.1 shows an example of a SOMns class named `InstantMessenger`, which represents a messenger user in a chat application. The class declaration starts with an *access modifier* (i.e., `public, private, protected`), the *class keyword*, the *name of the class*, and a *list of parameters.* The colons in the constructor is denoted for a place where a parameter is expected. In this case, the constructor method selector is `new:total:` (Line 1). This means that the constructor takes two parameters: the `name` of the user and `size` as the number of users in the chat. The argument `name` is stored in the instance variable named `name` (Line 3). The argument `size` is used as input for the dictionary `buddyMap` (Line 2). The constructor initializes the variable `textMessage` to `nil` (Line 4).

When a message is sent, the SOMns runtime system looks up for a method that matches this message (selector [3]), starting at the class of the receiver object. The first method found in the inheritance chain is invoked in the receiver. In Listing 4.1, the `InstantMessenger` class defines two methods, `startChat` starting in Line 7, and `sendMessage` starting in Line 11. The `startChat` method expects only one argument, i.e., `remoteMessenger`. This method adds the remote messenger to the dictionary of known messengers for this user (i.e., `buddyMap`) and later the current messenger user sends a 'Hello' message to the remote messenger. The `sendMessage` method expects two

[2]`https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/`

[3]Only the selector is used since it encodes the number of arguments.

arguments, i.e., `receiverName` and `content`. This method creates a new `TextMessage` object using the `content` argument, and send it asynchronously to the remote messenger, which is searched in the dictionary of buddies by its name (`receiverName`).

```
public class InstantMessenger new: name  total: size = (
  | private buddyMap = Dictionary new: size.
    public name = name.
    private textMessage ::= nil.
  |)(

  public startChat: remoteMessenger = (
    ...
  )

  public sendMessage: receiverName contentMsg: content = (
    ...
  )
...
)
```

Listing 4.1: Code snippet of `InstantMessenger` class and two of its methods, `sendMessage` and `startChat`.

The SOMns language uses the concept of *mixins* to reuse and extend classes. "*A class is either the empty class `Top` or the application of a mixin to another class known as its superclass.*" [Bra09]. The class thus inherits all the properties of its superclass that are not explicitly specified to be different by its mixin. Listing 4.2 shows an example[4] of class inheritance using the mixin operator <:. In the example the class `JsonParseError` extends from `Exception` and inherits all the properties of `Exception` class, as well as the properties of `Value` class which are not in `Exception`. Class `JsonParseError` corresponds to an implementation of an exception due to an unexpected character position when parsing incoming strings into JSON tokens. Instances of classes that extend from `Value` class as `JsonParseError`, are called in SOMns *value objects* (see Section 4.2.2).

```
private class JsonParseError signalFor: str at: idx = Exception <: Value (
  | public string = str.
    public index  = idx.
  |
    signal
  )(
    public message = (
      ^ 'JSON parse error. Unexpected character at ' +
        index + ' in "' + string + '".'.
    )
)
```

Listing 4.2: Class example in SOMns using a mixin operator.

[4]This code snippet has been extracted from `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/core-lib/demos/KomposDemo.ns`

4.2.1.1 Main Class Definition

The entry point of any SOMns program is the main method defined in a class called the *main class* in SOMns. The main class is declared as a regular class but it also needs to declare the parameter name `usingPlatform` with the parameter value `platform`. The `platform` object makes the standard library accessible. This is needed because Newspeak is a capability based language [Bra09].

Listing 4.3 shows the main class definition corresponding to our example of the chat application. The object instance of the `InstantMessengerApplication` class imports the `actors` module, the `Array` class from `kernel` module, and the `Dictionary` class from `collections` module. We will explain in Section 4.3 the facilities used from `actors` module. Line 16 and Line 17 adds two user names in the `users` array, calling the message `at:put:` from the `Array` class. The main method can return an integer as error code or a promise (see Section 4.3.3) to indicate program completion.

```
 1 class InstantMessengerApplication usingPlatform: platform = Value (
 2   | private actors = platform actors.
 3     private Array = platform kernel Array.
 4     private Dictionary = platform collections Dictionary.
 5   |)(
 6   ...
 7 
 8   public main: args = (
 9     | completionPP1 completionPP2 users messenger1 messenger2 pResult1
     pResult2 |
10     completionPP1:: actors createPromisePair.
11     completionPP2:: actors createPromisePair.
12 
13     '[INSTANT MESSENGER APPLICATION] Starting' println.
14 
15     users:: Array new: 2.
16     users at: 1 put: 'Joe'.
17     users at: 2 put: 'Marie'.
18 
19     ...
20   )
21 )
```

Listing 4.3: Code snippet of the `InstantMessengerApplication` that shows a main method implementation in SOMns.

4.2.2 Objects

As mentioned by [Bra09], "*an object is an entity that can perform computation in response to a message*" and "*every object is an instance of some class*". Objects in SOMns can be of two types:

Mutable objects: can change after they are created.

Deeply immutable objects: known also as *value objects* in SOMns cannot change after they are created. The main properties for immutable objects are [Bra09]:

- All its slots are immutable and contain value objects.
- Its enclosing objects are all value objects.
- Its class inherits from class `Value`.

In Listing 4.4, Line 1 to Line 4 show the implementation of the `TextMessage` class. *Object instances* of the `TextMessage` class are deeply immutable because inherits from the `Value` class. Line 6 to Line 18 show the implementation of the `InstantMessenger` class, which does not inherit from the `Value` class and thus object instances of this class are not deeply immutable objects.

Mutable slots can be updated, while immutable slots cannot be updated. Line 9 shows the declaration of the *mutable slot* `textMessage` with the mutable operator ::=. In Line 14 the `textMessage` variable is updated using the assignment operator ::.

Lines 7 and 8 show the declaration of *immutable slots*, using the operator = [5]. Declaring an immutable slot means that the slot cannot be updated (i.e., with the :: operator). However, it is possible to store mutable objects into immutable slots. In our code snippet, the `buddyMap` slot is an immutable object storing mutable objects (i.e., objects instances of the `InstantMessenger` class).

```
 1  class TextMessage new: content sender: senderName = Value (
 2   | public content = content.
 3     public sender = senderName.
 4   |)()
 5
 6  public class InstantMessenger new: name  total: size = (
 7    | private buddyMap = Dictionary new: size.
 8      public name = name.
 9      private textMessage ::= nil.
10    |)(
11
12  public sendMessage: receiverName contentMsg: content = (
13    | receiverActor pReceive |
14    textMessage:: TextMessage new: content sender: name.
15    ...
16  )
17  ...
18  )
```

Listing 4.4: Code snippet of `InstantMessengerApplication` program that shows the implementation of `TextMessage` class and a code snippet of the `InstantMessenger` class.

[5] In SOMns the equals operator = can be used with two different purposes, to declare immutable slots and to compare values.

4.2.3 Synchronous Messages

As Smalltalk, objects in SOMns communicate through messages. Messages make functionality of an object available. As mentioned before, when a message is sent, a matching method is looked up and invoked. A message send consists of a *receiver* followed by a *selector* and a *list of arguments*. Messages can be classified in three types according their structure:

unary messages: consists of a receiver and a selector, e.g., `25 sqrt`.

binary messages: consists of a receiver, a selector and an argument, e.g., `3 + 10`.

keyword messages: consists of a receiver and an argument for each keyword, e.g., `2.0 pow: 3.0`.

Messages are sent according to the following precedence rules:

1. Unary messages are always sent first, then binary messages and finally keyword messages.
2. Messages in parentheses are sent prior to any kind of messages.
3. Messages of the same kind are evaluated from left to right.

For example, in the expression `value:: 5+4*2` the variable `value` is equal to 18 instead of 13, because binary messages + and * have the same precedence and the evaluation happens from left to right. On the other hand, the expression `value:: 5+(4*2)` results in 13 because the use of parentheses makes the * binary message have the highest priority.

SOMns features reserved words that can be used as the receiver in message sends, e.g., `self`. As in Smalltalk, `self` is the receiver of the message and the method lookup starts in the class of the receiver. Listing 4.5 shows an example of a `self` send in Line 11 when the message `addMessenger` sent to the receiver object, in this case an instance of `InstantMessenger`.

So far, we have explained messages declare as *ordinary sends*, i.e., `object messageSelectorAndArguments`, and *self sends*, i.e., `self messageSelectorAndArguments`. But developers can also declare *implicit receiver sends*, i.e., `messageSelectorAndArguments`.

The receiver of an implicit message send can be `self` or an enclosing object to `self` [Bra09]. For example, in Listing 4.5 in Line 14 shows the message `sendMessage` whose receiver is the current executing object instance of `InstantMessenger` class. Arguments `remoteName` and `msg` represent the name of the user and the message content respectively.

4.2.4 Block Closures

An important syntactic element in SOMns is block closures (blocks in short). A block is an anonymous function with a definition context. Blocks are lexical closures since

```
public class InstantMessenger new: name  total: size = (
  | private buddyMap = Dictionary new: size.
    public name = name.
    private textMessage ::= nil.
  |)(

  public startChat: remoteMessenger = (
    | pDiscover pName pSend msg pp array |

    (* returns a far reference of the remote messenger *)
    pDiscover:: self addMessenger: remoteMessenger.
    ...

    pSend:: sendMessage: remoteName contentMsg: msg.
    ...
 )
 ...
)
```

Listing 4.5: Code snippet of `InstantMessenger` class that shows a `self` send and an implicit receiver send.

they can refer to the variables of the surrounding environment. Blocks are essential in SOMNS because there is no syntax for control-flow constructs. They are just implemented as expressions that operate on blocks.

A block is declared between brackets, and it can take arguments which are denoted in the syntax `:name |`. Also, blocks can declare local variables. The general syntax for blocks in SOMNS is `[:p1 ...:pN | | variables | body]`.

Listing 4.6 shows the `to:do:` message that implements a loop and takes a block as second argument; this block takes one argument `i` and the body adds random numbers to an array.

```
numbers:: TransferArray new: 10.

1 to: 10 do:[:i |
  rand:: Random new: i + 73425.
  numbers at: i put: (1 + (rand next % 100)).
].
```

Listing 4.6: Code snippet of the `PrimeNumber` class that shows a block closure example.

4.3 Concurrent Programming in SOMNS

SOMNS is a concurrent actor-based language, in which concurrent entities are represented as Communicating Event-Loops (see section 2.1.2), as introduced in the E programming language [MTS05]. In this concurrency model, objects are owned by *vats*, i.e., actors. Objects can refer to objects within the same actor via *near references* while objects can refer to objects in different actors via *far references*. Communication via near references happens by synchronous messages (as explained in Section 4.2.3). Communication via far references is always through asynchronous message passing. In this section we explain the concurrent programming model of SOMns by means of the program examples of instant messenger application and the Pythagoras calculator application. The full implementation is included in Appendix C.2 and Appendix C.3, respectively.

4.3.1 Actor Creation

In order to create an actor, SOMns provides the message `createActorFromValue` which takes as argument a value object. The message returns a far reference to the behavior object of an actor, which is an instance of the given value. Listing 4.7 shows the creation of an actor in the main method of the chat application. Line 12 creates an actor instance of the `InstantMessenger` class, which is stored in variable `messenger1`. Message `new` is sent to the far reference of the behavior object of the newly created actor which initializes the instance of the `InstantMessenger` actor. In SOMns asynchronous messages are defined by an explicit *receiver expression*, the *asynchronous send token* `<-:` and a *message selector* and its *arguments* [Bra09]. In this case `new` takes as argument the username of the messenger (e.g., Joe) and a number representing the total number of users in the chat (e.g., 2), which is needed to initialize the dictionnary that keeps all messengers.

```
 1 public main: args = (
 2   | completionPP1 completionPP2 users messenger1 ... |
 3   completionPP1:: actors createPromisePair.
 4   completionPP2:: actors createPromisePair.
 5
 6   '[INSTANT MESSENGER APPLICATION] Starting' println.
 7
 8   users:: Array new: 2.
 9   users at: 1 put: 'Joe'.
10   users at: 2 put: 'Marie'.
11
12   messenger1:: (actors createActorFromValue: InstantMessenger) <-: new: 'Joe'
       total: 2.
13   ...
14  )
```

Listing 4.7: Code snippet of the `InstantMessenger` class that shows an actor creation example in SOMns.

4.3.2 Asynchronous Messages

We now explain message semantics amongst actors, i.e., asynchronous message. Asynchronous messages in SOMns return a *promise*, a placeholder object to store the result of the computation that happens in a later point in time [MTS05].

```
 1 public class InstantMessenger new: name  total: size = (
 2   | private buddyMap = Dictionary new: size.
 3     public name = name.
 4     private textMessage ::= nil.
 5   |)(
 6
 7   public sendMessage: receiverName contentMsg: content = (
 8     | receiverActor pReceive |
 9     textMessage:: TextMessage new: content sender: name.
10
11     receiverActor:: buddyMap at: receiverName.
12     pReceive:: receiverActor <-: receive: textMessage.
13     pReceive whenResolved:[: r|
14       ('Receive message '+ r) println.
15     ].
16
17     ^ pReceive
18   )
19  ...
20  )
```

Listing 4.8: Code snippet of the `InstantMessenger` class that shows an asynchronous message send in SOMns.

Listing 4.8 shows an asynchronous message send in the chat application. Line 12 shows the message `receive` sent asynchronously to `receiverActor`, which is another actor instance of the `InstantMessenger` class. The *receiver* of an asynchronous message is an object in the actor where the message is sent. The result of the computation of the message sent is the promise `pReceive`. In the next section we detail the semantics of promises in SOMns. Finally, the message `receive` takes as argument a `TextMessage` object, which contains the content of the message and the username of the message sender.

As explained before, the receiver of the asynchronous message is a far reference to another actor. Besides, the receiver expression can be a promise (see Section 4.3.3) and a near reference. In what follows, we explain how the message will be processed for each of the three receivers [Bra09]:

- If the receiver is a *far reference*, the message is sent to the actor associated with the object corresponding to the far reference. The message is enqueued in the mailbox of the actor for later processing considering the following rules:
  - If actor A1 sends a message m1 to actor A2, and subsequently sends message m2 to A2, then A2 will receive and process m1 before m2.

  - If actor A1 sends a message m1 to a far reference object associated with actor A2, and A1 subsequently passes that object to A3, then m1 will be received and processed by A2 before any message from A3 to the object.

- If the receiver is a *near reference*, the message is sent to the current actor, and it is processed following the same approach as described for the far reference.

- If the receiver is a *promise*, the message will be sent to the result to which the promise is resolved. The message will be processed following the same approach as described for the far reference.

We now detail the passing parameter semantics for asynchronous messages. By default, objects which are shared amongst actors as an argument in messages or as return types of asynchronous message (i.e., mutable objects) are *passed by reference.* Objects instances of the `Value` class (i.e., immutable objects) or instances of the `TransferObject` [6] class are *passed instead by (deep) copy.* The deep copy of transfer objects ends when reaching value objects, regular objects, or far references. References to value objects do not change, while references to regular objects are changed into far references. Far references are replaced with near references when the receiving actor is the owner of the far referenced object. Value objects in SOMns can be shared amongst actors. Some examples of value objects are numbers, booleans, and strings.

4.3.3 Promises

As mentioned before, an asynchronous message returns a promise. SOMns considers a promise to be in either one of the following four states: *unresolved, successful, erroneous, or chained.* When a promise is created, i.e., it has not a return value yet, is said to be *unresolved.* A promise is *successful* when it has been resolved with a value (that is not a promise). A promise is *erroneous* when it has been resolved with an error. This happens when processing the asynchronous message raised an exception (which is transmitted to the sender). Finally, a promise is *chained* when it has been resolved with another promise. When the original promise is resolved, all the dependent promises are immediately resolved with the proper returned value (see Section 4.3.3.1).

In SOMns, a block can be registered to a promise, which is asynchronously applied when the promise becomes resolved with a value or with an error. This can be done through the messages `whenResolved:` (i.e., to handle a successful promise), `onError:` (i.e., to handle an erroneous promise) or `whenResolved:onError:` (i.e., to handle a successful or an erroneous promise).

As a concrete example, consider the Line 13 in Listing 4.8. When the return value of message `receive` is computed, the promise `pReceive` is resolved and a `whenResolved` block applied, i.e., this result value (i.e., `#ok`) is made available to the sender actor of the

[6]Implementation of `TransferObject` class can be found in `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/core-lib/Kernel.ns`

message, then it is said that the promise is *resolved*. The resolution value of a promise can be a far reference, or a near reference if the object can be passed by copy between the actors [MTS05]. In this case, the callback gets as argument `r` the return value of the computation of the message `receive`, and the body of the block prints the value. The `whenResolved:` message registers a block only for the case when the promise is *successful*, and returns a new promise which will be resolved with the return value of the block given as argument. In this case if an error occurs an exception is thrown (or signaled) by the SOMNs interpreter.

We now illustrate how the developer can register handlers to a *successful* or an *erroneous* promise using the message `whenResolved:onError:`. The message takes two blocks as input: the `whenResolved:` block is applied for the case in which the promise resolution is successful and the second one, `onError:`, is applied if the promise resolution was erroneous. To this end we employ the code example shown in Listing 4.9. In Line 1, the message `division`[7] is sent to the `math` actor and returns the promise `resultDiv`. At this point the promise created is *unresolved*. If the promise is resolved with a value, Line 3, then its state will be *successful*, but if for example a division by zero error occurs, Line 5, the promise is resolved with the error and then its state will be *erroreous*.

```
1 resultDiv:: math <-: division: 27 and: 5.
2 resultDiv whenResolved:[:div |
3   ('Division result: '+ div) println.
4 ] onError:[:e |
5   ('DivisionZeroError: ' + e) println.
6 ].
```

Listing 4.9: Code snippet that shows how a promise can be resolved with a value or with an error.

The message `onError:` can be also sent directly to a promise, i.e., without adding handlers for the success case (`whenResolved:`). In that case the promise to which this message is sent will have only registered a block for the erroneous resolution.

4.3.3.1 Promise Chaining

Asynchronous messages can be sent to promises even if they have not yet been resolved with a value or an error. When an asynchronous message is sent to an *unresolved* promise, it is said that the new promise representing the message sent is *chained* to the receiver promise [MTS05]. The message will not be delivered until the receiver's promise is resolved.

[7]The `division` message is not used in the Pythagoras calculator application, it has been added only to be used as demonstrative example for listing Listing 4.9.

Listing 4.10 shows a promise chaining in the context of the Pythagoras calculator application (see Appendix C.3). In this example, the `sqrt` message is chained. Once the result of the `add` message is available, the `sqrt` message is sent to that result.

```
public computePerimeter = (
  | sideA sideB squareA squareB perimeterPP |
  ...

  squareSumP:: math <-: add: 1 and: 25.
  hypotenusePromise:: squareSumP <-: sqrt.

  ...
)
```

Listing 4.10: Code snippet of the `Calculator` class that shows the `hypotenusePromise` promise chained to the `squareSumP` promise using values 1 and 25 as example.

4.3.3.2 Explicit Promises

Promises are *implicitly* created when an asynchronous message is sent using the asynchronous operator `<-:` in SOMns. In SOMns, developers can also *explicitly* create promises for conditional synchronization amongst actors. Explicit promises are created by sending the `createPromisePair` message. This construct creates a pair object[8], storing the promise object and its *resolver*. A resolver is an object which understands messages to resolve a promise with a value or with an error.

As an example of explicit promise consider Listing 4.11, Line 3. For the case of explicit promises the promise resolution is declared explicitly by the developer, using the `resolve` message, an example can be seen in Line 12. In this case the promise is resolved with the value of `perimeter` variable, making the promise successful. It can happen that the promise is resolved with another promise and then is chained. To resolve the promise with an error the developer can use the message `error`. A developer can send the message `resolve` or `error` directly to the pair object, (as shown in Line 12). The pair object understands these messages as the `resolver` object, i.e., the pair object invokes the `resolve` or `error` messages that the `resolver` object implements.

4.3.3.3 Promise Group

SOMns allows creating a promise group representing a combined promise for a list of promises for which a block can be applied when all the promises are resolved. Promises can be grouped using the concatenation operator (`,`) which returns the combined promise

[8]A promise pair in SOMns is an instance of the `Pair` class implemented in `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/core-lib/Actors.ns` which contains two objects, a promise and a resolver.

```
public computePerimeter = (
  | sideA sideB squareA squareB perimeterPP |
  perimeterPP:: actors createPromisePair.
  ...
  perimeterPromise:: (math <-: trianglePerimeter: sideA b: sideB c: sideC).
  perimeterPromise whenResolved:[:perimeter |
    ('Student assignment: '+ studentId + ',
    Triangle sides: A = '+sideA+',
    B = '+sideB+ ',
    C = '+sideC + ',
    Perimeter: '+perimeter) println.
    perimeterPP resolve: perimeter.
  ].
  ...

  ^ perimeterPP promise
)

```

Listing 4.11: Code snippet of the `Calculator` class that shows explicit promises.

in a vector[9]. Sending the `whenResolved:` message to the promise group registers a block to be applied when all the promises in the group have been resolved.

In the example shown in Listing 4.12, we create a promise group for applying some computation when the return value of two `square` messages is received. More concretely, the receiver of the `whenResolved` message in Line 11, is a vector of two promises for the two `square` asynchronous message sends. The `whenResolved` message gets as argument the `squares` variable, which is a vector of resolution values corresponding to the promises `squareA` and `squareB`.

4.4 SOMNs: a Language Implemented on Top of Truffle

SOMNs is implemented as an AST-based interpreter that runs on top of the Java Virtual Machine (JVM) using Truffle [MM15]. This section describes the necessary details on Truffle and the implementation of SOMNs to follow our technical contributions later.

Truffle is a language and tool development framework that allows developers to write programming languages as interpreters that perform self-optimizations at runtime, which allows these languages to reach the performance of state of the art virtual machines. Truffle languages run on top of the GraalVM [10], a high-performance platform capable of running applications written in different languages such as Java, Scala, JavaScript, LLVM-based languages such as C and C++, Python, Ruby, and more.

[9]The implementation of the `Vector` class is available at `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/core-lib/Kernel.ns`

[10]`https://github.com/oracle/graal`

```
public computePerimeter = (
  | sideA sideB squareA squareB perimeterPP |
  perimeterPP:: actors createPromisePair.

  sideA:: 1 + (rand next % numberStudents).
  sideB:: 1 + (rand next % numberStudents).

  squareA:: math <-: square: sideA.
  squareB:: math <-: square: sideB.

  squareA, squareB whenResolved:[:squares |
    | squareSumP hypotenusePromise |
    squareSumP:: math <-: add: (squares at: 1) and: (squares at: 2).
    hypotenusePromise:: squareSumP <-: sqrt.
    ...
  ].

  ^ perimeterPP promise
)
```

Listing 4.12: Code snippet of the `Calculator` class that shows a promise group.

Figure 4.1 gives an overview of the GraalVM architecture with a guest language implementation. The green layer represents the *language application*, i.e., actor-based programs written in SOMns in our case. The pink layer represents an *AST interpreter implementation*, e.g., SOMns. Truffle interpreters are written in the Java language. The gray layers correspond to the already mentioned *GraalVM* platform. Graal is a just-in-time compiler, from Java bytecode to machine code, which generates optimized compiled code from interpreters using advanced techniques such as partial evaluation [WWS$^+$12, WWW$^+$13]. The white and blue layers show the JVM, which is the *host language*, i.e., the language where the guest language is written [WWS$^+$12]. The JVM provides different services that can be leveraged by the guest language, e.g., memory management, exception handling, and others. In this thesis, SOMns is the guest language, and Java is the host language.

We now briefly describe how guest language programs are executed in GraalVM. Figure 4.2 shows an example of an AST with five uninitialized nodes (U) representing the execution of a JavaScript program. The execution of a program boils down to the evaluation of these nodes. During the program's execution, uninitialized nodes are replaced with type-specific nodes corresponding to the types seen at runtime, i.e., in the figure, the types shown belong to the guest language, in that case, JavaScript. The action of replacing nodes during execution by a type-specific version is called *node specialization*[11] [dVSH$^+$18] (i.e., "node replacement allows the node to specialize on a subset of the semantics of a particular guest language operation" [WWW$^+$13]). In the example, three nodes are specialized to integer (I), and two nodes are specialized to generic (G). Generic

[11]This action was initially refer as *node rewriting* by [WWW$^+$13]

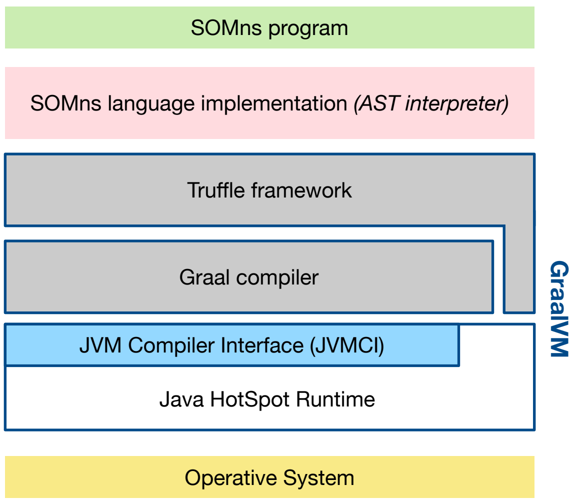


Figure 4.1: Overview of GraalVM architecture with SOMns as a guest language, inspired by [dVSH+18].

means that the specialization for that node in a specific execution was not valid, and a generic node is used instead to handle all possible cases. Node specialization allows updating profiling information, e.g., dynamic type information. Later, when the AST nodes representation is stable, GraalVM uses partial evaluation, to produce highly specialized machine code [WWW+13, WWH+17, dVSH+18].

When a specialization fails, i.e., the type of the node is no longer valid, then compiled code is reverted to the AST interpreter through *dynamic deoptimization*. Figure 4.3 extracted from [dVSH+18] shows the scenario in which a computation overflows the integer range, and the compilation to obtain machine code cannot handle the case for computation on a double. This way, the integer nodes rewrite themselves to double nodes. Later the AST is recompiled, producing double-specialized code.

4.4.1 Building Tools with Truffle Instrumentation API

Guest languages implemented in Truffle can use the Truffle Instrumentation API[12] to create language-agnostic tools, e.g., profilers, code coverage tools, debuggers, etc. on top of the GraalVM. This API allows developers to introspect the program's behavior and inject behavior by inserting nodes.

[12]The documentation about the classes of this API can be found in `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/package-summary.html`

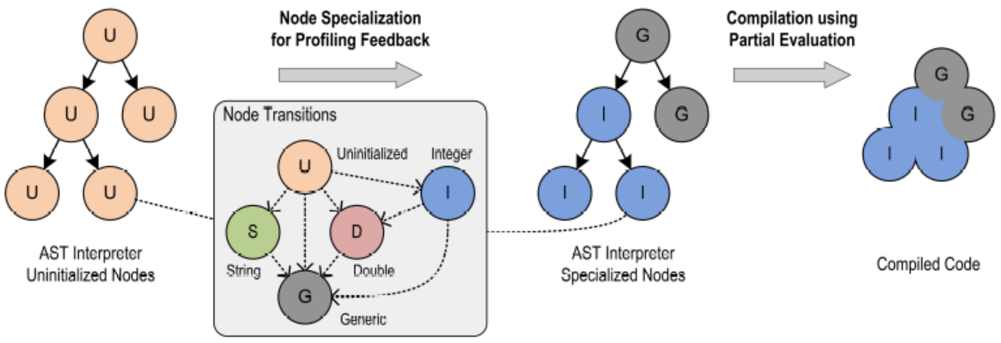

Node Specialization for Profiling Feedback
Compilation using Partial Evaluation
Node Transitions
Uninitialized
Integer
String
Double
Generic
AST Interpreter Uninitialized Nodes
AST Interpreter Specialized Nodes
Compiled Code

Figure 4.2: Node specialization for profiling feedback and partial evaluation for getting specialized machine code (i.e., speculate and optimize), extracted from [dVSH+18].

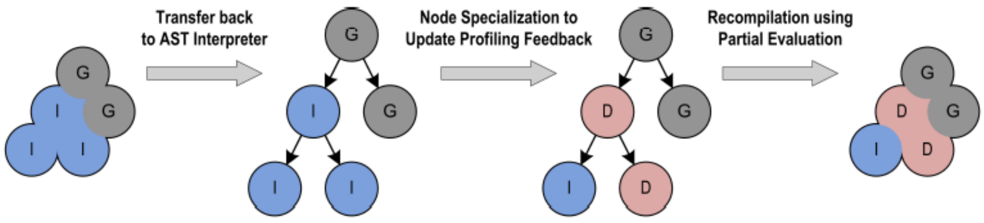

Transfer back to AST Interpreter
Node Specialization to Update Profiling Feedback
Recompilation using Partial Evaluation

Figure 4.3: Deoptimization back to the AST interpreter handles speculation failures (i.e., transfer back to the interpreter and reoptimize), extracted from [dVSH+18].

4.4.1.1 Code Coverage Example

We now illustrate how to use the Truffle Instrumentation API to implement a language-agnostic tool to perform code coverage [13].

Listing 4.13 shows the code to create the code coverage tool. The first step is to subclass the `TruffleInstrument` class to create an *instrumentation agent*. Instrumentation agents can monitor VM-level runtime events, e.g., source code related events, allocation events, thread creation events and application events [Gra21]. Besides, extending `TruffleInstrument` class, the instrument needs to override the `onCreate` method, in order to register itself in the GraalVM execution environment (Line 7).

[13]Example extracted from Truffle Instrumentation API code coverage example at `https://github.com/oracle/graal/blob/master/truffle/src/com.oracle.truffle.api.instrumentation.test/src/com/oracle/truffle/api/instrumentation/test/examples/CoverageExample.java` A second example that makes use of the Coveralls.io service is available at `https://github.com/MetaConc/CoverallsTruffle`

In this tool example, the goal is to instrument all expressions with a wrapper node to notify when that expression is executed. The nodes to be executed implement the `ExecutionEventNode` class, which instrument events during execution, i.e., implement methods to intercept runtime execution events such as `onEnter, onReturnValue, onReturnExceptional`. Instances of `ExecutionEventNode` class (Line 26), *wrap* AST nodes of interest, which are defined by the source filter in Line 8. In the example, the filter allows obtaining all nodes that are considered expressions, which is denoted by the `ExpressionTag` tag, in Line 9.

In order to attach our code coverage instrumentation agent to the GraalVM, we call the method `attachExecutionEventFactory` passing as argument a source filter and a `ExecutionEventNodeFactory` instance (in our case, an instance of `CoverageExampleEventFactory` in Line 12). The argument `ExecutionEventNodeFactory` provides instrumentation for the AST nodes to be executed by the agent every time a runtime event specified by the source filter is executed. As we can observe in Line 36, the source section of the current evaluated expression will be saved in the Set `coverage` after the node corresponding to the expression is evaluated. The flag `visited` in Line 31 keeps track of all code locations that have already been visited.

Finally, it is important to mention that each instrumentation node is bound to a code location, which can be accessed through the `EventContext` object corresponding to the node.

4.4.1.2 Instrumentation for Debugging

In our work, we will use the Truffle Instrumentation API to build debugging support for actor-based programs. Furthermore, we will use the features provided by the Truffle Debug API.

The Truffle Debug API [14] contains language-agnostic abstractions for adding traditional debugging functionalities to the guest language implementation. For example, this API can be used to add *line and conditional breakpoints*, *standard stepping* (step over, step into, step out), *stack information* for a suspension, and more. The classes provided by this API simplify the debugger implementation for a Truffle language [15]. Two relevant classes in the Truffle Debug API are:

- `DebuggerSession` implements a debugging session. An instance of this class requests the suspension of guest language execution threads when it receives a notification that the thread has reached an AST location, which occurs, for example, when setting a breakpoint or a step operation.

[14]`https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/debug/package-summary.html`

[15]A simple example for a debugger tool implemented with the instrumentation framework can be found in the GraalVM repository (`https://github.com/oracle/graal/blob/master/truffle/src/com.oracle.truffle.api.instrumentation.test/src/com/oracle/truffle/api/instrumentation/test/examples/DebuggerExample.java`).

```java
@Registration(id = CoverageExample.ID, services = Object.class)
public final class CoverageExample extends TruffleInstrument {
  public static final String ID = "test-coverage";
  private final Set<SourceSection> coverage = new HashSet<>();

  @Override
  protected void onCreate(final Env env) {
    SourceSectionFilter.Builder builder = SourceSectionFilter.newBuilder();
    SourceSectionFilter filter = builder.tagIs(ExpressionTag.class).build();
    Instrumenter instrumenter = env.getInstrumenter();
    instrumenter.attachExecutionEventFactory(filter,
                        new CoverageExampleEventFactory(env));
  }

  private class CoverageExampleEventFactory
                  implements ExecutionEventNodeFactory {

    private final Env env;

    CoverageExampleEventFactory(final Env env) {
      this.env = env;
    }

    public ExecutionEventNode create(final EventContext ec) {
      final PrintStream out = new PrintStream(env.out());
      return new ExecutionEventNode() {
        @CompilationFinal private boolean visited;

        @Override
        public void onReturnValue(VirtualFrame vFrame, Object result) {
          if (!visited) {
            CompilerDirectives.transferToInterpreterAndInvalidate();
            visited = true;
            SourceSection src = ec.getInstrumentedSourceSection();
            out.print(src.getCharIndex() + " ");
            coverage.add(src);
          }
        }
      };
    }
  }
}
```

Listing 4.13: Code example of an expression coverage instrument, from `CoverageExample` class.

- `SuspendedEvent` handles the state of a suspended guest language execution thread. An instance of this class is passed by the session via synchronous callback on the execution thread and has access to the stack frames corresponding to the suspension.

Despite the existing debugging functionalities, the Truffle Debug API does not support language-agnostic abstractions for concurrency.

As we saw in the coverage program example, nodes implemented with the class `ExecutionEventNode` listen to instrumentation events in a specific location of the guest language. However, building new debugging support (e.g., new breakpoints) requires more advanced instrumentation of the guest language. For instance, when defining an instrumentable node which can create their own specializations.

Listing 4.14[16] shows how developers can define instrumentable nodes implementing the interface `InstrumentableNode`. Truffle automatically generates *wrapper nodes* once the guest language is compiled, i.e., with the `@GenerateWrapper` annotation.

Here we describe how the instrumentation of a node works. Truffle instruments a node by inserting two additional nodes [dVSH+18]. First, a *wrapper node* (W), which is automatically generated, proxies for its child and report events before the wrapper child executes, e.g., events such as `onEnter()`, `onReturnValue()`, `onReturnExceptional()`. The second node is a *probe node* (P) that dispatches event reports to clients. Figure 4.4 shows a simple representation of the instrumentation of an AST node N with a wrapper node (proxy) and a probe node (dispatcher).

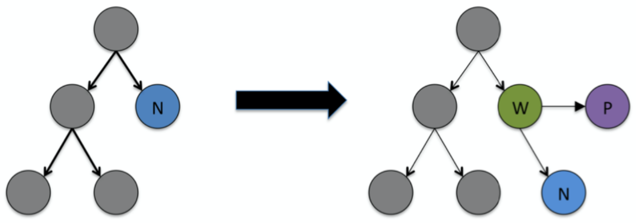


Figure 4.4: Truffle instrumentation of an AST node with a wrapper (W) and probe (P) node, extracted from [dVSH+18].

Each node that is instrumented keeps the information about its source location (i.e., source attribution) and tags (i.e., syntactic tags). These two elements are relevant for debugging [dVSH+18].

Source attribution is the source section of an AST node corresponding to the exact location of the node in the code, which is needed, for example, for defining and triggering breakpoints. In the example of Listing 4.14, methods in Line 31, Line 37, Line 44 shows how to enable source sections for instrumentable nodes. Usually, the language parser

[16]This code example was extracted from `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/InstrumentableNode.html`

```java
@GenerateWrapper
abstract static class StatementNode extends SimpleNode
                implements InstrumentableNode {

  private static final int NO_SOURCE = -1;
  private int sourceCharIndex = NO_SOURCE;
  private int sourceLength;

  @Override
  public final Object execute(VirtualFrame frame) {
    executeVoid(frame);
    return null;
  }

  public abstract void executeVoid(VirtualFrame frame);

  @Override
  public final InstrumentableNode.WrapperNode createWrapper(ProbeNode probe) {
    // ASTNodeWrapper is generated by @GenerateWrapper
    return StatementNodeWrapper.create(this, probe);
  }

  public boolean hasTag(Class<? extends Tag> tag) {
    if (tag == StandardTags.StatementTag.class) {
          return true;
    }
    return false;
  }

  // invoked by the parser to set the source
  void setSourceSection(int charIndex, int length) {
    assert sourceCharIndex == NO_SOURCE : "source should only be set once";
    this.sourceCharIndex = charIndex;
    this.sourceLength = length;
  }

  public final boolean isInstrumentable() {
    // all AST nodes with source are instrumentable
    return sourceCharIndex != NO_SOURCE;
  }

  @Override
  @CompilerDirectives.TruffleBoundary
  public final SourceSection getSourceSection() {
    if (sourceCharIndex == NO_SOURCE) {
      // AST node without source
      return null;
    }
    RootNode rootNode = getRootNode();
      if (rootNode == null) {
        // not yet adopted yet
        return null;
      }
      Source source = rootNode.getSourceSection().getSource();
      return source.createSection(sourceCharIndex, sourceLength);
    }
}
```

Listing 4.14: Implementation example of an instrumentable node to tag nodes as statements and support for source section.

handles the source section[17]. Truffle source attribution allows configuring a debugger to step through expressions instead of the traditional line breakpoint.

Declaring *syntactic tags* is useful for a debugger to identify on which AST nodes the program's execution might halt. In other words, tags are used by guest languages to categorize nodes which will be later required by the debugger. In Listing 4.14, Line 23, we can see how the language implementor overrides the method `hasTag` to decide if the instrumented node is tagged with a specific tag. In the code example, the node is classified as a statement. This way, a debugger can recognize all statement nodes. Truffle Instrumentation API provides a set of tags by default in the `StandardTags` class. Some examples are:

- `RootTag:` marks program locations as the root of a function, method, or closure.
- `StatementTag:` marks program locations that represent a statement of a language.
- `CallTag:` marks program locations that represent a call to other guest language functions, methods, or closures.
- `ExpressionTag:` marks program locations as to be considered expressions of the languages.

We leverage the Truffle Instrumentation API and the Truffle Debug API to build a debugger for SOMns programs. Our novel debugging techniques for actor programs are built on asynchronous message passing nodes in the SOMns implementation. In the next section, we describe what is needed from SOMns message passing implementation to understand the implementation of our debugging features (see Chapter 6).

4.4.2 Implementation of Asynchronous Message Passing in SOMns

Asynchronous messages in the SOMns interpreter are instances of the class `EventualMessage`. SOMns distinguishes between two types of eventual messages: *direct* and *promise* messages. A *direct message* is an asynchronous message sent to a far reference and will be executed on the actor owning the receiver (i.e., the far reference). A *promise message* is an asynchronous message sent to a promise. SOMns further distinguishes between promise send and promise callback message. A *promise send message*, is a message sent to a promise while it is unresolved, and the message will be delivered once the promise gets resolved. A *promise callback message* is a message to be sent after a promise is resolved, e.g., the message created to apply a `whenResolved` block, which is a callback for the resolved promise.

[17]https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/source/SourceSection.html

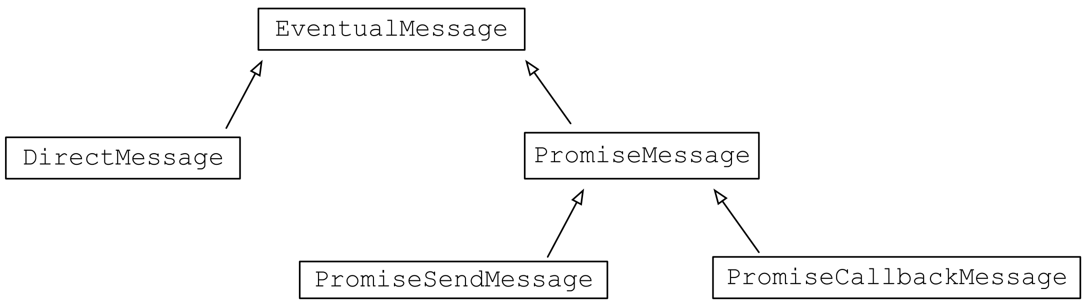


Figure 4.5: Messages classes in SOMns.

Using Truffle specializations[18] the SOMns interpreter creates eventual messages depending on the receiver of the message and the result. Table 4.1 provides an overview of the different operations that create eventual messages in SOMns. These operations are implemented in the `EventualSendNode` class, i.e., the AST node for the asynchronous send operator. Besides the *operation name*, the table specifies the *receiver* (i.e., a message can be sent to a near reference, a far reference, or to a promise) as well as its *result*. Asynchronous messages return promises. Operations may also differ depending on how the promise *resolution* happens:

- normal resolution: the promise is resolved with a value or an exception.
- null resolution: the promise has been optimized out because the program does not use the return value.
- chained resolution: a promise is resolved with another promise, which is not itself.

Also, the table shows the *message type* created by each operation, i.e., a direct message or a promise message.

Operation name	Receiver	Result	Type resolution	Message type
toFarRefWithResultPromise	far reference	promise	normal resolution	direct msg
toPromiseWithResultPromise	promise	promise	chained resolution	promise msg
toNearRefWithResultPromise	near reference	promise	normal resolution	direct msg
toFarRefWithoutResultPromise	far reference	nil	null resolution	direct msg
toPromiseWithoutResultPromise	promise	nil	null resolution	promise msg

Table 4.1: Operations that create eventual messages in SOMns.

Figure 4.6 and Figure 4.7 show the message invocation protocol for an asynchronous message at the sender side and receiver side, respectively. We first describe Figure 4.6 showing the steps taken by the SOMns interpreter as a result of an asynchronous

[18]`https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html`

message send: `pReceive:: receiverActor <-: receive: textMessage` from Listing 4.8, Line 12. This message is sent from an instant messenger actor to another and corresponds to the `toFarRefWithResultPromise` in Table 4.1. Evaluating the message `receiverActor <-: receive: textMessage` results in the execution of the node that represents the asynchronous operator in the abstract syntax tree (AST) of the program, namely a `SendNode` instance. Figure 4.6 uses straight lines for direct messages and dotted lines for promise messages. We now detail the sequence of steps at sender side:

- Step 1: A first path (annotated with number 1) is denoted for the case of a message sent directly to a far reference. The `sendDirectMessage` method creates the new message and invokes the `send` method on the `Actor` class.
- Step 2: The `send` method is called for the `receiverActor`.
- Step 3: Then the far reference corresponding to the receiver actor appends the message in the actor's mailbox.
- Step 4: The `execute` method of the `Actor` class is invoked, which makes the thread of a pool execute the messages of the current actor.
- Step 5: The current actor processes all messages contained in its mailbox. This is enabled by the invocation of the `processCurrentMsg` method by the runnable task corresponding to the thread.

The sequence of steps at sender side of a *message sent to a promise* are similar to the ones described above, except for step 1. After executing the `SendNode`, instead of invoking the `sendDirectMessage` method, the interpreter calls the `sendPromiseMessage` method (step 1a). Here a `PromiseSendMessage` message is created and passed as argument to the register method of the `RegisterPromiseNode` class (step 1b). This method also expects as argument a promise object, a resolver object and the arguments corresponding to the message. Step 1c corresponds to scheduling the promise message to the current executing actor. Only when the promise is resolved the message will be delivered (step 2a). Hence the promise message is added in the mailbox (step 3) and waits to be processed in the actor pool (step 4).

Figure 4.7 shows the sequence of steps for the message invocation protocol at the receiver side.

- Step 1: Processing the current message in the actor pool triggers the `execute` method of the `EventualMessage` class, which is the abstract class for all message types supported in SOMns[19] (see figure 4.5).
- Step 2: Executing the message means to call the node that corresponds to the invoked message. In the diagram the node is `ReceivedRootNode`.

[19]Its implementation is available at `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/src/som/interpreter/actors/EventualMessage.java`

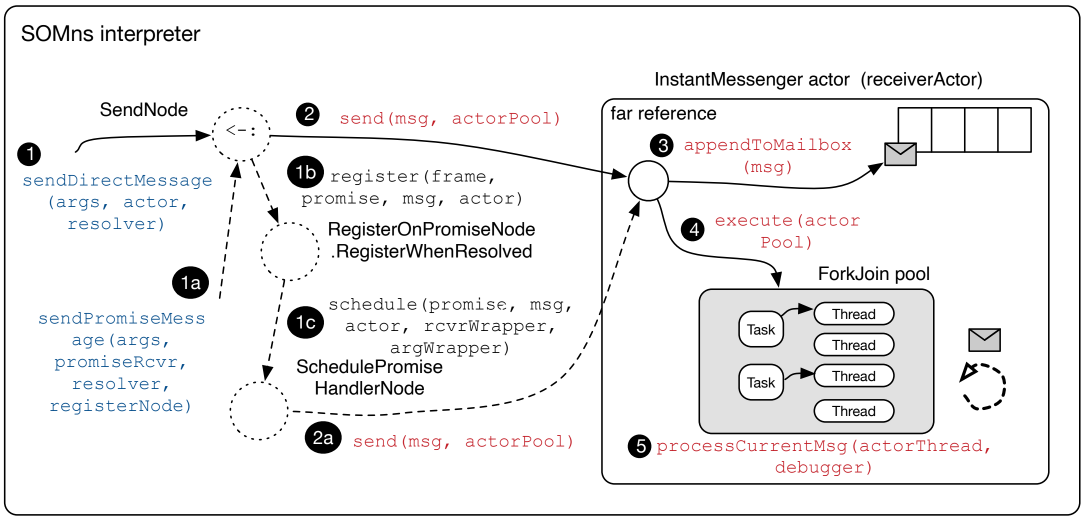

SOMns interpreter
SendNode
<-:
1
sendDirectMessage (args, actor, resolver)
1a
sendPromiseMessage (args, promiseRcvr, resolver, registerNode)
2
send(msg, actorPool)
1b
register(frame, promise, msg, actor)
RegisterOnPromiseNode .RegisterWhenResolved
1c
schedule(promise, msg, actor, rcvrWrapper, argWrapper)
SchedulePromise HandlerNode
2a
send(msg, actorPool)
InstantMessenger actor (receiverActor)
far reference
3
appendToMailbox (msg)
4
execute(actor Pool)
ForkJoin pool
Task
Task
Thread
Thread
Thread
Thread
5
processCurrentMsg(actorThread, debugger)

Figure 4.6: Message invocation protocol at sender side. The blue font stands for message operations, the black font stands for AST node operations, the red font stands for actor operations. Dotted circles represent AST nodes, and normal circles represent objects.

- Step 3: The `executeBody` method in the `ReceivedMessage` node evaluates the arguments of the message in the method body.
- Step 4: Once the body of the message is executed, the result value is used to resolved the promise `pReceive` through the node `AbstractPromiseResolutionNode`, which is the abstract class for resolving promises.
- Step 5: In this case, the promise follows a *normal resolution*, i.e., the promise is resolved with a value that is not another promise (i.e., `#ok` returned by the `receive` method). This resolution is executed in the node `ResolveNode`.
- Step 6: The method `resolveAndTriggerListenersUnsynced` in the `SPromise` class handles the resolution of the promise with a proper value. If there are callbacks they will be scheduled, and if the promise was chained to other promises, the chained promises will be resolved.
- Step 7: The resolver object, i.e., an instance of `SResolver` class, executes the `scheduleAllWhenResolvedUnsync` method to schedule all `whenResolved` callbacks for the promise. In our running example, there is one callback registered, that prints the result once the promise has been resolved.
- Step 8: The `scheduleCallbacksOnResolution` method adjusts the target of the eventual message and calls the `send` message for the target actor. In this case, a promise callback message is created corresponding to the callback registered to the promise.

- Step 9: The `SPromise` instance invokes the `send` method, which is executed by the far reference of the receiver actor.
- Step 10: Then, the promise callback message is appended in the mailbox.
- Step 11: The `execute` method is called immediately. This way the callback message is processed once the promise `pReceive` has been resolved.

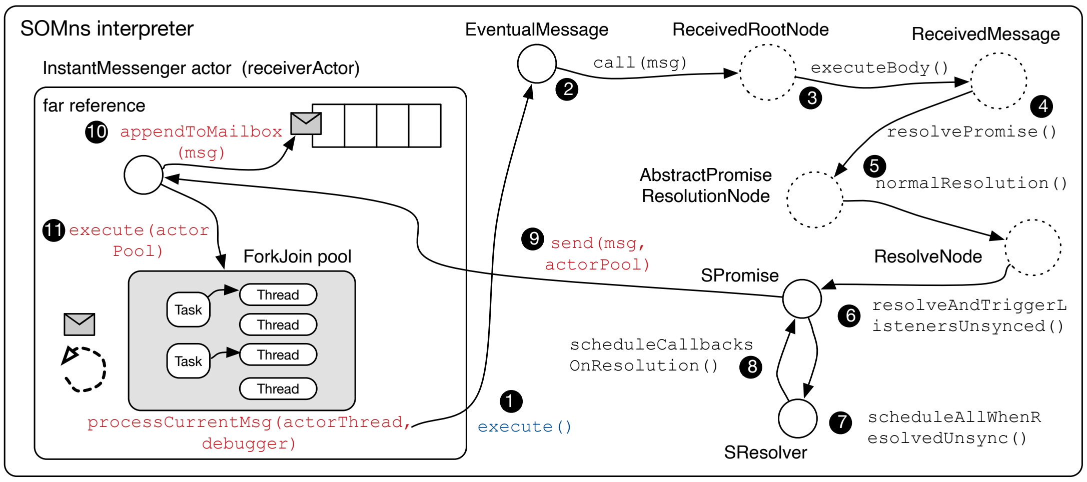


Figure 4.7: Message invocation protocol at receiver side. The blue font stands for message operations, the black font stands for AST node operations, the red font stands for actor operations. Dotted circles represent AST nodes, and normal circles represent objects.

As mentioned in step 5, the described example follows a *normal promise resolution.* Recall that two more cases that SOMNS implementation considers: null and chained resolutions.

The case of the *null resolution* is possible when the result of a message is never and can never be used, i.e., it's not the receiver of a message, and it is not assigned to a variable. For this case, the resolver object is not allocated. The following code snippet illustrates this case: `receiverActor <-: receive: textMessage.` Here, the program does not use the promise resulting from the asynchronous message send, but if a callback is registered to that asynchronous send, the promise resolver object is not null because the result is used by the program in the callback. For example `(receiverActor <-: receive: textMessage) whenResolved:[: r| ...].` The implementation of the null resolution case is handled by the `ReceivedRootNode` node, and the interpreter knows that a result is not used based on whether the result value is ever consumed by another expression in the program. The case of null resolution is shown for the operations `toFarRefWithoutResultPromise` and `toPromiseWithoutResultPromise` of Table 4.1.

The following code snippet shows the case of *chained resolutions*, i.e., in which a promise is resolved with another promise. More concretely, the promise for the second eventual message sent is chained to the promise stored in `pReceive`:

```
pReceive:: receiverActor <-: receive: textMessage.
pPrint:: pReceive <-: println.
```

The implementation of this code example is handled by the `AbstractPromiseResolutionNode` node, where the current promise is added to the list of chained promises owned by the receiver promise. When the first promise is resolved the chaining promises are resolved. This task is handled by the promise object (i.e., an instance of `SPromise` class). Chained promises are created by the operation `toPromiseWithResultPromise` shown in Table 4.1.

4.5 Conclusion

In this chapter, we have described SOMns, an implementation of the Newspeak programming language on the SOM class libraries. SOMns is a dynamically typed class-based programming language featuring the Communicating Event-Loops concurrency actor model, which avoids data races and deadlocks by design. We have illustrated through three running examples how an actor-based program can be implemented in SOMns. We have also described the main features of the Truffle framework and GraalVM, the runtime on which the SOMns language is built, and the key parts of the SOMns language implementation needed to understand our technical contributions.

In the following chapters, we describe novel techniques for debugging actor-based programs and how we implemented them as an extension to the SOMns interpreter.

Chapter 5

Online Debugging Techniques for Actor-based Programs

As we described in Chapter 3, nowadays, only three debuggers support breakpoint on messages to inspect the actor's state (i.e., REME-D [GBNDM14], Actoverse [SW17] and IDeA [MOM18]). To the best of our knowledge, there is no debugger that combines message-oriented stepping with sequential stepping so that developers can seamlessly apply step-by-step execution at asynchronous message level as well as method or function calls. Besides, few debuggers provide dedicated visualizations for easing the comprehension of message causality relations.

This chapter presents novel techniques to interactively debug programs in online debugging mode. First, we introduce a breakpoint catalog and a catalog of stepping operations on the level of messages, promises, and turns. Moreover, we design techniques to visualize information related to the actor's state, the causality of messages, and an asynchronous stack trace. For explaining the new debugging features, we will use a simple SOMns program that identifies prime numbers (Appendix C.1). For illustrating visualizations related to actor state inspection, message causality, and asynchronous stack trace, we will use code snippets from the Pythagoras calculator program (Appendix C.3).

Afterwards, we introduce a proof-of-concept debugger called Apgar, which is implemented in the IntelliJ IDE that features the mentioned advanced debugging techniques for SOMns. Later, we describe the extensions to the Kómpos protocol needed to implement the proposed visualizations, and we conclude the chapter by comparing our approach to related work.

5.1 Design of Online Debugging Techniques for Actor-based Programs

Inspired by the work of [GBNDM14], which introduced developing message breakpoints in the context of actor programming, our goal is to investigate a breakpoint and stepping operations catalog to allow developers to explore a target program written in the Communicating Event-Loops model (CEL) interactively. Central to both breakpoints and stepping operations is the identification of interesting halting locations for pausing the execution of an actor for further inspection.

Since turns run till completion in the CEL model, operation interleavings happen at the message level. As such, we consider the point in time right before and right after a message is processed as relevant for breakpoints. Figure 5.1 shows the points of interest involved in an asynchronous message send in CEL, in particular for the case of sending an asynchronous message to a far reference that returns a promise. We consider here the case where the promise has `whenResolved` listeners attached. The diagram represents the message send `pIsPrime:: math <-: isPrime: n.` shown in Line 48, Listing 5.1 corresponding to the prime number program.

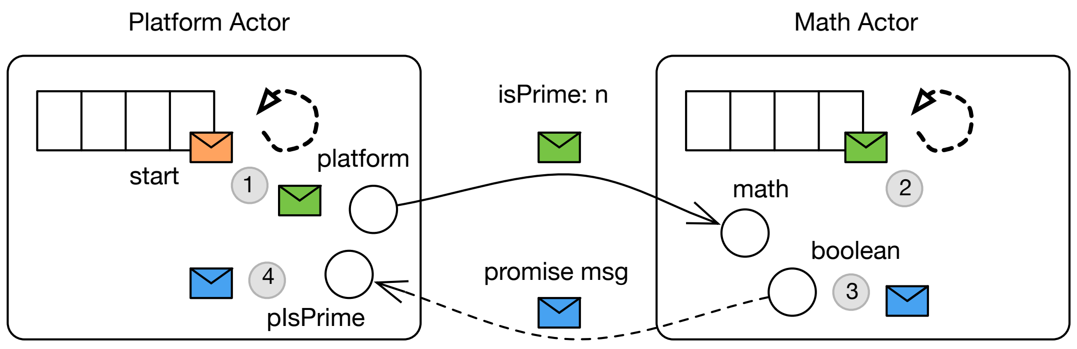


Figure 5.1: Points of interest for debugging actor-based programs.

In the diagram, the `Platform` actor is executing the message `start` -*orange color*- in which the behavior object of the actor `platform` sends the message `isPrime: n` to the behavior object of actor `Math` (point 1) -*green color*-. The *orange* message corresponds to the `start` message, which as mentioned in Section 4.2.1.1 is the first message sent by the Platform actor, which invokes the `main` method of the program. The `isPrime: n` message is appended to the receiver actor's mailbox, i.e., `Math` actor, and a promise object `pIsPrime` is immediately returned to the `Platform` actor. When the new message reaches the head of the mailbox of the `Math` actor (point 2), then the actor proceeds to execute it. After the value for the promise is computed (point 3), which is `true` or `false` in our code example, a new message -*blue color*- with the result value is created and sent back to the actor hosting the sender object, i.e., the `Platform` actor. The *blue* message carrying the result of the computation is not sent explicitly in the code, but internally in the language interpreter (for implementation details see section 4.4.2, figure 4.7). This new message carrying the result value is appended to the sender's mailbox, and when the message reaches the head of the mailbox of the `Platform` actor (point 4) the promise resolution listeners are finally triggered. At this point, a promise can be resolved or ruined, but for debugging, there is no explicit difference. We will refer to point 4 as “resolved" in this chapter. In this case, the listener attached to the returned promise is declared in the `whenResolved` block, which starts in Line 49.

```
 1 class PrimeNumber usingPlatform: platform = Value (
 2 | private actors = platform actors.
 3   private TransferArray     = platform kernel TransferArray.
 4   private harness = (platform system loadModule:
 5   'core-lib/Benchmarks/Harness.ns' nextTo: self) usingPlatform: platform.
 6   private Random = harness Random.
 7 |)(
 8
 9  public class Math = ()(
10
11    public isPrime: number = (
12      | limit |
13        number > 1
14          ifTrue:[
15            limit:: number/2.
16            2 to: limit do: [: counter|
17              number%counter = 0
18              ifTrue:[
19                ^ false
20              ]
21            ].
22
23            ^ true
24           ]
25           ifFalse:[
26             ('ERROR in Math: Prime numbers should be greater than 1,
27             number received: ' + number) println.
28             ^ false
29           ].
30    )
31   )
32
33   public main: args = (
34     | math numbers completionPP rand |
35
36     completionPP:: actors createPromisePair.
37     numbers:: TransferArray new: 10.
38
39     1 to: 10 do:[:i |
40       rand:: Random new: i + 73425.
41       numbers at: i put: (1 + (rand next % 100)).
42     ].
43
44     math:: (actors createActorFromValue: Math) <-: new.
45
46     numbers do:[:n |
47       | pIsPrime pWR |
48       pIsPrime:: math <-: isPrime: n.
49       pWR:: pIsPrime whenResolved:[:isPrime |
50        isPrime ifTrue:[ ('Number '+ n + ' is prime') println.].
51        isPrime ifFalse:[ ('Number '+ n + ' is not prime') println.].
52
53        n = (numbers at: (numbers size))
```

```
        ifTrue:[
          completionPP resolve: true.
        ]
       ].

       pWR <-: println.
     ].

     completionPP promise <-: println.

     ^ completionPP promise
  )
)
```

Listing 5.1: Implementation of a prime number program in SOMns.

5.1.1 Message Breakpoints

As we mentioned before, one of the main features of online debuggers is *breakpoints* (see Section 3.2). In this work, we propose an extension of the breakpoint catalog for debugging actor-based programs introduced by [GBNDM14]. The breakpoints correspond to the 4 points of interest shown in Figure 5.1. Table 5.1 summarizes the catalog in which we classified the message-oriented breakpoints into five dimensions:

Halt represents the point of interest defined in figure 5.1, specifically the point where a breakpoint will pause the program's execution.

Actor activation determines the place where the breakpoint halts regarding the actor. We distinguish between two sides, at the *sender* or the *receiver* actor.

Message execution defines whether the breakpoint's goal is to halt the execution *before* or *after* the message is processed.

Category groups the breakpoints regarding its halting scope, which could be related to a passive entity (i.e., *message* or a *promise*) or a dynamic scope (i.e., a *turn*). We do not have breakpoints with the *turn* category because the halting does not include changing of turn's execution[1]. This is different for some stepping operations (see Section 5.1.2), that only one command can lead the debugger to pause the program in a different turn.

Definition distinguishes the place where the breakpoints are defined in the underlying language implementation. In particular, in our work we develop breakpoints applied to AST nodes and we distinguish primitives and operators that return promises, on which we can define promise breakpoints:

[1]The developer needs to define a second breakpoint in a different turn and execute the *resume* command if he or she wants to pause in a different turn than the one in which the program is currently paused, i.e., the debugger requires two debugging operations a breakpoint and resuming execution.

- asynchronous send operator (e.g., in SOMNS `<-:`)
- promise primitives, i.e., primitives that return a promise (e.g., in SOMNS `createPromisePair`, `whenResolved:`, `whenResolved: onError:`, `onError:`)

Breakpoints	Halt	Actor activation		Message execution		Definition	Category
		Sender	Receiver	Before	After	AST node	
message sender	1	X		X		`<-:`	message
message promise resolution	4	X			X	`<-:` promise primitives	promise
message receiver	2		X	X		`<-:`	message
asynchronous before	2		X	X		method declaration	message
message promise resolver	3		X		X	`<-:` promise primitives	promise
asynchronous after	3		X		X	method declaration	message

Table 5.1: Catalog of breakpoints for actor-based programs.

Now we describe and illustrate the semantics of each of the breakpoints in Table 5.1 on the running example shown in Listing 5.1.

A message sender breakpoint triggers before the message is sent. For example, if one sets a message sender breakpoint in the asynchronous operator node located in Line 48, the debugger will pause the program's execution in the same line *before* sending the message to the `Math` actor.

A message receiver breakpoint triggers before the actor processes the received message. For example, if one sets a message receiver breakpoint in the asynchronous operator located in Line 48, the debugger will pause the program's execution in the AST node starting in Line 13, before the `Math` actor processes the `isPrime` message.

A message promise resolver breakpoint triggers before the computed value of the message is used to resolve the promise. For example, if one sets a promise resolver breakpoint in the asynchronous operator node located in Line 48, the debugger will pause the program's execution in Line 29 corresponding to the last line of the sequence node, which represents the body of the method. The message has been executed, and the result of the computation is known, but the promise has not been resolved yet with the result value.

A message promise resolution breakpoint triggers after the promise is resolved but before executing the callback. For example, if one sets a promise resolution breakpoint in the asynchronous operator node located in Line 48, the debugger will pause the program's execution in the node starting in Line 50, before the statement in this line is executed. This node corresponds to the body of the callback.

The breakpoints named as *asynchronous before* and *asynchronous after* exhibit similar semantics as message receiver and promise resolver, but the actor will suspend execution when it is about to process *any* asynchronous message invoking the selected method.

An asynchronous before breakpoint triggers before the message is processed. For example, if one sets an asynchronous before breakpoint in the declaration of method `isPrime` located in Line 11, the debugger will pause the program's execution in Line 13, before the method is executed.

An asynchronous after breakpoint triggers after the message is processed. For example, if one sets an asynchronous after breakpoint in the declaration of method `isPrime` located in Line 11, the debugger will pause the program's execution in Line 29, after the execution of the body of the method.

Table 5.2 summarizes the positions mentioned for the breakpoints examples on our proof of concept debugger for SOMns programs. Specifically, the AST location where each breakpoint is defined, the AST location where the halt should occur and the actor that is paused. In Table 5.2 the starting line number for asynchronous before breakpoint and for the asynchronous after breakpoint is 13 instead of 11 because we send to the interpreter the coordinates of the node corresponding to the method's body. In Section 6.1.1 we explain the implementation strategy for these breakpoints in our proof of concept debugger.

Breakpoint	AST location	Halt AST location	Actor paused
message sender	ln: 47, cn: 24, cl:3	ln: 47, cn: 24, cl: 3	Platform
message receiver	ln: 47, cn: 24, cl:3	ln: 13, cn: 11, cl: 414	Math
message promise resolver	ln: 47, cn: 24, cl:3	ln: 28, cn: 12, cl: 414	Math
message promise resolution	ln: 47, cn: 24, cl:3	ln: 49, cn: 9, cl: 233	Platform
asynchronous before	ln: 13, cn: 11, cl: 414	ln: 13, cn: 11, cl: 414	Math
asynchronous after	ln: 13, cn: 11, cl: 414	ln: 28, cn: 11, cl: 414	Math

Table 5.2: Breakpoint examples from Listing 5.1. AST locations are defined by the line number (ln), a column number (cn) and the character length (cl).

5.1.1.1 Message breakpoints and promises

Breakpoints on messages are as important as having breakpoints on promises since, as we mentioned before (Section 2.1.2), promises are used in the CEL model to get the return value of asynchronous messages and for synchronization amongst actors.

As we mentioned in Chapter 4, there are promises that can be created implicitly (e.g., created by the interpreter with the asynchronous message operator) and explicitly (i.e., promises resolved explicitly by the developer in the program). In our proposal, we cover both cases, using the concrete constructs that SOMNS features for them but the concepts and ideas can be applied to other CEL languages, e.g., AmbientTalk.

Here we describe three different cases where breakpoints in the category of promises can be defined.

5.1.1.1.1 Breakpoints on implicit promises In our approach, it is possible to define promise resolver and promise resolution breakpoints that work on implicit promises. Recall from Section 4.3.3 that SOMNS features constructs such as `whenResolved`, `onError` and `whenResolvedOnError` that return implicit promises. We now show how they work by means of our running example.

For example, if one sets a promise resolver breakpoint in the `whenResolved` message located in Line 49, the debugger will pause the program's execution in Line 56, after the statement was executed. If you set a promise resolution breakpoint in the `whenResolved` message located in Line 49, the debugger will pause the program's execution in Line 77 of the `Thing` class[2], in which the `println` method is implemented. This is because in Line 59 of Listing 5.1 a `println` message is sent to the promise returned by the `whenResolved`. Table 5.3 summarizes the mentioned examples.

Breakpoints	AST location	Halt AST location	Actor paused
message promise resolver	ln: 48, cn: 23, cl: 275	ln: 55, cn: 10, cl: 233	Platform
message promise resolution	ln: 48, cn: 23, cl: 275	ln: 77, cn: 25, cl: 36 (`Thing`) class	Platform

Table 5.3: Promise breakpoint examples defined on the `whenResolved` construct.

Similar to the `whenResolved`, promises breakpoints are applicable in the constructs `onError` and `whenResolvedOnError`.

5.1.1.1.2 Breakpoints on explicit promises We now turn our attention to breakpoints on explicit promises. Recall from Section 4.3.3.2, SOMNS offers the `createPromisePair` construct to create an *explicit* promise. The construct `createPromisePair` returns a pair (promise, resolver), and the developer can use it to explicitly resolve a promise by sending the `resolve:` message to the resolver. We now illustrate the semantics of explicit promise breakpoints on our running example. For example, if one sets a promise resolver breakpoint in the message `createPromisePair` located in Line 36, the debugger will pause the program's execution in Line 93 of the `Pair`[3] class, before executing the

[2]The Thing class is part of the SOMNS runtime system, can be found in `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/core-lib/Kernel.ns`

[3]The Pair class is part of the SOMNS runtime system, can be found in `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/core-lib/Actors.ns`

`resolve:` primitive. If you set a promise resolution breakpoint in Line 36, the debugger will pause the program's execution in Line 77 of the class `Thing` of SOMns, in which the `println` method is implemented. This is because in Line 62 a `println` message is sent to the promise of the promise pair. If there is no message sent to that promise, the breakpoint does not trigger. Table 5.4 summarizes the mentioned examples.

Breakpoints	AST location	Halt AST location	Actor paused
message promise resolver	ln: 35, cn: 29, cl: 17	ln: 93, cn: 16, cl: 14 `Pair` class	Platform
message promise resolution	ln: 35, cn: 29, cl: 17	ln: 77, cn: 25, cl: 36 `Thing` class	Platform

Table 5.4: Promise breakpoint examples defined on the `createPromisePair` construct.

5.1.1.1.3 Promise breakpoints on chained resolution In the case of setting a promise resolution breakpoint in a *chained resolution*, i.e., on a promise that is chained with another promise, the resolution breakpoint will trigger before executing the message that was sent to that promise. For example, in appendix C.3, a promise resolution breakpoint in Line 61 will paused the program before executing `sqrt` message of the class Integer[4]. This is the expected behavior because message `sqrt` is sent to the `squareSumP` promise in Line 62, which is the result message of the message sent of Line 61.

5.1.2 Message Stepping

Stepping is a crucial feature in an online debugger that allows developers to follow a program's execution between various points of interest. In sequential programs, stepping operations typically allow stepping through the program line by line. In CEL programs, stepping also needs to allow developers to step through the program's execution concurrently, i.e., let them follow the execution between the points of interest involved in asynchronous message passing. Thus, stepping operations can be applied at each of the points shown in Figure 5.1, and it will allow the developer to step to the next point of interest. Our concrete proof of concept debugger will enable the combination of both message and sequential stepping based on AST node wrapping (as we will explain in Chapter 6).

Table 5.5 shows 6 stepping operations we propose for actor-based programs. We classify the stepping operations in three dimensions:

Halt before step distinguishes the place (s) where the actor is halted due to, for example, a previous stepping or a breakpoint. In this place, a stepping operation can be defined. Numbers refer to the point of interest shown in figure 5.1. The last three operations can be defined from anywhere inside the current executing

[4]The Integer class is part of the SOMns runtime system, can be found in `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/core-lib/Kernel.ns`

turn. These nodes are represented in the table by * symbol. Our implementation includes points 2 and 3 in figure 5.1, which corresponds to the beginning and end of the turn, respectively. Besides, the program could be halted in a turn node [5] due to a classical *sequential stepping operation*, i.e., step into, step over, step out.

Halt after step shows the halting location for each actor related to the message being stepped. The actor that halts in the *next node* means that the actor will be suspended due to a *step over*. This means that it cannot process the next message, but the actor is not blocked, it still can receive messages. The actor that *resumes* will continue its execution, i.e., it will not be suspended.

Category groups the breakpoints regarding its halting scope. The halting is related to a passive entity (i.e., *message* or a *promise*) or a dynamic scope (i.e., a *turn*).

Stepping operation	Halt before step	Halt after step		Category
		Sender actor	Receiver actor	
step to message receiver	1	next node	2	message
step to promise resolver	1, 2	next node	3	promise
step to promise resolution	1, 2, 3	next node	4	promise
return from turn to promise resolution	2, 3, *	4	resume	turn
step to next turn	2, 3, *	resume	2	turn
step to end turn	2, 3, *	resume	3	turn

Table 5.5: Catalog of stepping operations for actor-based programs.

Here we describe and illustrate the semantics of each of the stepping operations shown in Table 5.5 on the running example of Listing 5.1.

A step to message receiver allows the developer to halt the program execution before a message is executed by the receiver actor (i.e., is equivalent to define a message receiver breakpoint). E.g., if the program of Listing 5.1 is suspended in the asynchronous operator node located in Line 48 due to a message sender breakpoint, and step to message receiver is triggered, the `Platform` actor will be suspended in the AST node starting in Line 49 and the `Math` actor will be suspended in the AST node corresponding to the method body in Line 13.

A step to promise resolver allows the developer to halt the program execution before a promise is resolved (i.e., is equivalent to define a promise resolver breakpoint). E.g., if the program is suspended in the asynchronous operator node located in

[5]We refer a turn node to any expression node that is declared inside a specific turn, i.e., in the method body corresponding that turn.

Line 48 due to a message sender breakpoint, and step to promise resolver is triggered, the `Math` actor will be suspended at the end of the method body Line 29, after the value for resolving the promise is computed. The `Platform` actor will be suspended in the AST node starting in Line 49.

A step to promise resolution allows the developer to halt the program execution after a promise is resolved but before executing its callbacks (i.e., is equivalent to define a promise resolution breakpoint). E.g., if the program is suspended in the asynchronous operator node located in Line 48 due to a message sender breakpoint, and step to promise resolution is triggered, the `Platform` actor will be suspended in the AST node starting in Line 49.

A return from turn to promise resolution allows the developer to halt before the execution of the first statement of all handlers registered on a promise that is resolved by the current turn (i.e., is equivalent to define a promise resolution breakpoint for the current turn). E.g., if the program is suspended at the beginning of the method body in Line 13 due to a message receiver breakpoint, and return to promise resolution is triggered, the `Math` actor resumes execution but the `Platform` actor will be suspended in the AST node of the body of the callback starting in Line 50.

A step to next turn allows the developer to halt the suspended actor before the next message of its mailbox is processed (i.e., is equivalent to define a message receiver breakpoint in the next message of the mailbox). E.g., if the program is suspended at the beginning of the method body in Line 13 due to a message receiver breakpoint, and step to next turn is triggered, the `Math` actor will be suspended in the next turn, in the same AST node of Line 13. For this code example, the next turn happens to be the same message but with a different value for the `number` variable.

A step to end turn allows the developer to halt the actor before returning from the method (i.e., it is equivalent to define a promise resolver breakpoint in the current turn). Here the code example refers to stepping to the end of the turn when the program is suspended at the beginning of the turn. More cases can be considered, such as stepping from inside synchronous calls and from inside blocks. E.g., if the program is suspended at the beginning of the method body in Line 13 due to a message receiver breakpoint, and step to end turn is triggered, the `Math` actor will be suspended at the end of the method body in Line 29, after the value to resolve the promise is computed.

Table 5.6 summarizes the positions mentioned for the stepping examples. Specifically, the AST node source coordinates where each stepping is defined, i.e., the line number, the column number, and the character length, where the halt should occur and the actor that is paused.

Step to	Halt before step	Halt after step	
		Sender	Receiver
message receiver	ln: 47, cn: 24, cl: 3	Platform ln: 48, cn: 8, cl: 290	Math ln: 13, cn: 11, cl: 414
promise resolver	ln: 47, cn: 24, cl: 3	Platform ln: 48, cn: 8, cl: 290	Math ln: 28, cn: 12, cl: 414
promise resolution	ln: 47, cn: 24, cl: 3	Platform ln: 48, cn: 8, cl: 290	Math resume
return from turn to promise resolution	ln: 13, cn: 11, cl: 414	Platform ln: 49, cn: 9, cl: 233	Math resume
next turn	ln: 13, cn: 11, cl: 414	Platform resume	Math ln: 13, cn: 11, cl: 414
end turn	ln: 13, cn: 11, cl: 414	Platform resume	Math ln: 28, cn: 12, cl: 414

Table 5.6: Stepping operation examples from Listing 5.1.

We argue that halting the program's execution on the level of messages is a useful debugging technique for actor-based programs because developers can inspect the program's state at relevant halting locations for messages, promises, and turns. E.g., developers can inspect the actor state at the sender or the receiver actor and before or after a message is processed. Even more, we can inspect computed values before and after resolving a promise. These possibilities can be helpful when debugging lack of progress issues such as *livelocks* and *behavioral deadlocks* because the developer will see whether the message arguments have the expected values, e.g., with a breakpoint at the sender side. Also, a breakpoint after a promise is resolved, allows us to inspect the actor state at that point which can be helpful as well for *message protocol violations*. The combination of message stepping with the possibility of seeing the sequential operations that the actor executes inside of a turn (and even more, navigating to other turns) gives developers a better toolbox to identify the root cause of a bug.

5.1.3 Trace-based Visualizations

The debugging features explained on breakpoints and stepping operations are essential to enable interactive debugging of a program in an online debugger. As already explained in Chapter 3, postmortem techniques may still be beneficial to understand the root cause of a concurrency bug since the distance between the failure and the fault is large in concurrent programs. As such, we propose a *hybrid approach* in which we complement the catalogs of online techniques with visualization based on trace information when programs are suspended on a breakpoint or due to a stepping operation.

In this section, we propose two visualization designs for showing the actor state and the message causality information of an actor-based program. One of the main contributions in our proposal is that the debugging information to build these visualizations are obtained from trace data generated by the debugger backend and send to the debugger frontend using the Kómpos debugging protocol (see Section 5.3), to be shown in an online mode.

5.1.3.1 Actor State Inspection

Our approach for showing the actor state follows very close to the one of the REME-D debugger, i.e., exhibit a *list of actors* created in the running program and the *mailbox* for the paused actors. For each asynchronous message REME-D showed two properties, its selector and the id of the sender actor. We have extended the mailbox visualization with four more message properties than in REME-D:

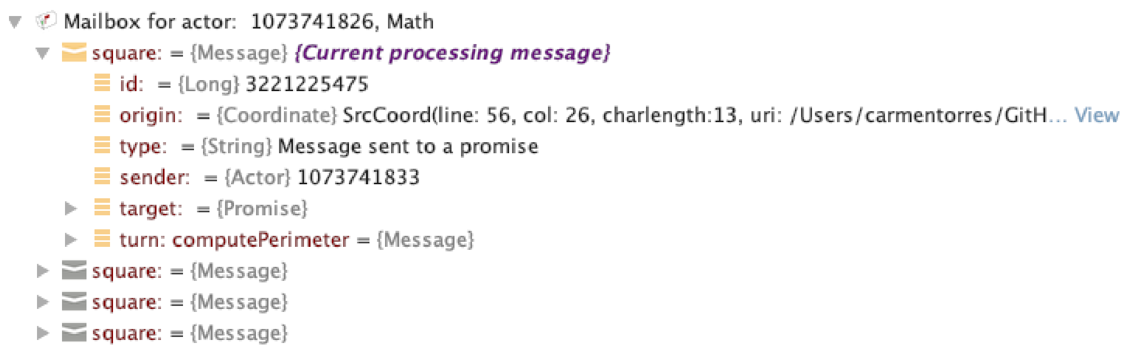
origin the source location where that message was sent, i.e., the source coordinate of the corresponding node for the asynchronous operator.

type the message type (i.e., if a message is sent to a far reference or to a promise).

target the message target (i.e., the receiver of the message).

turn the message turn, i.e., the message object that corresponds to the turn where the current processing message was sent. This property allows developers to later inspect all messages in a turn view, which visualization we explain in Section 5.1.3.2.

Figure 5.2 shows our proposed visualization of a mailbox that contains a list of messages for a paused actor. With this visualization, the developer can also see causality information in the actor state. Moreover, in the next section, we show the design of another visualization for showing not only causal information for the paused actor but for all of the actors of the program.



```
▼ Mailbox for actor:  1073741826, Math
  ▼ square: = {Message} {Current processing message}
      id:  = {Long} 3221225475
      origin:  = {Coordinate} SrcCoord(line: 56, col: 26, charlength:13, uri: /Users/carmentorres/GitH... View
      type:  = {String} Message sent to a promise
      sender:  = {Actor} 1073741833
    ▶ target:  = {Promise}
    ▶ turn: computePerimeter = {Message}
  ▶ square: = {Message}
  ▶ square: = {Message}
  ▶ square: = {Message}
```

Figure 5.2: Mailbox visualization for a paused actor.

The mailbox visualization is crucial to enable developers to inspect the actor to determine if it is in some *behavioral deadlock* (see Section 2.3.1.2). Developers can witness this issue because although the actor is paused, it is not blocked, i.e., it can receive messages from the other running actors of the program. Furthermore, the mailbox inspection can also help to identify unexpected interleavings of messages received by the actor.

5.1.3.2 Message Causality

Inspired by Causeway and REME-D debuggers, we designed a visualization to show the happened-before relationship of messages in an actor-based program. In our research, we explore the notions of Lamport's *space-time diagrams* used by [BWBE16] in the context of distributed systems, and we apply them to a message-oriented debugger for actor-based programs. Particularly, we propose a *graph visualization in a space-time diagram.*

We now briefly introduce the necessary elements from Lamport's happened-before relationship to be able to understand our visualizations. Our message causality design targets concurrent programs, then we consider Lamport's view of a program that consists of several processes, i.e., actors, consisting of a sequence of events, i.e., an event can be represented by sending or receiving a message in an actor. Here we present the conditions to be satisfied in the *happened-before relationship* defined by [Lam78]:

1. If a and b are events in the same process, and a comes before b, then $a \longrightarrow b$.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \longrightarrow b$.
3. If $a \longrightarrow b$ and $b \longrightarrow c$ then $a \longrightarrow c$. Two distinct events are said to be **concurrent** if $a \not\longrightarrow b$ and $b \not\longrightarrow a$.

As argued by [SCM09] showing the partial ordering of events narrows the search for the root cause of a bug. For example, an event a that happened before an event b can influence the later one. On the contrary, if a did not happen before b, then to explain a bug in event b we can ignore a. In addition to showing the order of messages sent by an actor, having information about the process order can give better debugging exploration space to developers [SCM09]. Like Causeway and REME-D, we consider that a debugger for actor-based programs can use information about executed turns and messages sent by each of the program actors. Although we propose different visualizations, we adopt two orders naming from Causeway debugger in our approach:

Process order represents the turns executed[6] by each actor of the program in chronological order, i.e., in the order the actor received the messages.

Message order shows the order in which each actor of the program sends messages.

We propose to show message causality information in an actor system through a *graph visualization* in which columns represent turns executed for one actor, i.e., the process order. In contrast to Lamport space-time diagrams, in our diagram, earlier turns are represented higher in the graph in our visualization.

Figure 5.3 shows an example of our graph visualization of turns and causal messages. The first row in the diagram shows all the actors of the program, which are represented

[6]turns executed = messages processed

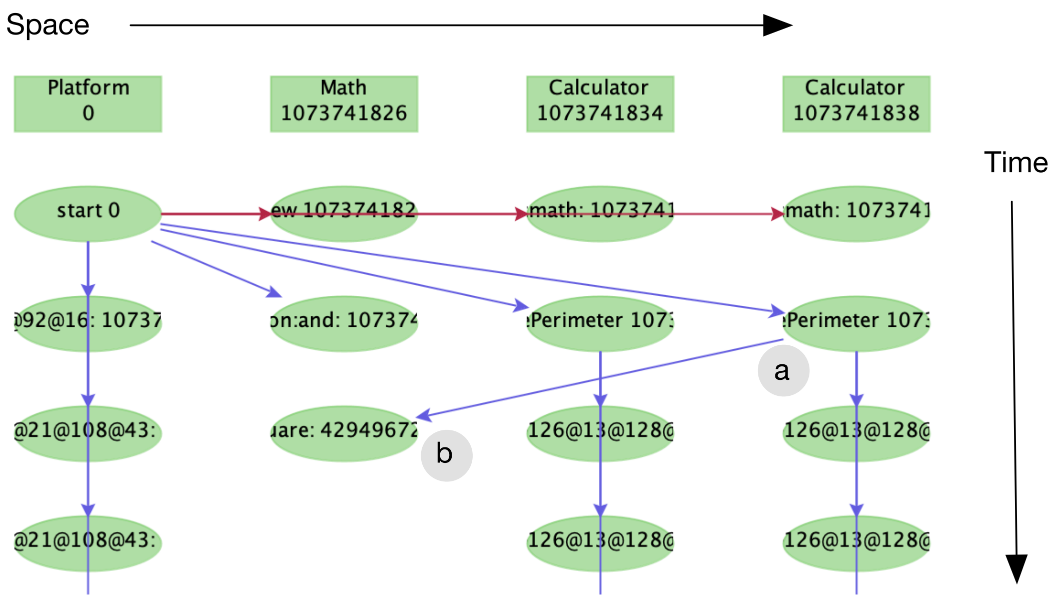
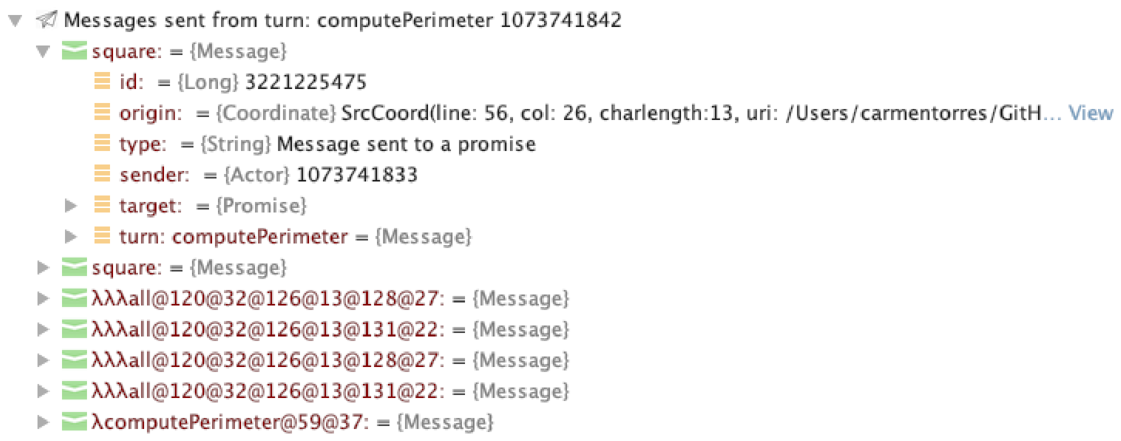


Figure 5.3: Graph visualization to show the process order with happened-before relationship between messages. Rectangles depict actors, ellipses depict turns and arrows depict messages. Turn a is causally link to turn b.

with rectangles. The color green indicates the actors are in run state. Each column represents the turns processed by each actor, and each turn is shown with an ellipse. Arrows indicate message sends between the actors and are causally link to the turn where it was sent. If the developer clicks one turn, there will be another view, i.e., a *sentbox view*, which shows a list of all messages sent (in order) from a selected turn in the graph (see figure 5.4). Similar as we showed in the mailbox, each message sent shown in the sentbox contains information about its turn. These views are part of our proof of concept debugger implemented as a plugin in the IntelliJ IDE (see section 5.2 and 5.2.2.2).

An initial version of our graph visualization was prototyped for the Kómpos debugger [MLA+17], but without the notion of a sentbox and message properties information. During the development of our graph visualization, we have discussed our design with different researchers from our laboratory as well as externals (e.g., INRIA RMOD group).

We think that exploring turns in a process and message order can be especially useful for detecting bugs such as message protocol violations, i.e., *message order violations* and *bad message interleavings*. Developers will be able to see in the process order turns execution and notice when there is a turn that is in the incorrect place of execution for an actor. Similarly, having the possibility to see the message sent in order will help to identify wrong or null receivers, e.g., if the expected target actor received the message.



```
▼ Messages sent from turn: computePerimeter 1073741842
  ▼ square: = {Message}
      id:  = {Long} 3221225475
      origin:  = {Coordinate} SrcCoord(line: 56, col: 26, charlength:13, uri: /Users/carmentorres/GitH... View
      type:  = {String} Message sent to a promise
      sender:  = {Actor} 1073741833
    ▶ target:  = {Promise}
    ▶ turn: computePerimeter = {Message}
  ▶ square: = {Message}
  ▶ λλλall@120@32@126@13@128@27: = {Message}
  ▶ λλλall@120@32@126@13@131@22: = {Message}
  ▶ λλλall@120@32@126@13@128@27: = {Message}
  ▶ λλλall@120@32@126@13@131@22: = {Message}
  ▶ λcomputePerimeter@59@37: = {Message}
```

Figure 5.4: Sentbox visualization that shows messages sent in order. The turn property denotes the causal turn for the selected message in the sentbox.

5.1.4 Asynchronous Stack Trace

Using the happened-before relationship of messages, one can understand the asynchronous message send that leads to a particular point of interest (i.e., a pausing state). However, this information needs to be combined with information on method activations. To understand how we got to a code location in a concurrent program is needed a stack that can show the asynchronous and synchronous events that activated the current frame.

We propose an asynchronous stack trace that shows the *calling context* and the *send context* for asynchronous messages. Furthermore, we propose that an asynchronous stack trace can be configured to provide information not only about all asynchronous messages that were sent previous to the suspension (i.e., the *control flow*), but the messages sent that lead to the resolution of promise objects (i.e., the *data flow*).

Visualizing the frames related to control flow, data flow, and method activations in the same stack can be overwhelming for a developer. As such, we propose to categorize the different kinds of information which are clearly depicted in the asynchronous stack of Figure 5.5. In this figure we show a proposal for visualizing the mentioned frames in a debugging session for a SOMNS program. In particular, we distinguish the following frame types:

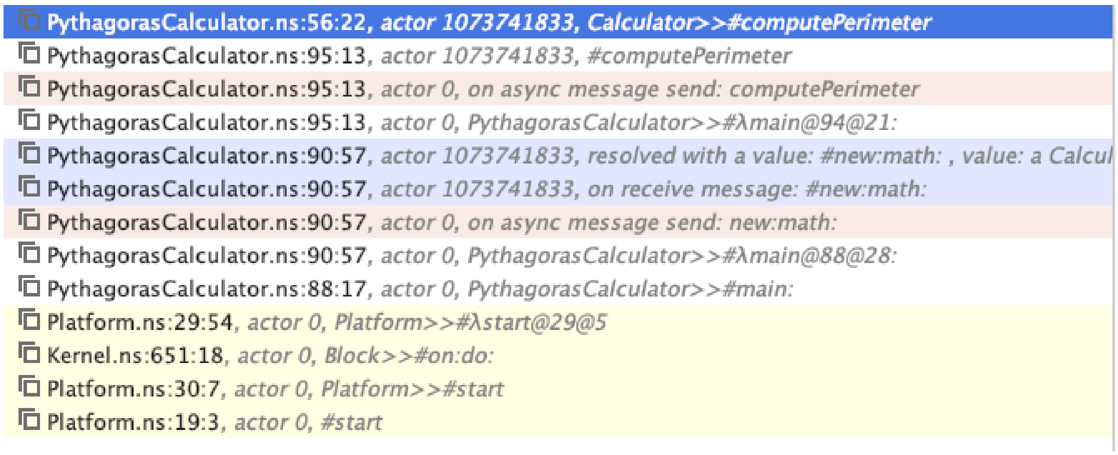
method activations frames in *white* color.

asynchronous messages sent frames in *pink* color.

promise resolution frames in *purple* color.

method activations in the runtime system frames in *yellow* color.

As can be observed, we also consider one color to distinguish method activations from the runtime system, as shown by traditional stack traces. We distinguish method activations that are due to runtime so that it is easier to identify end-user code. The coloring should help to reason about synchronous and asynchronous computation. As mentioned, in section Section 3.3.1, both information is necessary to find the root cause of bugs.



```
PythagorasCalculator.ns:56:22, actor 1073741833, Calculator>>#computePerimeter
PythagorasCalculator.ns:95:13, actor 1073741833, #computePerimeter
PythagorasCalculator.ns:95:13, actor 0, on async message send: computePerimeter
PythagorasCalculator.ns:95:13, actor 0, PythagorasCalculator>>#λmain@94@21:
PythagorasCalculator.ns:90:57, actor 1073741833, resolved with a value: #new:math: , value: a Calcul
PythagorasCalculator.ns:90:57, actor 1073741833, on receive message: #new:math:
PythagorasCalculator.ns:90:57, actor 0, on async message send: new:math:
PythagorasCalculator.ns:90:57, actor 0, PythagorasCalculator>>#λmain@88@28:
PythagorasCalculator.ns:88:17, actor 0, PythagorasCalculator>>#main:
Platform.ns:29:54, actor 0, Platform>>#λstart@29@5
Kernel.ns:651:18, actor 0, Block>>#on:do:
Platform.ns:30:7, actor 0, Platform>>#start
Platform.ns:19:3, actor 0, #start
```

Figure 5.5: Asynchronous stack trace visualization that shows frames for method activations (white color), frames for asynchronous messages send (pink color) and frames related to promise resolution (purple color).

An asynchronous stack trace is different from the mentioned message causality design (see section 5.1.3.2). An asynchronous stack trace will only show all the method activations, and the asynchronous message sends related to the suspension. However, the message causality feature provides a more extensive spectrum for the *send context* of the program because it shows *all* the messages processed and sent that have been executed until the suspension. Hence, we consider that the asynchronous stack trace narrows the control flow path of the program's execution.

5.1.5 Advanced Visualization Techniques

Finally, we would like to mention that as part of this research, we also explored advanced visualization techniques through interrogative debugging for actor-based programs. Interrogative debugging was first proposed for thread-based programs [KM08, KM10], with the goal of augmenting debugging tools with facilities to ask questions interactively related to program runtime failures in a debugging session.

We have supervised a bachelor thesis at the Vrije Universiteit Brussel [Ver20], which prototyped our ideas on the questions which we envisioned to support interrogative debugging for actor-based programs. Table 5.7 shows an overview of the questions and answers we devised for an online debugger for actor-based programs written in the CEL model. We categorized the questions into three groups. Two of the categories are related to entities in the CEL model, such as actors and messages. The third category is related to variables, which includes promises and primitives types [Ver20]. We believe interrogative debugging offers developers an alternative interactive way to inspect the actor state. The current proposal also explored questions to get information on promise resolutions. For example, developers can ask for an unresolved promise, which is a feature suggested to be added to Apgar by participants of the user study we conducted to validate our techniques (see Chapter 7). It remains as future work to integrate the prototype implementation by [Ver20] into Apgar, our proof-of-concept debugger implementing the proposed novel debugging features for SOMns programs, which we describe in the next section.

Question	Answer
1. Actors	
- Which promises are resolved? - Which promises are not resolved?	- All resolved promises of the actor. - All unresolved promises of the actor.
2. Messages	
- Which is the turn of message X?	- Information about the turn.
3. Variables	
All variables	
- Why has variable X that value?	- Timeline of assignments.
Promises	
- What is the resolution value of promise X? - When did promise X resolve?	- Resolution value. - Information about a selected promise in the timeline + Timeline of promise dependencies

Table 5.7: Overview of questions and answers when debugging a CEL actor-based program, from [Ver20].

5.2 Apgar, a Proof of Concept Online Message-oriented Debugger for SOMns

In this section, we describe our proof of concept debugger for the SOMns language where we integrate the debugging features designed for actor-based CEL programs described in the previous section. We call Apgar[7] to our debugging tool, which consists of the debugger backend, the Kómpos protocol, and the debugger frontend. A demo video of the debugger is publicly available[8] and it was used as part of a user study material we describe in Chapter 7.

[7]The name is inspired by the work of Dr. Virginia Apgar, which created the first standardized tool to evaluate newborn health, i.e., the Apgar score, `https://medlineplus.gov/ency/article/003402.htm`.

[8]`https://www.youtube.com/watch?v=3m-VuAfIrK0`

We start by giving an overview of the debugger architecture. Later, we describe the debugger views implemented in the frontend as a plugin for SOMns language in the IntelliJ IDE.

5.2.1 Architecture Overview

Figure 5.6 shows an overview of the debugger architecture. It consists of three main elements: the frontend, the backend, and a protocol to exchange information needed for debugging.

Apgar frontend has been implemented as a custom language plugin for the SOMns language in IntelliJ IDEA IDE. Besides incorporating the features for debugging actor-based programs described in the previous section, the frontend also has support for the development of SOMns programs expected from a modern IDE (e.g., editor with syntax highlighting, project packages, etc.)

Apgar backend (Medeor) has been implemented in the SOMns interpreter. Medeor[9], the main set of classes that manages the debugging information are built on top of the Truffle instrumentation framework, as mentioned in Section 4.4.1. This framework makes it possible to debug on the AST node level, i.e., to be able to set a breakpoint on a node definition instead of a line number. However, it does not provide support for concurrency. In the next chapter, we will explain the main implementation details we built to implement concurrent debugging support on Truffle by means of AST node wrapping.

Kómpos protocol is a concurrency-agnostic debugging protocol that mediates interactions between the backend and frontend.[10]. The Kómpos protocol was proposed in collaboration with Dr. Stefan Marr in the context of the MetaConc project with the Johannes Kepler University in which this research was funded [11]. In this work, we extended the original protocol for pausing and resuming actors and enabling our visualizations on actor state and message causality (see Section 5.3).

[9]Latin word meaning heal, cure, good against a disease.
[10]Messages between frontend and backend are exchanged using JSON format
[11]`https://ssw.jku.at/Research/Projects/MetaConc/`

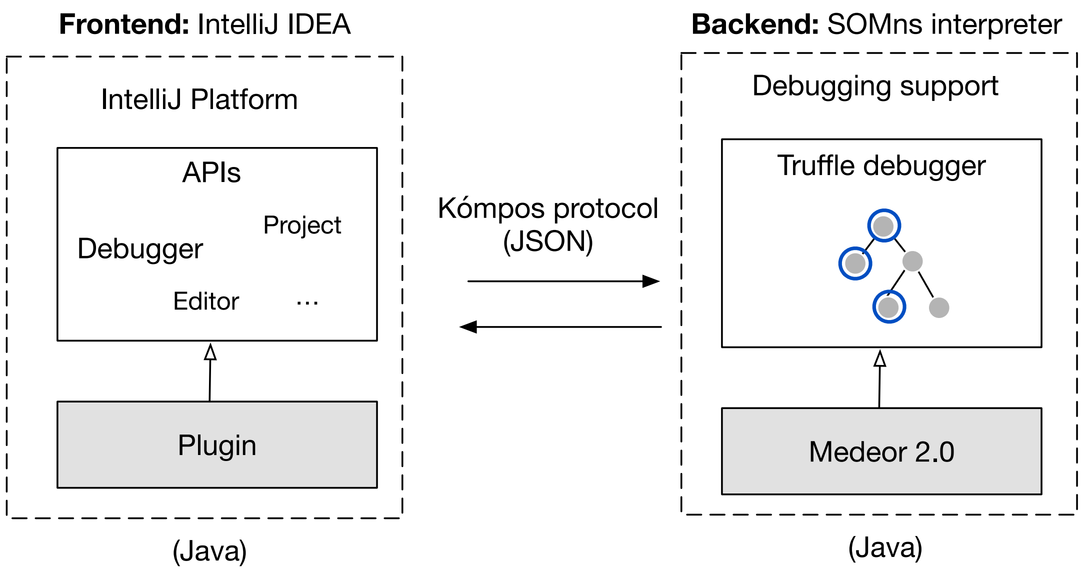


Figure 5.6: Architecture of the Apgar debugger.

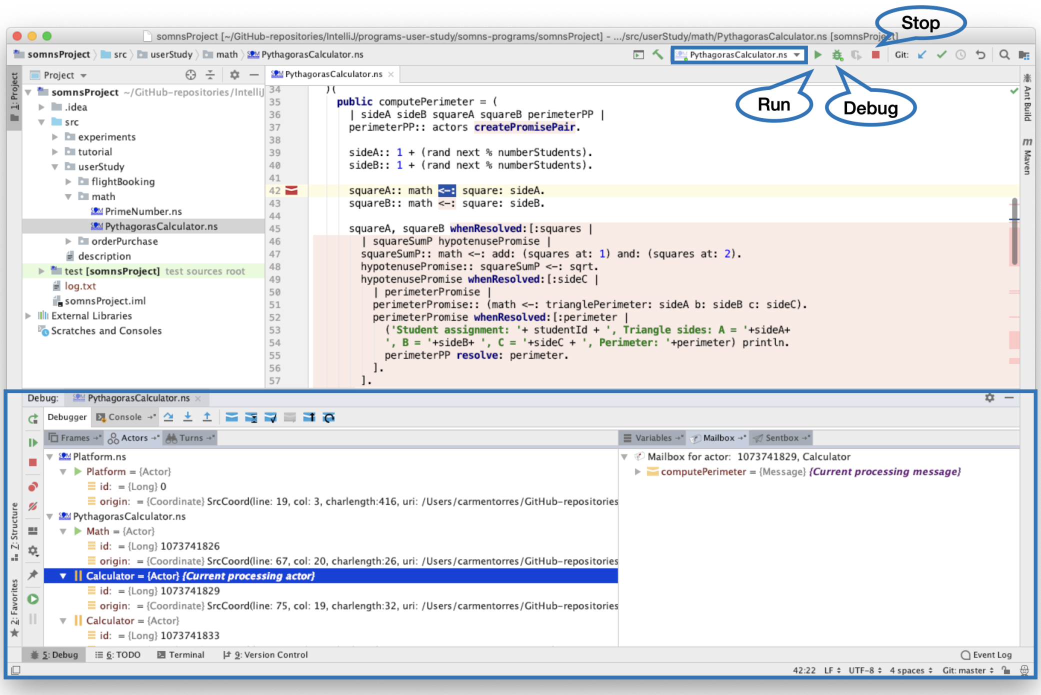
5.2.2 Apgar Frontend

In this section, we illustrate Apgar from the programmer's perspective interacting with the debugger frontend. Figure 5.7 shows the user interface of Apgar in IntelliJ, which consists of three main panels:

Project panel is located on the left and contains all the available programs of the open project. In the editor is shown the Pythagoras calculator program (see Appendix C.3). The red envelope next to line 42 indicates that a message breakpoint has been defined, in this case for the asynchronous operator declared in that line for the `square` message.

Editor panel is located in the center where the program's source code is visualized. Next to the program's name, there is the Run button (for running the program), the Debug button (for debugging the program), and the Stop button (for stopping the program's execution). On the top right, we can see highlighted a blue rectangle that shows the program's name to be run or debug. The figure shows in the editor in dark blue that the debugger paused the program execution on line 42 due to a message sender breakpoint, i.e., before the message `square` is sent (it is highlighted the asynchronous send operator node).

Debugger panel is located at the bottom and contains all the views for interactively debugging actor-based programs explained in the previous section. We describe the different debugging views in the next sections. We will use as a running example the Pythagoras calculator program.



```
34   )(
35     public computePerimeter = (
36       | sideA sideB squareA squareB perimeterPP |
37       perimeterPP:: actors createPromisePair.
38
39       sideA:: 1 + (rand next % numberStudents).
40       sideB:: 1 + (rand next % numberStudents).
41
42       squareA:: math <-: square: sideA.
43       squareB:: math <-: square: sideB.
44
45       squareA, squareB whenResolved:[:squares |
46         | squareSumP hypotenusePromise |
47         squareSumP:: math <-: add: (squares at: 1) and: (squares at: 2).
48         hypotenusePromise:: squareSumP <-: sqrt.
49         hypotenusePromise whenResolved:[:sideC |
50           | perimeterPromise |
51           perimeterPromise:: (math <-: trianglePerimeter: sideA b: sideB c: sideC).
52           perimeterPromise whenResolved:[:perimeter |
53             ('Student assignment: '+ studentId + ', Triangle sides: A = '+sideA+
54             ', B = '+sideB+ ', C = '+sideC + ', Perimeter: '+perimeter) println.
55             perimeterPP resolve: perimeter.
56           ].
57         ].
```

```
Platform.ns
  Platform = {Actor}
    id: = {Long} 0
    origin: = {Coordinate} SrcCoord(line: 19, col: 3, charlength:416, uri: /Users/carmentorres/GitHub-repositories
PythagorasCalculator.ns
  Math = {Actor}
    id: = {Long} 1073741826
    origin: = {Coordinate} SrcCoord(line: 67, col: 20, charlength:26, uri: /Users/carmentorres/GitHub-repositories
  Calculator = {Actor} {Current processing actor}
    id: = {Long} 1073741829
    origin: = {Coordinate} SrcCoord(line: 75, col: 19, charlength:32, uri: /Users/carmentorres/GitHub-repositories
  Calculator = {Actor}
    id: = {Long} 1073741833
```

```
Mailbox for actor: 1073741829, Calculator
  computePerimeter = {Message} {Current processing message}
```

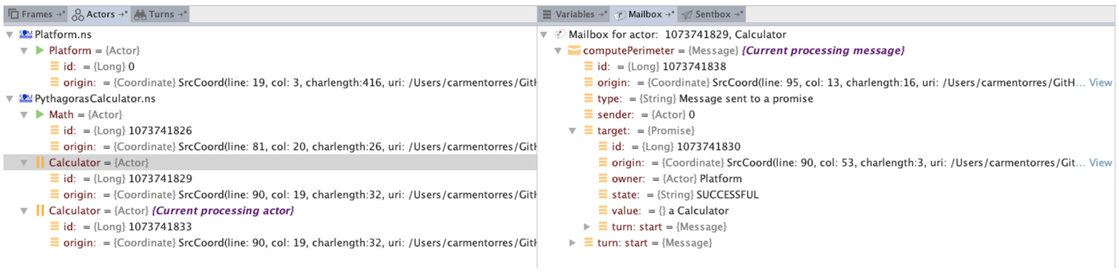
Figure 5.7: User interface for the Apgar frontend in IntelliJ IDE.

5.2.2.1 Actors and Mailbox

In Section 5.1.3.1 we mentioned how we designed visualizations to inspect the state of a paused actor. Figure 5.8 shows the *Actors view* on the left panel and the *Mailbox view* on the right panel. These views show the state of the mailbox at the point where the actor got suspended (corresponding to line 42 in Figure 5.7 as mentioned before). The Actors view shows all actors active in the program execution. In this example four actors are active, i.e., `Platform`, `Math`, and two `Calculator` actors. Actors can be in *run* or in *pause* state. Actors are described with its *id*, a *state* (`running` represented with a green icon and `paused` represented with a yellow icon) and a *source coordinate* corresponding to the code location where it was created using the `createActorFromValue` message. Recall from Chapter 4 that `Platform` actor is the main actor in the runtime system, which runs a SOMNs program.

In the Actors view, it is also possible to select one of the program's running actors and pause it. This means that the actor will halt execution before processing the next message in its mailbox. The actors' view allows developers to see how many actors have been created in the program and inspect their state. An enlarged view of the actors' view is shown in Figure 5.9.

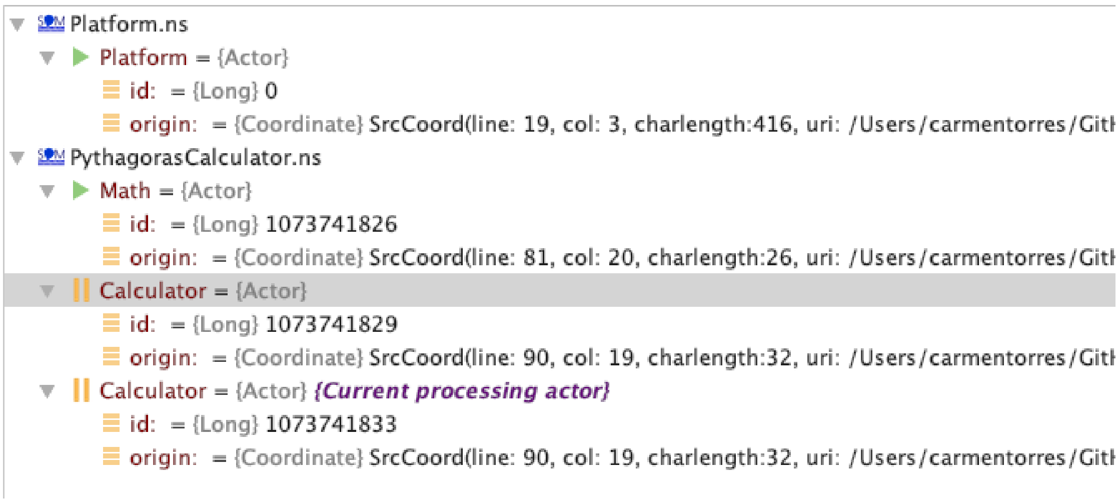
The state for the selected paused actor is shown in the Mailbox view, with all the messages they have received until the suspension. The yellow message corresponds to the current processing message, and gray messages are waiting to be processed. If the developer clicks on the message the following properties are displayed: an *identifier*, its *origin* (code location where it was created), the *type*, the *id* of the sender actor, the *target* of the message and the *turn* where this message was sent. Right-clicking on one message allows developers to jump to the source location in the editor where that message was sent. In Figure 5.8 we can see that `computePerimeter` message is the message about to be processed by the selected `Calculator` actor. The message properties are visualized and the properties for the target promise object of that message. An enlarged view of the mailbox, but for another debugging session was shown in, Section 5.1.3.1, Figure 5.2.



```
Platform.ns
  Platform = {Actor}
    id:  = {Long} 0
    origin:  = {Coordinate} SrcCoord(line: 19, col: 3, charlength:416, uri: /Users/carmentorres/Gitl
PythagorasCalculator.ns
  Math = {Actor}
    id:  = {Long} 1073741826
    origin:  = {Coordinate} SrcCoord(line: 81, col: 20, charlength:26, uri: /Users/carmentorres/Gitl
  Calculator = {Actor}
    id:  = {Long} 1073741829
    origin:  = {Coordinate} SrcCoord(line: 90, col: 19, charlength:32, uri: /Users/carmentorres/Gitl
  Calculator = {Actor} {Current processing actor}
    id:  = {Long} 1073741833
    origin:  = {Coordinate} SrcCoord(line: 90, col: 19, charlength:32, uri: /Users/carmentorres/Gitl
```

```
Mailbox for actor:  1073741829, Calculator
  computePerimeter = {Message} {Current processing message}
    id:  = {Long} 1073741838
    origin:  = {Coordinate} SrcCoord(line: 95, col: 13, charlength:16, uri: /Users/carmentorres/GitH... View
    type:  = {String} Message sent to a promise
    sender:  = {Actor} 0
    target:  = {Promise}
      id:  = {Long} 1073741830
      origin:  = {Coordinate} SrcCoord(line: 90, col: 53, charlength:3, uri: /Users/carmentorres/Git... View
      owner:  = {Actor} Platform
      state:  = {String} SUCCESSFUL
      value:  = {} a Calculator
      turn: start = {Message}
    turn: start = {Message}
```

Figure 5.8: Actors and Mailbox views.



```
Platform.ns
  Platform = {Actor}
    id:  = {Long} 0
    origin:  = {Coordinate} SrcCoord(line: 19, col: 3, charlength:416, uri: /Users/carmentorres/Gitl
PythagorasCalculator.ns
  Math = {Actor}
    id:  = {Long} 1073741826
    origin:  = {Coordinate} SrcCoord(line: 81, col: 20, charlength:26, uri: /Users/carmentorres/Gitl
  Calculator = {Actor}
    id:  = {Long} 1073741829
    origin:  = {Coordinate} SrcCoord(line: 90, col: 19, charlength:32, uri: /Users/carmentorres/Gitl
  Calculator = {Actor} {Current processing actor}
    id:  = {Long} 1073741833
    origin:  = {Coordinate} SrcCoord(line: 90, col: 19, charlength:32, uri: /Users/carmentorres/Gitl
```

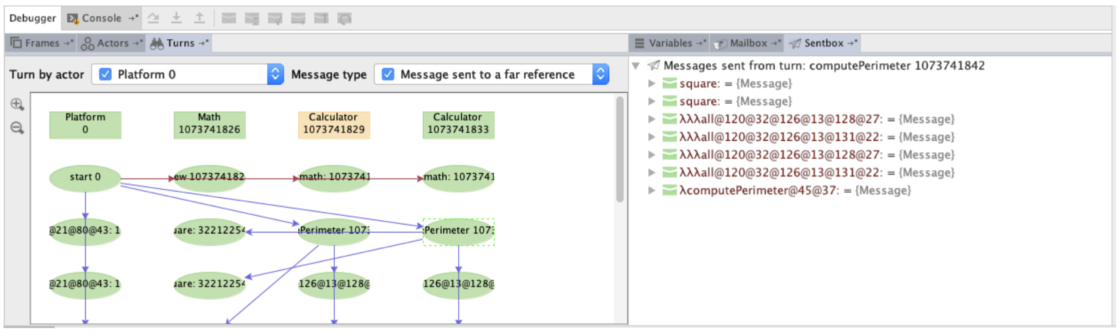
Figure 5.9: Actors visualization showing running (green color) and paused state (yellow color).

5.2.2.2 Turns and Sentbox

Section 5.1.3.2 explained the main elements of our novel graph visualization for causality of messages based on a space-time diagram. We now discuss further the interactions between the *Turns view* showing the space-time diagram and the *Sentbox view* for a given turn.

Figure 5.10 shows the Turns view on the left panel and the Sentbox view on the right panel. The Turns view provides a visualization of turns per actor. Rectangles represent actors, and ellipses represent turns. The yellow color indicates the actor or turn is paused, and the green color indicates that the actor or turn is running. Arrows represent messages. The ones in red color indicate the *message was sent to a far reference* and the purple color indicates a *message sent to a promise*. There are also buttons for zoom in and zoom out the graph. The visibility of each actor column and the messages can be selected using the checkboxes at the top of the visualization.

When developers select a turn in the graph, highlighted by the green dotted line, the Sentbox view shows messages that have been sent in the selected turn. The visualization in Figure 5.10 shows one of the `Calculator` actors in paused state, and the sentbox for the second turn of the last `Calculator` actor.



```
▼ Messages sent from turn: computePerimeter 1073741842
  ▶ square: = {Message}
  ▶ square: = {Message}
  ▶ λλλall@120@32@126@13@128@27: = {Message}
  ▶ λλλall@120@32@126@13@131@22: = {Message}
  ▶ λλλall@120@32@126@13@128@27: = {Message}
  ▶ λλλall@120@32@126@13@131@22: = {Message}
  ▶ λcomputePerimeter@45@37: = {Message}
```

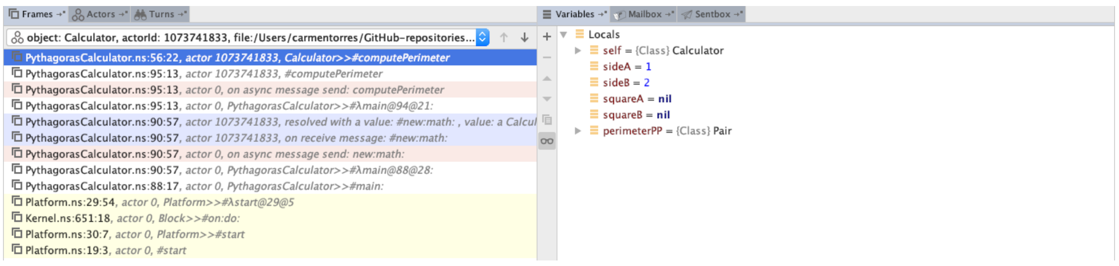
Figure 5.10: Turns and Sentbox views.

5.2.2.3 Frames and Variables

In Section 5.1.4 we showed the conceptual design of an asynchronous stack trace for actor-based programs. We now describe the integration of asynchronous stack traces in the *Frames view*. Figure 5.11 shows that view on the left panel, and the variables state for the selected frame in the stack on the right panel, called *Variables view*.

The Frames view shows the current frame corresponding to the method of the suspension (highlighted in dark blue), together with frames corresponding to the *method activations* (white color), *asynchronous message sent* (in pink color) and *promise resolution frames* (in purple color). Above the list of frames, developers can inspect the list of paused actors and the variables state on the right panel.

By default, the debugger frontend enables the creation of an asynchronous stack trace in the debugger backend, but it can be disabled also by developers to see a traditional call stack (i.e., without frames for asynchronous message send or promise resolution).



```
object: Calculator, actorId: 1073741833, file:/Users/carmentorres/GitHub-repositories...
PythagorasCalculator.ns:56:22, actor 1073741833, Calculator>>#computePerimeter
PythagorasCalculator.ns:95:13, actor 1073741833, #computePerimeter
PythagorasCalculator.ns:95:13, actor 0, on async message send: computePerimeter
PythagorasCalculator.ns:95:13, actor 0, PythagorasCalculator>>#λmain@94@21:
PythagorasCalculator.ns:90:57, actor 1073741833, resolved with a value: #new:math: , value: a Calcul
PythagorasCalculator.ns:90:57, actor 1073741833, on receive message: #new:math:
PythagorasCalculator.ns:90:57, actor 0, on async message send: new:math:
PythagorasCalculator.ns:90:57, actor 0, PythagorasCalculator>>#λmain@88@28:
PythagorasCalculator.ns:88:17, actor 0, PythagorasCalculator>>#main:
Platform.ns:29:54, actor 0, Platform>>#λstart@29@5
Kernel.ns:651:18, actor 0, Block>>#on:do:
Platform.ns:30:7, actor 0, Platform>>#start
Platform.ns:19:3, actor 0, #start
```

```
Locals
  self = {Class} Calculator
  sideA = 1
  sideB = 2
  squareA = nil
  squareB = nil
  perimeterPP = {Class} Pair
```

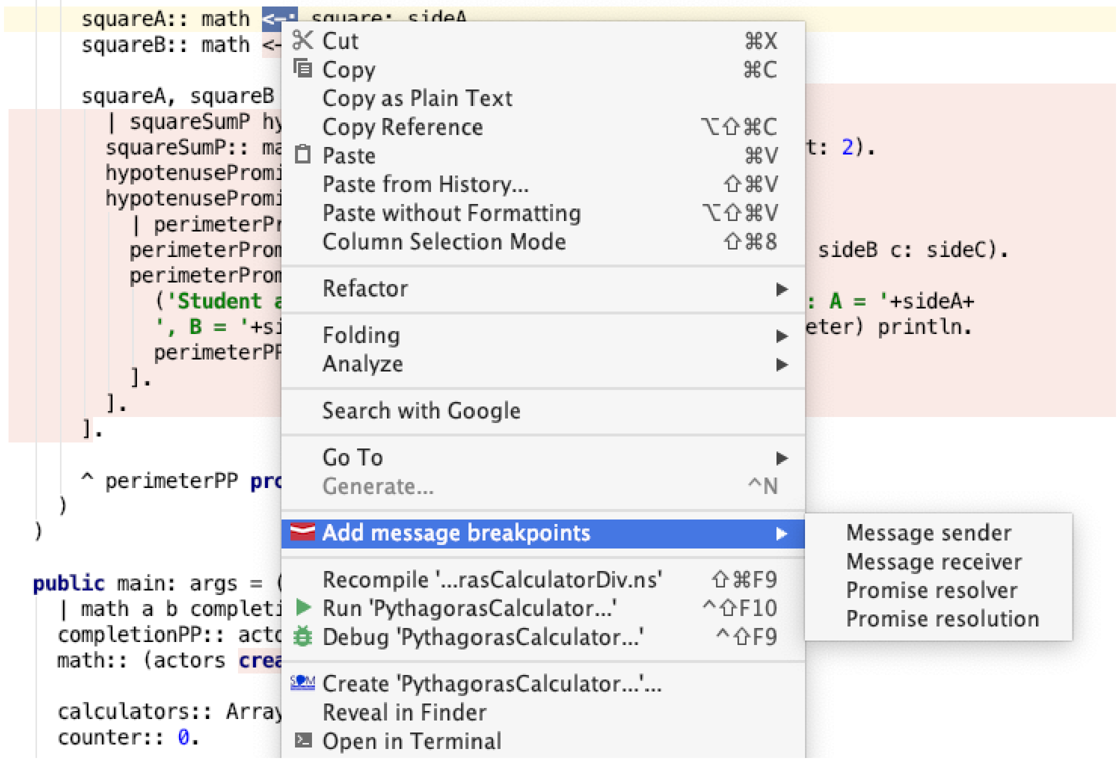
Figure 5.11: Frames and Variables views.

5.2.2.4 Breakpoints and Stepping commands

Section 5.1.1 and Section 5.1.2 describe the catalogs we proposed for message-oriented breakpoints and stepping operations. In Figure 5.12 we illustrate with an example how we can define message breakpoints in Apgar. The figure shows a code snippet of the Pythagoras calculator program opened in the editor panel of Apgar frontend. The contextual menu depicts four breakpoint options that are visualized due to the selection of the asynchronous send operator in Line 38. Regions in the editor in pink color represent AST nodes where a breakpoint can be placed. Besides, as mentioned in Section 5.1.1 it is possible to add promise breakpoints on messages that return promises such as `createPromisePair:`, `whenResolved:onError:`, `whenResolved:`, `onError:`. Finally, the developer can set asynchronous before and asynchronous after breakpoints on method names. Figure 5.13 shows all the buttons related to the sequential and message-based stepping operations.

5.3 Extension to Kómpos Protocol

Inspired by Visual Studio Code debugging protocol [Mic21], Kómpos is a concurrency-agnostic debugger protocol that decouples the debugger from the concurrency models employed by the target application [MLA+17]. In particular, the protocol defines communication between the debugger frontend and the interpreter in a concurrency agnostic way by modeling concurrency concepts relevant for debugging with breakpoints and stepping operations. Kómpos original version was derived from the first version of our catalogs for message-oriented breakpoints and stepping operations for actors and transposed to other concurrency models [MLA+17]. As mentioned before, in this work, we report the extended version of the protocol to support our online debugging for actor-based programs and more advanced visualizations than initially foreseen.



```
    squareA:: math <-: square: sideA.
    squareB:: math <-

    squareA, squareB
      | squareSumP hy
      squareSumP:: ma                                  t: 2).
      hypotenusePromi
      hypotenusePromi
        | perimeterPr
        perimeterProm                                  sideB c: sideC).
        perimeterProm
          ('Student a                                  : A = '+sideA+
          ', B = '+si                                  eter) println.
          perimeterPP
        ].
      ].
    ].

    ^ perimeterPP pro
  )
)

public main: args = (
  | math a b completi
  completionPP:: acto
  math:: (actors crea

  calculators:: Array
  counter:: 0.
```

Figure 5.12: Breakpoints menu visualized for the asynchronous send operator.

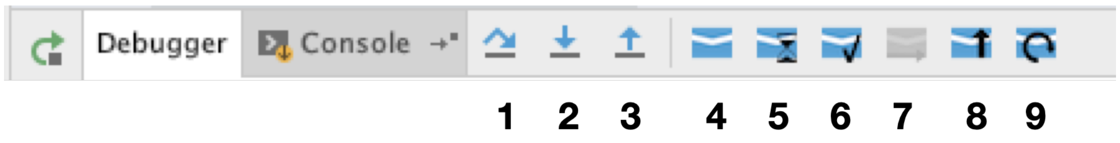


Figure 5.13: Stepping commands. Sequential stepping: (1) step over (2) step into (3) step out. Message stepping: (4) step to message receiver (5) step to promise resolver (6) step to promise resolution (7) step next turn (8) return from turn to promise resolution (9) step end turn.

Figure 5.14 shows a class diagram for the main elements of the Kómpos protocol organized in three categories: Meta model, Debugger messages, and Trace events. The meta model (1) describes the concurrency and debugger concepts supported by the interpreter and metadata used to interpret the exchanged messages. The debugger messages (2) are used to update the frontend or the interpreter, e.g., breakpoints, stepping, pause and resume commands. The trace events (3) encode data collected by the interpreter on program behavior for visualization. Elements in blue color represent our extension to the original version of the protocol, published in [MLA$^+$17]. In the following sections, we described the semantics for each of the elements in the protocol, focusing on its application for debugging actor-based programs.

5.3.1 Meta Model

The Kómpos protocol derived many concurrency concepts relevant for debugging based on activities, dynamic scopes, passive entities, send operations and receive operations. Table 5.8 shows the definition of those concepts applied to the actor-based model of concurrency. The debugger, however, requires meta data to match debugging operations to these concurrency concepts. This information is part of the meta model. In what follows, we detail the concurrency and debugger concepts in Kómpos meta model.

Concurrency concept	Definition	CEL
activity	active entity that execute code	actors
dynamic scope	well-structured and nested parts of a program's execution during which certain concurrency related properties hold	turns
passive entity	entities that do not act themselves, but are acted upon	messages promises
send operations	interaction between entities that initiates communication or synchronization	send message resolve promise

Table 5.8: Concurrency concepts for the CEL concurrency model from the Kómpos protocol [MLA+17].

The Kómpos meta data consists of the eight concepts for debugging shown at the top of Figure 5.14.

An `EntityType` defines data common to all entity types, i.e., activity types, dynamic scopes types and passive entities types. All entities have a label, i.e., a name and a unique id to distinguish them. In CEL programs, entity types correspond to actors, turns, messages, and promises, as shown in Table 5.8.

An `ActivityType` represents active entities that execute code, i.e., *actors* in our work. A `DynamicScopeType` is used to determine the scope for operations. In our work, *turns* will delimit for example the possible message stepping operations. A `PassiveEntityType` is used mainly for visualization, and in CEL programs, corresponds to messages and promises.

A `BreakpointType` defines the possible breakpoints. Each breakpoint has a unique name, a label to be used in the user interface, and an applicability criterion based on source tags to identify the semantics elements contained in a source range. We will use Truffle's tagging mechanism to annotate AST nodes in the interpreter (see Chapter 6). If a breakpoint type does not specify any source tags, it applies to all source locations.

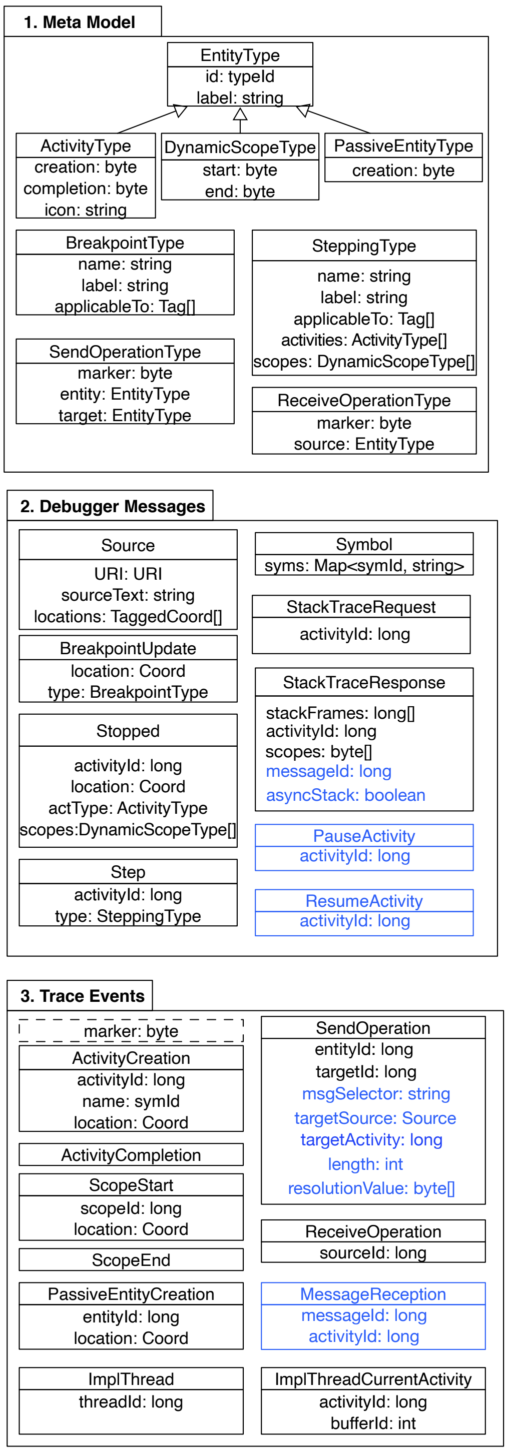


Figure 5.14: Class diagram of the main elements of the Kómpos protocol.

A `SteppingType` defines the stepping operations to follow program execution sequentially or concurrently. As for breakpoints, stepping operations are distinguished by the name and the source tags that will be used to define whether a stepping operation is applicable. For instance, the operation to step to the receiver of a message is only available on a message send operation. An additional applicability criterion is the current activity type, e.g., to enable stepping to the next turn for actors. Similarly, stepping can be conditional to the current dynamic scope. As such, the third applicability criterion is the scope active for the current execution.

A `SendOperationType` is used for visualization of *send message* and *resolve promise*. A send operation type defines a unique marker, the sender entity, and the target to which this operation belongs to. A `ReceiveOperationType` is used for visualizations of receive operations, but currently, they are not used for CEL programs.

5.3.2 Debugger Messages

In its original version, the Kómpos protocol provides five messages that the debugger and interpreter can exchange for debugging. Figure 5.14 shows two additional messages, i.e., requests and responses for stack trace information. We also added two new messages to indicate that an activity (i.e., an actor in our case) has been paused or resumed in the debugger frontend.

A `Source` message provides the source information to the frontend. The Source message includes a URI to identify the source file or resource, the source text, and a list of tagged source locations. Source locations specify the exact coordinates, for instance, based on a line number, column number, and character length. The tags are opaque identifiers that identify concurrency operations. Tags are also useful for highlighting the breakpoint nodes in the source editor, i.e., all the nodes from the program AST in which it is possible to define breakpoints. In our concrete debugger implementation, we use the notion of PSI elements from IntelliJ[12] to obtain source coordinates for breakpoints that the debugger frontend sends to the backend when initializes connection (see Appendix D.3.1)[13].

A `BreakpointUpdate` message is used to communicate breakpoints from the frontend to the backend. It encodes the source location and the breakpoint type.

A `Stopped` message is sent from the backend to the frontend to indicate that either a breakpoint was hit or a stepping operation completed. It identifies the current location and the suspended activity with id and type. Furthermore, it includes a list of currently active dynamic scopes for this activity. Note that the activity type and active dynamic scopes can also be determined from the trace data, but providing them explicitly simplifies the debugger implementation.

[12]`https://plugins.jetbrains.com/docs/intellij/psi.html`

[13]This strategy differs from the first debugger frontend implementation for SOMns which was implemented as a web application.

A `Step` message is sent from the debugger frontend to the debugger backend to instruct the latter to resume execution of a specified activity with a given stepping type.

A `Symbol` message avoids sending long strings repeatedly by sending a symbol table from the interpreter to the debugger.

The `StackTraceRequest` and `StackTraceResponse` are messages exchanged by the debugger frontend and interpreter due to a breakpoint or a stepping operation. A `StackTraceRequest` models the request message and `StackTraceResponse` models the response carrying the information to show state of the paused actor. We added in `StackTraceResponse` the message id corresponding to the current processing message at which the suspension occurs. This is needed for the visualization of debugging information in the frontend (e.g., to show the state of the paused actor). Besides, we declare a flag that indicates if the asynchronous stack is enabled.

In the original version of the protocol, pausing actors was only possible via breakpoints or stepping operations. For indicating that an activity has been paused or resumed explicitly by the end-user at the debugger frontend, we added `PauseActivity` and `ResumeActivity`. A `PauseActivity` message is sent from the debugger frontend to the backend to pause a given actor by activating a flag in the actor to execute a step. This means the actor will pause execution before processing the next message in its mailbox. The message response for pausing an actor that is sent to the frontend will update in the frontend the actor's state to running. A `ResumeActivity` message is sent from the debugger frontend to the backend to resume a given actor. In the debugger, the actor's state will be updated to running. This message is defined by the actor id.

5.3.3 Trace Events

The Kómpos protocol provides details on the execution of a concurrent program through trace events that encode the program execution in terms of eight different trace entries depicted in Figure 5.14. These trace events can be used, for instance, to visualize concurrent interactions. In general, each trace event starts with a marker, which is indicated by the dashed line in Figure 5.14. The relation between the concrete marker and a concurrency concept is defined via the metadata elements previously described. Here we describe each of the trace events, which the exception of `ReceiveOperations` which is not used when recording actor events.

An `ActivityCreation` event records the id of the created actor, its name, and the source location of the creation operation. An `ActivityCompletion` is merely a marker recording that an actor terminated. The corresponding activity id can be determined from the complete trace.

A `ScopeStart` event records the beginning of a dynamic scope. It records the id of a scope, which corresponds to, e.g., the message id for an actor turn. It also includes the source location for the scope, e.g., the method invoked for an actor turn. A `ScopeEnd` event is also a marker that can be matched to the scope start implicitly.

A `PassiveEntityCreation` event records the id of the passive entity created and the source location of the operation.

A `SendOperation` event is defined by the id of the involved passive entity, e.g., a message, and the target entity id, e.g., the receiving actor. Also, we added an entry to record the message selector, the target source section, and the target actor id. This new information is needed for showing the state of a paused actor at the debugger frontend. Information about the sending entity can be inferred from the trace based on the dynamic scope or current activity. Moreover, we added two fields needed to communicate to the frontend the resolution value of promises, i.e., resolution value and its length. Specifically, this is visualized in the mailbox and sentbox, in the properties of a promise object (see Figure 5.8).

An `ImplThreadCurrentActivity` indicates the actor that is currently executing. Also, when the buffers are swapped, the current activity should be received (see Section 6.2.1.2). This event is defined by an actor id and a buffer id. Moreover, an `ImplThread` event indicates when a new chunk of trace starts. It is defined by a thread id.

In the first work using the Kómpos protocol, the `ReceiveOperation` message was not used in the trace of the CEL model because messages are implicitly received, and the `SendOperation` message carries the target entity id. We extended the tracing implementation in the debugger backend to incorporate this information for CEL actors, but to avoid confusion with the initial version of the protocol, we use a different name, i.e., `MessageReception`. This message represents the reception of a message in the actor's mailbox. This information is needed to show the received messages of the mailbox for the paused actor in order of arrival. Besides, this information is necessary for the visualization of turns processed by an actor in the frontend. One message reception is defined by the message id and the actor id.

5.4 Comparison to Related Work

In this section, we compare the debugging techniques we propose in this chapter with the state of the art of debuggers for actor-based programs which we surveyed in Chapter 3. Table 5.9 compares Apgar features with state of the art. We have marked with an 'X' every debugger that has addressed a technique. We have employed 'X*' those cases in which limited support is provided for the feature. We now discuss the approaches per debugging technique.

As we can observe, most of the debuggers provide *actor state inspection*. More concretely, all debuggers except Causeway have reported facilities to inspect the mailbox as well as local variables (or objects) of the actor. Chrome DevTools, McFly, Erlang and Actoverse do not provide an independent view dedicated to show the messages in the mailbox.

	Actor St. Insp.	Msg. Bkp	Msg Stepp.	Visualiz. Causality	Async. Stack.	Record Replay	Reverse Debug.
Online debuggers							
REME-D [GBNDM14]	X	X	X	X*			
Chrome DevTools [Not17]	X*				X		
Erlang [AB20]	X*						
ScalaIDE [fED]	X		X*		X		X*
Offline debuggers							
Causeway [SCM09]				X			
Actoverse [SW17]	X*	X*		X		X	X*
IDeA [MOM18]	X	X*	X*	X		X	X
McFly [VBMM18]	X*					X	X
Apgar	X	X	X	X	X		

Table 5.9: Overview of Apgar features with the state of the art debuggers for actor-based programs. A 'X' indicates that the debugging technique is addressed by the debugger. A 'X*' indicates that the support for that debugging technique is limited.

Besides Apgar, REME-D, Actoverse, and IDeA debuggers have support for *message-oriented breakpoints*. However, IDeA only provides breakpoints to halt before messages are received by the actor [MOM18], and Actoverse mentioned breakpoints on the sender and receiver side of the message [SW17]. Neither IDeA or Actoverse feature breakpoints on promises. In comparison to REME-D, our proposal further explores breakpoints on promises, and integrates it with stepping operations; REME-D only features one type of breakpoint for promises [GBNDM14].

Concerning the *message stepping operations* in state of the art, besides REME-D, there are two approaches for the Akka framework that have proposed stepping operations on the level of messages, i.e., ScalaIDE with step to the receiver of the message [fED] and IDeA which steps to the next message execution [MOM18]. However, those approaches do not offer a range stepping operations as Apgar (e.g., none of the approaches supports stepping at the level of promises), and do not integrate message stepping with sequential stepping operations.

With respect to *visualizations* features for message causality, it has only explored in state of the art of offline debuggers such as Causeway [SCM09]. REME-D debugger keeps the information of the turn where a message was sent, however, it does not provide a dedicated visualization to show the happened-before relationship of messages in its

Eclipse plugin frontend [GBNDM14]. As shown in Section 3.3.3, Causeway used tree views for displaying the causal information related to processes and messages. More recently, some works have explored visualizations based on sequence diagrams [SW17] and virtual reality [MOM18]. In Apgar, we propose a graph visualization in a space-time diagram to show process and message order. Space-time diagrams has been used before for visualizing messages in distributed systems [BWBE16], but not for debugging actor-based programs as in Apgar.

Similar to Apgar, mainstream debuggers for Scala and JavaScript feature *asynchronous stack traces*. In particular, the ScalaIDE for Akka programs shows a stack of asynchronous messages, including two kinds of stack frames, i.e., the history of frames when a promise is created and when a message is sent to an actor [Doc21]. Interestingly, they include state for asynchronous message sends, something that Apgar does not store yet. However, ScalaIDE is not reported to show frames corresponding to promise resolution or promise dependencies (i.e., promise callbacks). The Chrome debugger, on the other hand, visualizes the stack of functions that are called asynchronously and can show promise dependencies for JavaScript programs [LM18]. Similarly, Apgar includes frames related to asynchronous message send, and promise resolutions, including promise dependencies. In particular, Apgar asynchronous stack support is represented by frames corresponding to AST locations causally related to promise resolutions. In other words, the Apgar stack can distinguish different entries in the asynchronous stack, e.g., when a promise is resolved (i.e., with a value, with an error, or with another promise) and when a callback has been registered to a promise (e.g., with `whenResolved:` message in SOMNS). Also, the stack will show frames not only for implicit promises created with the asynchronous send operator and promise callbacks but also it will show the program execution flow for explicit promises. We believe that Apgar's stack frame visualization based on colors can aid developers to identify faster the frames related to interactions between actors, i.e., asynchronous message sends and promise resolution, from methods activations. That is useful since, many times, erroneous steps leading to an application failure are due to both sequential code and concurrent interactions.

Even though we have focused on online debugging, Apgar provides support for message causality, a feature typically only supported by offline debuggers. As we mentioned in Section 3.3.2, in the recent past, some academic work has focused on offline debugging techniques for actor-based languages, in particular reverse debugging and record and replay. For completeness, we include them in the overview table, but incorporating further offline debugging features in Apgar is future work which we discuss later in Section 9.4.

5.5 Conclusion

In this chapter, we have proposed the design of different online debugging techniques for actor-based programs.

First, we have devised a breakpoint catalog to allow developers to pause the program at four different halting locations in the context of asynchronous message passing. Besides, we have classified the debugging operations to cover three concurrency concepts present in communicating event-loops programs, i.e., messages, promises, and turns. Our work models also combine sequential and message-oriented stepping. We think this is a relevant aspect because the root cause of the bug can be originated in sequential or asynchronous code, which can help developers to identify *lack of progress issues* and *message protocol violations*.

Second, we proposed two visualizations related to the actor's mailbox inspection and the happened-before relationship of messages sent by the actors. Current debuggers for actor-based programs lack good visualization techniques to show the debugging information. Then we think our proposal gives a step in the direction to overcome that problem. These visualizations have been implemented using execution trace data that encodes the program's behavior on different trace events. In this approach, the debugger frontend requests to the backend the information about the concurrent entities until the point of the suspension in the program. We consider that the actor's mailbox visualization can help developers to identify mainly *lack of progress issues*. Besides, a graph visualization in a space-time diagram to show causality between the messages sent of the actors of the program can aid to identify particularly *message protocol violations*.

Third, we have designed an asynchronous stack trace that provides information about the calling and send context of a suspension in a debugging session. The novel feature of our stack trace is the combination of different kinds of frames related not only to asynchronous messages but also to promise resolutions. This could help to identify, for example, unresolved promises due to a behavioral deadlock. Although our implementation is a first proposal for the SOMns research language, we think asynchronous stack traces that provide the mentioned frame types can be applied to other actor languages to assist developers in identifying *message protocol violations*.

Finally, we have extended the Kómpos protocol with some elements needed to model online debugging features (e.g., pause and resume activities) as well as the visualization of the actor's state and message causality. Furthermore, this extension allows developers to switch between visualizing a traditional stack trace or an asynchronous stack. In the next chapter, we describe the implementation details related to our proof of concept debugger for SOMns programs, Apgar.

Chapter 6

Implementation of Online Debugging Techniques for SOMns

This chapter describes the implementation of the different online debugging features for the SOMns language. First, we focus on the relevant parts of the debugger backend in SOMns (Medeor). We describe the implementation of the catalogs of breakpoints and stepping operations for online debugging as well as support for asynchronous stack traces. Besides, we explain our changes in the Kómpos trace to provide support for inspecting the actor state and showing message causality. Finally, we explain our implementation of our debugger frontend as an IntelliJ plugin. In particular, we detail how we process the trace data obtained from the backend through the Kómpos protocol to visualize the actor's state, message causality, and the asynchronous stack when a suspension occurs. For reproduction purposes, implementation details, including class diagrams and sequence diagrams for the interaction between frontend and backend, are included in Appendix D.

6.1 Apgar Backend (Medeor), Debugging Support in SOMns

Medeor is the backend component of our online debugging support in the SOMns interpreter. It is implemented as an extension of the Truffle Debug API. We added support in Medeor for message-oriented breakpoints and stepping, and implemented asynchronous stack trace support that provides control flow and data flow information about the program suspension. Besides, we extended the implementation of the Kómpos protocol in SOMns to obtain debugging information related to the added debugger and the trace events described in Section 5.3.

Appendix D.1.1 shows an overview of the main classes of Medeor [1] to create a debugger on top of the Truffle Debug API. Recall that the Truffle Debug API only supports sequential debugging for the SOMns interpreter. In the following sections, we explain

[1]The complete implementation of Medeor is available in a fork of SOMns at `https://github.com/ctrlpz/SOMns/tree/somns-intellij-4.5/src/tools/debugger`

our implementation in Medeor to provide advanced debugging support for concurrent programs and its integration with sequential debugging.

6.1.1 Message Breakpoints

Medeor implements two types of breakpoints, line breakpoints and the message-oriented breakpoints we proposed in the breakpoint catalog of Section 5.1.1. Appendix D.1.2 details the implementation classes. We classify the breakpoints into two *categories* according to their implementation:

Breakpoints managed directly by Truffle Debug API Those breakpoints correspond to types of breakpoints which can be implemented with only the information of syntactic tags (e.g., to set breakpoints on expressions) and by specifying conditions (e.g., to evaluate if the current node is an instance of `ReceivedRootNode`). These breakpoints are created with the breakpoint builder provided by the `Breakpoint`[2] class in the Truffle Debug API. Then we just need to install them in the Truffle debugging session. Line breakpoints and simple section breakpoints are instances of `Breakpoint` and are created through this strategy.

Breakpoints managed by SOMns implementation Those breakpoints correspond to more complex breakpoints which require executing a stepping strategy to halt in an AST node besides a flag on the message. They are more complex section breakpoints which declare *wrapper nodes* in the AST to control each breakpoint's state, i.e., when is enabled or disabled. These breakpoints are instances of the `AbstractBreakpointNode` class (see Figure D.2), which extends from the Truffle `Node`[3] class, and it controls the state of breakpoints. These breakpoints are saved based on their source coordinate and their type.

Table 6.1 summarizes the implementation strategy for each of the proposed message-oriented breakpoints. Column *name* corresponds to the breakpoint name, as shown in Table 5.1. Column *category* refers to the category in which we consider the breakpoint according to its implementation, i.e., using Truffle `Breakpoint` class or dedicated wrapper nodes. Column *strategy* describes the halting strategy we implemented for each breakpoint. Column *syntactic tag* shows the tag name that the node to halt needs to have annotated. Column *node* indicates the node where the halt should occur.

To define breakpoints in the frontend, the backend provides syntactic tags for identifying syntactic elements of the language as part of the Kómpos protocol. As explained in Section 5.3, use the notion of PSI elements from IntelliJ to identify syntactic elements of the language in the frontend, e.g., the asynchronous operators and primitives that return promises. Once we have the source section information for the breakpoint definition, we

[2]https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/debug/Breakpoint.html

[3]Abstract base class for all Truffle nodes, `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/Node.html`

send it to the backend (see Section 6.2). In what follows, we describe the implementation strategies for the each of the breakpoints of the catalog based on the breakpoint categories.

Name	Category	Strategy	Syntactic tag	Node
message sender	truffle-based	before execution of expression nodes	`Expression Breakpoint`	`SendNode`
asynchronous before	truffle-based	before the execution of root nodes which are instances of `ReceivedRootNode`	`RootTag`	`ReceivedRootNode`
asynchronous after	truffle-based	after the execution of root nodes instance of `ReceivedRootNode`	`RootTag`	`ReceivedRootNode`
message receiver	wrapper nodes	before the execution of the next root node + flag on message	`RootTag`	`ReceivedRootNode`
message promise resolver	wrapper nodes	after the execution of the next root node + flag on message	`RootTag`	`ReceivedRootNode`
message promise resolution	wrapper nodes	before the execution of the next root node + flag on promise + flag on message	`RootTag`	`ReceivedRootNode`

Table 6.1: Implementation strategies for message-oriented breakpoints.

6.1.1.1 Breakpoints Managed Directly by Truffle Debug API

As can be seen in Table 6.1 we have three breakpoints in the category of breakpoints managed directly by the Truffle Debug API. In general, the implementation strategy consists of specifying the syntactic tag that marks the node in which we want to halt, and the suspended position, i.e., before or after the node execution. In the following we describe the implementation for each breakpoint in the Truffle-based category[4].

- Message sender: This breakpoint will halt *before* the execution of the node annotated with the tag `ExpressionBreakpoint` [5] for the given source section. This

[4]The classes `ReceivedRootNode`, `EventualMessage`, `EventualSendNode`, `SPromise`, `RegisterOnPromiseNode`, `PromisePrims`, `ResolveNode`, `AbstractPromiseResolutionNode` are available at `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/src/som/interpreter/actors/`

[5]SOMns implements different syntactic tags in `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/src/tools/concurrency/Tags.java`

tag marks an expression that can be the target of a breakpoint. In SOMns a node annotated with this tag is the asynchronous send operator `<-:` which is an instance of the `SendNode` (see step 1 and 1a in Figure 4.6).

- Asynchronous before: The debugger will pause the execution *before* the node annotated with the tag `RootTag` in the given source section. This tag marks program locations as the root of a function, method, or closure. Besides, a condition is declared to validate that the current evaluated root node is an instance of `ReceivedRootNode`, which represents the receiver node of the asynchronous message send (see step 3 in Figure 4.7).
- Asynchronous after: The debugger will pause the execution *after* the node executed with tag `RootTag` in the selected source section. The difference with the asynchronous before breakpoint is that we specified here as suspended position *after execution*, instead of the *before execution* position. This breakpoint reuses the same condition that the previous breakpoint to guarantee that the halted node corresponds to an asynchronously sent message.

6.1.1.2 Breakpoints managed by SOMns implementation (with wrapper nodes)

The last three breakpoints in Table 6.1 were implemented using wrapper nodes. Their implementation strategy can be generalized in the following steps:

1. When a message is created in the interpreter we ask the wrapper node (i.e., an instance of `AbstractBreakpointNode` class) declared as child of the `SendNode` if the *breakpoint state* is enabled or not.
2. With the breakpoint state we update the corresponding *flag* on the message (i.e., on `EventualMessage` or on `PromiseMessage`).
3. When the actor is about to process the message (i.e., `TracingActor` see Section 6.1.3) we checked the flags on the message and then instruct the debugger (i.e., `WebDebugger`) to proceed to execute the breakpoint.

Here we describe the detailed implementation for each breakpoint in the wrapper nodes category.

- Message receiver: For this breakpoint we use the *step next* stepping strategy, that we implement in the `SteppingStrategy`[6] class of Truffle Debug API, which consists in halting *before* the execution of the *next root node* annotated with the `RootTag` tag. The `EventualMessage` class contains a flag named `haltOnReceive`, which is enabled to true when this breakpoint is set in the asynchronous operator. In

[6]`https://github.com/ctrlpz/truffle/blob/debugger/step-end-turn/truffle/src/com.oracle.truffle.api.debug/src/com/oracle/truffle/api/debug/SteppingStrategy.java`

the implementation of the class `SendNode` corresponding to the operator, we have declared a child node of type `AbstractBreakpointNode` which handles the state of enabled or disabled. When an actor is about to process a new message from the mailbox, it checks if the message has the `haltOnReceive` flag enabled. The halting position in the AST is equivalent to the asynchronous before breakpoint (see step 3 in Figure 4.7).

- Message promise resolver: For this breakpoint we use the same stepping strategy as the previous breakpoint, but specifying that we want to halt *after*, instead of before the next root node is executed. The `EventualMessage` class contains a flag named `haltOnResolver`, which is enabled to true when this breakpoint is defined in the asynchronous operator. Like the previous breakpoint, we declared a child node of type `AbstractBreakpointNode` in the message which handles the state of enable or disable when the breakpoint has been defined in the asynchronous send operator. The flag is checked in the `ReceivedRootNode` class, when this node is about to execute the body of the method (see step 3 in Figure 4.7). There we communicate to the debugger that should do the halt after the execution of the next root node. The halting position in the AST is equivalent to the asynchronous after breakpoint.
- Message promise resolution: This breakpoint reuses the implementation for the message receiver breakpoint. In the class corresponding to the promise object `SPromise` we declared a flag `haltOnResolution`. This flag is verified when scheduling the callbacks for the promise, just after the promise has been resolved, i.e., in `RegisterOnPromiseNode` node (see step 1b in Figure 4.6). If it is enabled, then a message receiver breakpoint is activated enabling the `haltOnReceive` flag in the message. This message corresponds to the `PromiseMessage` [7], and it will halt before it is processed by its receiver actor, which is the sender actor of the original message that was sent.

As explained in Section 4.4.2 the SOMNS language considers that promises can be resolved in three ways: *normal resolution, null resolution and chained resolution.* So far we described message promise resolution breakpoints for *normal resolution* on the asynchronous send operator. We now detail how we handle the other two cases.

- If the promise has a *null resolution*, the promise has been optimized out because the return value is not used. This means that there is no value for its resolution, and then a promise resolution breakpoint cannot be defined at the frontend. However, a promise resolver breakpoint can be defined for this case.
- If the promise that has the promise resolution breakpoint is *chained* to another promise, the flag `haltOnResolution` is enabled for that other promise, i.e., the

[7]The promise message can be a `PromiseSendMessage` or a `PromiseCallbackMessage` (see Section 4.4.2).

promise to be resolved. This is implemented in `AbstractPromiseResolutionNode` class (see step 4 in Figure 4.7, instead of `resolvePromise()` method is called `chainedPromise()`). A promise resolver breakpoint will halt execution after the next executed root node.

We now turn our attention to how to deal with resolved and unresolved promises. If the promise is *unresolved*, the message sent is registered in the promise, and after it is scheduled in the actor. If the promise is *resolved*, the message is directly scheduled in the actor. For triggering the breakpoint in this case, it was implemented in the `RegisterOnPromiseNode` class, updating the flag of the message receiver breakpoint in the message `PromiseSendMessage` just before it is scheduled in the actor.

We now describe particular considerations to implement breakpoints on explicit promises and on promises returned with the message `whenResolved` message.

Considerations for explicit promises For *explicit promises* created with `createPromisePair` message we did a similar implementation as for the asynchronous send operator. The class `CreatePromisePairPrim`[8] contains two child nodes of type `AbstractBreakpointNode` to handle the activation, one for the resolver breakpoint and one for the resolution breakpoint. The class `ResolveNode` implements the wrapper node for the case when the promise is resolved with a non-promise value. Here we distinguish the explicit promises to ask directly to the promise if a promise resolver or a promise resolution breakpoint was set. For triggering the promise resolver breakpoint a wrapper node is executed (`haltNode`). This one will do the suspension just before the `resolve:` primitive is executed. The value returned by the wrapper node for the promise resolution breakpoint, will be propagated when registering the callbacks for the promise, in the `SPromise` class, where its value is verified, i.e., `haltOnResolution` flag. If that flag is true, then the interpreter will enable a message receiver breakpoint to halt before the callbacks are executed. This means that if a promise has registered more than one callback (i.e., an asynchronous message send to a promise or more than one `whenResolved` message), a promise resolution breakpoint pauses the program's execution before each of the callbacks is executed.

Considerations for promises returned with `whenResolved` Promises returned with the message `whenResolved` follow a similar implementation as for `createPromisePair` message. `WhenResolvedPrim`[9] class is similar to `CreatePromisePairPrim` class because contains wrapper nodes for the promises breakpoints, which in this case can be defined on the promise returned by the `whenResolved`. When the promise is created, the wrapper

[8]`https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/src/som/primitives/actors/PromisePrims.java`

[9]Implementation of classes `WhenResolvedPrim`, `WhenResolvedOnErrorPrim`, `OnErrorPrim` are available at `https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/src/som/primitives/actors/PromisePrims.java`

nodes update the flags in the `SPromise` class, and then the message is scheduled in the actor. A similar implementation was done for the messages `whenResolvedOnError` and `onError`. We declared wrapper nodes for the promises breakpoints in the classes `WhenResolvedOnErrorPrim` and `OnErrorPrim`, respectively.

6.1.2 Message Stepping

For each stepping operation received in the debugger backend, we implemented a stepping strategy (see `SteppingType` class Figure D.2). This strategy consists of the actions of the sender and the receiver actor of the suspended message for the stepping operations depicted in Table 5.5.

As we described in Section 5.1.2 stepping operations follow the program's execution between various breakpoints. Then, stepping operations in the category of messages and promises have the same semantics of their corresponding breakpoint and thus can reuse their implementation directly. However, this is not the case for stepping operations that goes beyond the boundaries of the current turn, i.e., that halt the actor's execution in a different turn. Even more, when the program's execution is suspended inside a turn, if we would like to resume until the end of that turn, we need to consider different cases of implementation to halt in that location. Thus, we distinguish two categories for the stepping according to the place where the stepping strategy for the current actor is implemented in the backend:

Stepping operations managed directly by a breakpoint This case is applied to stepping operations defined for messages and promises. Stepping operations in this category handle the halting of the actor reusing the wrapper node of its corresponding breakpoint.

Stepping operations managed in the `Suspension` class This case is applied to stepping operations in the category of turns. In this case of enabling the halting in the desired turn (also using the wrapper node of the corresponding breakpoint), we need additional flags and check the received stepping strategy in the `Suspension` class.

In Table 6.2 we summarize the stepping strategies we followed for the implementation of each stepping operation. Column *name* refers to the name of the stepping operation (as designed in Table 5.5). Column *category* refers to our categorization according to the implementation for the stepping operations. Column *strategy* describes the halting strategy we implemented for that stepping command. Finally, column *breakpoint*, relates a stepping operation with its equivalent breakpoint(s) regarding its halting semantics.

Name	Category	Strategy	Breakpoint
step to message receiver	managed directly by wrapper node	before the execution of the next root node + flag on message	message receiver
step to promise resolver	managed directly by wrapper node	after the execution of the next root node + flag on message	promise resolver
step promise resolution	managed directly by wrapper node	before the execution of the next root node + flag on promise + flag on message	promise resolution
return from turn to promise resolution	managed in `Suspension` class	before the execution of the next root node + flag on promise + flag on message	promise resolution
step next turn	managed in `Suspension` class	before the execution of the next root node + flag in the actor	message receiver
step end turn	managed in `Suspension` class	after the execution of the next root node in that turn	asynchronous after or promise resolver

Table 6.2: Implementation strategies for stepping operations.

6.1.2.1 Stepping Operations Managed Directly by a Breakpoint

As we can observe in Table 6.2 the three first stepping operations are grouped in the category of managed directly by a wrapper node. The general stepping strategy consists of the following steps:

1. When a stepping command is received in the debugger backend (i.e., through a `StepMessage` of the Kómpos protocol), the thread for the current actor is updated with the selected stepping strategy.

2. Afterwards, the debugger backend resumes execution.

3. The wrapper node of the equivalent breakpoint is updated with the stepping strategy information of the current actor, in the `BreakpointNode` class. If the corresponding breakpoint has the stepping strategy enable, i.e., due to a stepping operation received from the debugger frontend, the breakpoint is enabled, otherwise is disabled (unless an explicit breakpoint has been set previously).

Following the described strategy, when a *step to message receiver* is defined in the frontend, a message receiver breakpoint is enabled for the selected message. Similarly, a promise resolver breakpoint is enabled for a *step to promise resolver*, and a promise resolution breakpoint for a *step to promise resolution.*

6.1.2.2 Stepping Operations Managed in the `Suspension` class

The general strategy for the last three stepping operations of Table 6.2 adds a new step in the strategy described for the stepping operations managed directly by a breakpoint. More concretely, between step 2 and step 3, once the debugger backend resumes execution, we check in the `Suspension` class the next task received from the debugger frontend. We have different implementation strategies if the stepping operation has been defined on the level of the current (or a different) turn. In what follows, we describe each case in detail.

- **Return from turn to future resolution**: when this stepping command is requested to the backend, we need to enable the flag `haltOnResolution` for the promise corresponding to the current processing turn, i.e., for the promise returned by the message of the current turn [10]. Consequently, when the actor processes the message corresponding to the callback of the promise of the turn, it will check the flag `haltOnResolution` of the promise, i.e., in `TracingActors` class. Then, if true, the debugger will halt the program's execution before executing the next root node, i.e., the root node corresponding to the callback of that promise.
- **Step to next turn**: when this stepping command is enabled, the value for the `stepToNextTurn` flag declared in the `TracingActors` class is updated to true. In this case, if the flag is true, we invoke the stepping until next root node strategy, similarly as for a message receiver breakpoint. Then, the debugger will halt the program's execution before executing the next root node, i.e., before the execution of the next message in the actor's mailbox.
- **Step to end turn**: if the actor suspended has defined this stepping strategy, we need the information of the current executing turn, the node suspended, and the stack frames corresponding to the suspension. This information is needed for the different cases in which this stepping command can be defined, for example:
    1. at the beginning of a turn
    2. from a node declared inside a turn
    3. from a synchronous call
    4. from inside a block

[10] A promise resolution breakpoint is different because it halts at the resolution of a promise that belongs to a message that is sent inside a turn

In the implementation, using the mentioned information, the debugger runs until the next root node and checks if the source section corresponding to that node matches the source section of the turn where the suspended node belongs. If that is the case, then the debugger will halt execution after the root node is executed, i.e., before returning the result value [11].

Table 6.3 shows code examples for each variant where the step to end turn can be defined. For the example scenario of *beginning of a turn* we consider that the `InstantMessenger` actor is first paused due to a message receiver breakpoint in Line 44 in Appendix C.2. Then the stepping end to turn is executed. For the case of a *node declared inside a turn*, we start from the previous scenario with message receiver breakpoint in Line 44, and we define a step over command. Then the stepping end to turn is executed. We consider a program paused in a *a synchronous call*, for example, due to a line breakpoint in Line 53, which is called synchronously in Line 85, then we step end turn. Also, we can step end turn if the program is paused *inside a block*, for example, due to a promise resolution breakpoint in Line 60.

Step to end turn from	Halt before step	Halt after step	
		Sender	Receiver
beginning of a turn	ln: 82, cn: 7, cl: 131	Platform resume	InstantMessenger ln: 87, cn: 11, cl: 131
a node declared inside a turn	ln: 82, cn: 7, cl: 29	Platform resume	InstantMessenger ln: 87, cn: 11, cl: 131
a synchronous call	ln: 53, cn: 34, cl: 7	Platform resume	InstantMessenger ln: 87, cn: 11, cl: 131
inside a block	ln: 62, cn: 9, cl: 303	Platform resume	InstantMessenger ln: 70, cn: 45, cl: 303

Table 6.3: Step end turn examples from Appendix C.2.

From the described strategies in Table 6.2 we can observe that a *return from turn to promise resolution* is similar to define a promise resolution breakpoint, but for a different turn. The semantics for the *step next turn* operation is similar to the message receiver breakpoint, but also for a different turn. Finally, in the case of a *step end turn* the halting semantics is similar to a promise resolver or an asynchronous after breakpoint, but the implementation strategy is different in this case.

In short, we summarize the five stepping strategies for implementing breakpoint and stepping:

- halting before the execution of the next root node
- halting before the execution of the next root node with a condition

[11]The implementation of the stepping strategy for this command can be found in `https://github.com/ctrlpz/truffle/blob/debugger/step-end-turn/truffle/src/com.oracle.truffle.api.debug/src/com/oracle/truffle/api/debug/SteppingStrategy.java`

- halting after the execution of the next root node
- halting before the execution of expression nodes
- halting after the execution of the next root node in a turn

6.1.3 Trace-based Visualizations

As we mentioned in Section 5.1.3, we use the trace information from the Kómpos protocol (see Figure 5.14) to visualize the actor state and message causality information in Apgar frontend (see Section 6.2).

The trace events are encoded in binary format using thread-local buffers [LBM15] and considers actors being executed in different threads [AMB$^+$18]. The recording approach uses subtraces to record the different events per actor. New subtraces are started when an actor starts executing or when a buffer becomes full. Subtraces are ordered by a `bufferId` and `actorId`, because events of one actor can be recorded by different threads. When buffers are full, they are swapped to minimize runtime overhead. In Medeor, the trace events are written in buffers and afterward to a file. In this work, we do not parse the file, but we read the information directly from the buffers, which are sent to the frontend.

`KomposTrace` is the implementation class in Medeor that records the trace events, i.e., the start of subtraces, the creation, and completion of actors, the start and end of dynamic scopes (i.e., turns), the creation of passive entities (i.e., messages and promises), sending operations (e.g., a message send and a resolution of a promise). As we mentioned in Section 5.3, we added in this class support for recording the received messages by each actor in order, which was not present in the original version of the Kómpos protocol. In particular, we use the notion of messages received by the actor to guarantee that the order of messages visualized in the mailbox corresponds to the messages that will be executed by the actor, i.e., messages are shown in order of arrival. To this end, we record message reception at two points in the `Actor` class in the debugger backend, i.e., when the actor appends the received message in its mailbox, and when the actor is about to process the messages of its mailbox. We need both recordings because the actor can be paused and still receive messages.

Appendix D.1.3 describes the implementation of the trace events in the SOMns interpreter needed to implement the visualizations for the actor state and message causality at the frontend. In the next section, we focus our attention on the strategy to implement asynchronous stack traces.

6.1.4 Asynchronous Stack Trace

Thanks to the Truffle Debug API, SOMns features a traditional synchronous stack, which consists of frames returned by the method `SuspendedEvent.getStackFrames()`.

The `SuspendedEvent` class is located in the Truffle Debug API, and it describes the suspended thread's location in the guest language code.

We extended the sequential stack with information needed to trace asynchronous interactions. The implementation we present here[12] builds on an initial version of an asynchronous stack implemented by Dr. Clement Bera[13]. Appendix D.1.4 shows the main classes related to asynchronous stack trace support. An asynchronous stack consists of a list of frames of different types:

1. an entry created when an asynchronous message is sent (`EntryAtMessageSend` class).

2. an entry created when a promise is resolved (`EntryForPromiseResolution` class).

3. an entry for method activations (`ShadowStackEntry` class).

The overall idea is to traverse the runtime stack and manage the different kinds of entries. If the stack trace frame corresponds to an asynchronous message send or an entry related to promise resolution we employ the first frame and then rely on the shadow stack to get the next stack entries.

To get the control flow frames, we followed the implementation strategy of instrumenting the place where an asynchronous message is sent `<-:`, in the `EventualSendNode` class. Specifically, before returning the arguments for the node execution, in the `InternalObjectArrayNode` class.

To get the data flow frames, we consider instrument eight different places in the interpreter. Table 6.4 shows a summary of the class locations in the interpreter that we need to instrument to get all the entries for the asynchronous stack. Because promise callbacks have different stack traces, due to the asynchronous nature of the program, we need to causally related these traces. This implementation is located in the `EventualMessage` class. Specifically, we saved the entries for the promise resolution in two places:

- at the resolution of a promise to which an eventual message was sent, in the `AbstractPromiseSendMessage` class. There, we concatenated the entries of the promise that has been resolved, with the entry of the message sent to this promise, i.e., which is a callback of the promise.

- at the resolution of a promise, that has a callback registered with the `whenResolved` message, in the class `AbstractPromiseCallbackMessage`. There, we concatenated the promise resolution stack entries corresponding to the promise to which this callback is registered on.

[12]`https://github.com/ctrlpz/SOMns/blob/somns-intellij-4.5/src/tools/debugger/asyncstacktraces/`

[13]`https://github.com/clementbera/SOMns/tree/AsyncStackTraceStructureFromDev`

Entry type class	Node class	Description
`EntryAtMessageSend`	`InternalObjectArrayNode`	entry for asynchronous message send
`EntryForPromiseResolution`	`ResolvedNode`	resolved with a value
	`ErrorNode`	resolved with an error
	`SResolver`	resolved with a promise
	`ReceivedCallback`	on `whenResolved` block
	`RegisterWhenResolved`	on `whenResolved` message
	`RegisterOnError`	on `whenResolved` error message
	`ReceivedMessage`	on receive message
	`SchedulePromiseHandlerNode`	on schedule promise
`ShadowStackEntry`	`IntToByDoMessageNode`	loop message `to:do:`
	`DoPrim`	array message `do:`
	`DoIndexesPrim`	array message `doIndexes:`
	`BlockPrims`	block closures
	`TimerPrim`	message `actorDo:after:`
	`CreatePromisePairPrim`	message `createPromisePair`
	`ExceptionsPrims`	message `exceptionDo:` `catch:onException:`
	`AbstractGenericDispatchNode`	method activations

Table 6.4: Main nodes instrumented in SOMNS to create entries for the asynchronous stack trace.

6.2 Apgar Frontend, an IntelliJ Plugin

In this section, we will describe the main elements of the debugger frontend implementation and its interaction with Medeor, the debugging support in the backend.

6.2.1 A Custom Language Support Plugin

The debugger frontend is implemented in a plugin for SOMNS[14] in the IntelliJ IDE. The *IntelliJ Platform* provides APIs to build common IDE functionality, such as a project model and a build system. This platform also provides an infrastructure for debugging with language-agnostic advanced breakpoint support, call stacks, watch windows, and expression evaluation. Appendix D.2 shows the main classes of the plugin implementation. We used the program structure interface (PSI) support that the IntelliJ platform provides to parse the SOMNS program files. We have implemented a grammar in BNF language that supports SOMNS classes, methods, and expressions declarations. The following sections explain how we use the data for showing actor state and message causality. Appendix D.3 shows the three main interactions between backend and frontend by means of sequence diagrams.

[14]Our plugin for SOMNS language is publicly available at `https://plugins.jetbrains.com/plugin/13213-somns`.

6.2.1.1 Actor State Inspection

As we can observe in Figure D.4 the `TraceParser` is the main class that handles the trace data that is sent from the debugger backend to the frontend. After parsing the trace data[15] that the frontend receives from the backend, it updates the actors' view and mailbox with new data.

The *actors view* loads all the actors created by the program, i.e., parsing the `Activity-Creation` event. The view updates the state of running or paused for an actor using the debugger messages from the Kómpos protocol, i.e., when the frontend receives a `StoppedMessage` from the backend. This is the case when an actor is paused explicitly in the frontend (with a button) or when an actor is paused due to a breakpoint or a stepping operation.

In the *mailbox view* the frontend shows the current processing message of the paused actor and other messages that the actor has received. For this, the frontend needs to parse the different events generated and sent for the backend (see Section 6.1.3). In particular the frontend uses data parsed from the events `SendOperation`, `MessageReception`, `PassiveEntityCreation` and `ScopeStart`. In the next section, we explain how the frontend obtains the turn information from the trace data.

From the `StackTraceResponse` debugger message the frontend obtains a message identifier that is needed to visualize the current processing message in the mailbox of the paused actor. The mailbox distinguishes messages sent to far references and sent to promises. The messages shown in the mailbox correspond to messages not yet executed by the actor, then our implementation class of the mailbox view searches in the trace data for messages which receiver is the paused actor. If the message has been sent to a far reference, the mailbox class in the frontend asks the identifier of the actor to the `SendOperation` directly, i.e., the `targetId`. If the message has been sent to a promise, we do not have the information of the target actor that will resolve the promise until it is resolved, then the frontend needs to wait for the send operation corresponding to that promise resolution. Therefore, the mailbox class in the frontend asks the creation activity of the resolution, which indicates the actor that resolved the promise[16] and the mailbox class checks if this actor matches with the pausing actor. For example in Listing 6.1, Line 8 the eventual message `computePerimeter` is sent to `c`, which is a promise for a new instance of a `Calculator` actor. This way when the frontend visualizes in the mailbox the `computePerimeter` message the mailbox view can show its target promise (see Figure 5.8).

[15]By parsing the trace we refer to parse the trace data, i.e., a `ByteBuffer`, we receive from the backend using the Kómpos protocol.

[16]A promise is resolved with a value, an error or can be chained to another promise, that can be seen in the properties state and value of the promise in the mailbox.

```
    calculators doIndexes: [:i |
      | c |
      c:: (actors createActorFromValue: Calculator) <-: new: i math: math.
      calculators at: i put: c
    ].

    calculators do: [:c |
      c <-: computePerimeter whenResolved:[:p |
         counter:: counter + 1.
         counter = numberStudents
           ifTrue: [
             completionPP resolve: true.
           ].
      ].
    ].
```

Listing 6.1: Code snippet that shows a message sent to a promise (from Appendix C.3).

6.2.1.2 Message Causality

To obtain the happened-before relationship between messages, the frontend needs information about the turn in which a message was sent. The frontend obtains this information parsing the trace events, in particular, from the `ScopeStart` event. However, we have one problem we needed to consider in the implementation.

When the debugger frontend requests trace data to the debugger backend, the buffers where the trace events are saved are forcibly swapped, and this can happen in the middle of a turn, which can cause that the remaining events of a turn are recorded in a different buffer. For example, in Listing 6.2 we can see a representation of events for the same actor's turn recorded in different buffers.

```
[Thread 2][Buffer 0][Activity 1][Turn 1][turn 1 events] ...
[Thread 2][Buffer 1][Activity 2][Turn 3][Activity 1] [remaining turn 1
  events]
```

Listing 6.2: Representation of events for the same actor's turn recorded in different buffers.

The three dots (...) means the buffers have been swapped when executing the events of `Turn 1` of `Activity 1`. We expect the threads where the buffers are executed are the same, because actors cannot switch threads in the middle of a turn, i.e., the remaining events of `Turn 1` should be on the same thread (`Thread 2`)[17].

[17]We do not consider using the thread ids for our solution to the problem because we have observed trace outputs were the threads ids are different for the previous example. Further research needs to be conducted here to find the cause of when the threads are blocked, the thread pool does not behave as expected.

In short, we can observe in the Kómpos trace the following problem: when recording the actor's events in the trace, it can happen that a turn may be open in one buffer, and its events are executed in another buffer. Besides, together with the swapping problem, it can happen that buffers may be written out of chronological order from the perspective of an actor because an actor can be executed on different threads over time [AMB+18].

To solve the problem related to the turn's events divided between different buffers, we implemented the trace parser in the frontend to use the buffer identifiers to *reorder* the events and get the correct turn identifier, i.e., the id of the causal message for the current processing events. This message id is declared in the Kómpos protocol as `scopeId` of the `ScopeStart` event, which is saved in the frontend in the variable `currentTurn` of the `TraceParser` class. The events an actor processes in a turn are activity creations, send operations, message receptions, and passive entity creations.

Therefore, to get the correct turn data information for the recorded actor's events, we declared a sorted map in our `TraceParser` class, i.e., `eventsByActor` in Figure D.4, to keep all events ordered by `actorId` and `bufferId`. The sorted map data structure grows incrementally with each new information received from the backend. This data structure is initialized for each new debugging session. Every time an event is parsed from the trace, the trace parser saves the corresponding `actorId` and `bufferId`. This is possible because when the buffers are swapped, the current activity is recorded in the trace using a designed marker, i.e., the `ImplThreadCurrentActivity` event in figure 5.14.

Then, after parsing all the events of the trace received in the frontend, the trace parser resolves the turns for each event. There can be multiple turns opened in one buffer, but only one turn open by actor because actors processed one message at a time. We save the open turns by actor in a map, every time an event `ScopeStart` is received. Then, when an event such as `ActivityCreation`, `PassiveEntityCreation`, `MessageReception` and `SendOperation` are received, the frontend queries the map of open turns, to get the turn where this event was created, i.e., the *causal turn*. When the trace parser parses the event `ScopeEnd`, we remove the turn from the map of open turns, for the current `bufferId`.

6.2.1.3 Asynchronous Stack Trace

As mentioned in Section 6.1.4 in Medeor we implemented support for asynchronous stack traces. Our debugger frontend receives the stack trace information through the debugger message `StackTraceResponse` of the Kómpos protocol. There, we added the flag `asyncStack`, which indicates that the data received corresponds to the asynchronous stack instead of the traditional stack. With the flag information, the frontend updates the rendering our the stack trace that is visualized in the *Frames view* and the available variables in the *Variables view*. We build our stack trace in the frontend thanks to the classes provided by IntelliJ Debug API, e.g., `XExecutionStack` and `XStackFrame`.

6.3 Conclusion

In this chapter, we described the implementation of online debugging techniques for the SOMns language. We have added support for debugging concurrent concepts such as actors, promises, messages, and turns by implementing several breakpoints and stepping operations on top of the Truffle instrumentation and debugger API. The strategy of instrumenting nodes of the AST allows define breakpoints on the node instead of the line numbers, as usually done by traditional online debuggers. The combination of sequential and asynchronous stepping enables developers to inspect the actor messages at different interesting locations, which has not been available until this work.

Besides, extending the Kómpos trace implementation allows us to include data events relevant to enable actor state inspection and show message causality information in our tool. Moreover, the asynchronous stack trace's work can enable developers to see not only method activations but to visualize entry points for asynchronous message send and promise resolution.

At the frontend side, the IntelliJ platform offers flexible APIs to integrate custom language functionalities in a plugin. The interaction between the frontend and the backend is made straightforward thanks to the Kómpos protocol, which provides the necessary debugging messages and events for visualization. We think the implementation of an online debugger presented here can be used as guidance to provide advanced tool support for other actor languages. Our next step is to evaluate the online debugging features proposed in this dissertation in an experimental study.

Chapter 7

Evaluation of Online Debugging Techniques for Actor-based Programs

In the literature of debugging techniques, two common approaches have been employed to validate debugging operations: *user studies* and *formal specifications*. User studies [SM16, GBNDM14, PSTH16, KM08, LSX$^+$17] mainly focus on evaluating usability and effectiveness of the approach, e.g., giving participants debugging assignments and measure their success, time, and perception of the tool. On the other hand, formalization of debugging techniques has been used to prove correctness when identifying problematic states in a program [LC06] and to analyze issues in concurrent programs [LLL14, CMMRT18, VMV17].

In this dissertation, we employed both approaches. In particular, in this chapter, we describe a user study we conducted to evaluate the online debugging techniques we proposed for actor-based programs and implemented for the SOMns language. The study aims to know if the debugging techniques help identify the root cause of concurrency bugs and understand an actor-based program behavior. In the next chapter, we make use of formal specifications.

We now describe the user study conducted with Apgar based on a *mixed methods research design* approach [CC17]. We will conduct quantitative (QUAN) and qualitative (qual) studies sequentially, first the quantitative and later the qualitative. This is called a QUAN $\rightarrow$ qual design [CJT15]. Subsequently, we explain our findings and threats to validity in our experiment.

7.1 Design of the User Study

Our goal is to study the effect that has on developers the use of the advanced debugging techniques we proposed in Chapter 5. This section explains our research design selection and how we apply it to the proposed experimental conditions. Besides, we mention the main elements we consider to formulate our research hypothesis about the proposed online debugging techniques for actor-based programs, i.e., research problem and variables to analyze.

7.1.1 A Mixed Methods Experimental Research Design

Our experimental study measures as independent variable our *novel debugging techniques*, and as a dependent variable, the *time* developers spend solving the debugging assignments. To this end, we use a **mixed methods experimental design** [CC17]. First, we will conduct a **quantitative study** to measure the time taken by participants when solving debugging assignments in SOMns. Then, we will conduct a **qualitative study** to measure developers' perception of our debugging techniques. This design allows researchers to embedded qualitative data in a quantitative experiment, before, during, or after the experiment. We now detail the designs we used for quantitative and qualitative studies.

A quantitative research study is based on collecting numerical data to answer a research question [CJT15]. A quantitative research study can be classified as experimental and nonexperimental. Experimental research is the best type of quantitative research for demonstrating *cause-and-effect relationships*, where the researcher can actively manipulate the independent variable (IV). For example, in the relationship between debugging techniques and identifying the root cause of concurrency bugs, the IV is the *debugging techniques*, and the dependent variable (DV) is *identifying the root cause of concurrency bugs*. The levels of manipulation of IV have been referred to as *treatment or experimental conditions*. For instance, one level of manipulation of the IV in our example is the use of advanced debugging techniques, whereas another level is the control condition of traditional debugging techniques. In nonexperimental quantitative research, the researcher is not able to manipulate the independent variable. In this work, we will conduct an **experimental quantitative study**.

Christensen et al. mention three groups of experimental research design: *Weak, Quasi, and Strong* [CJT15]. We adopt a **strong experimental design** because they have greater internal validity than *weak or quasi-experiments*, this means that the effect of the independent variable on the dependent variable can be isolated and tested. Besides, strong designs allow controlling all possible extraneous variables using random assignment of participants. Strong experimental designs can be further classified in four design categories [CJT15] [CS63] [SCC01]:

- Between participants (posttest-only control group design): this research design consists of participants randomly assigned to two groups. The groups are exposed to different levels of the independent variable, and afterward, a posttest is administered. A control group is a research group in which participants do not receive the active level of the independent variable. A treatment group is a research group in which participants receive some level of the independent variable that is intended to produce an effect. Hence, this design addresses two experimental conditions at the same time, but each group performs only one experimental condition.

- Within participants (within posttest only design): this research design uses one group in which all participants are exposed to each experimental condition. After each experimental condition is administered, a posttest is applied. Participants need to do all experiments sequentially.
- Mixed (pretest-posttest control group design): this research design is a combination of a between-subjects and within-subjects design, in which participants are randomly assigned to two groups, then a pretest is administered, and later the treatment conditions are executed. Finally, a posttest is administered. In literature about research methods [SCC01, CJT15], it has been recommended to include pretests to the basic randomized design.
- Factorial: this research design studies more than one independent variable. A factorial design can be based on each of the (three) other variants. This design requires a more significant number of research participants [1]. Consequently, it presents a greater difficulty to manipulate different independent variables.

In this work, we used a **between participants design** because it allows measuring the effect of having the advanced debugging techniques for actor-based programs compared to a control condition in which only traditional debugging techniques are provided to the participants. We do not include a pretest in our experiment because we can only quantify our dependent variable at posttest, i.e., the *time* in which participants solve the debugging assignments. In literature, other debugging studies have also used the same design [KM10, SM16, XBLL16, LSX$^+$17].

Thus, in our between participants design, we expose the participants to two *experimental conditions*:

1. Solve debugging assignments using traditional debugging techniques (i.e., line breakpoints, sequential stepping, variables state visualization).
2. Solve debugging assignments using the proposed debugging techniques for actor-based programs (i.e., message breakpoints, sequential and message stepping, visualization of causality relations, asynchronous stack, visualization of actor state and variables).

After the quantitative experiment, we will conduct a qualitative study. A qualitative research study collects some type of nonnumerical data to answer a research question [CJT15], e.g., statements or the observed behavior of a person during an interview. A

[1]E.g., for two independent variables. An arrangement of 4 cells is needed. If each cell requires 15 participants, a total of 60 participants is required.

qualitative research study is an interpretive research approach that relies on subjective data to investigate people in particular situations rather than manipulating independent variables. In our work we want to capture qualitative data after the treatment, i.e., after using the debugger, because it can help us explain if the treatment or intervention (i.e., the advanced debugging techniques) helped developers to find the root cause of concurrency bugs.

Figure 7.1 gives an overview of the workflow of the overall approach for our mixed methods experimental design. Before the experiment, we measure participants' concurrent programming experience, i.e., an expertise score, to create matched groups of participants, i.e., the control and treatment groups (1). Later, we create the groups using random assignment of the matched participants (2). Steps 3 and 5 represent the quantitative study, in which we measure the time participants spend solving the debugging assignments. We perform qualitative measures after the experiment (steps 4, 6, and 7) to explain and interpret the results of the experiment. In particular, we measure participants' perception of the debugging assignments and advanced debugging features they have seen during the experiment. This design keeps the quantitative and qualitative studies sequential, and their results separated.

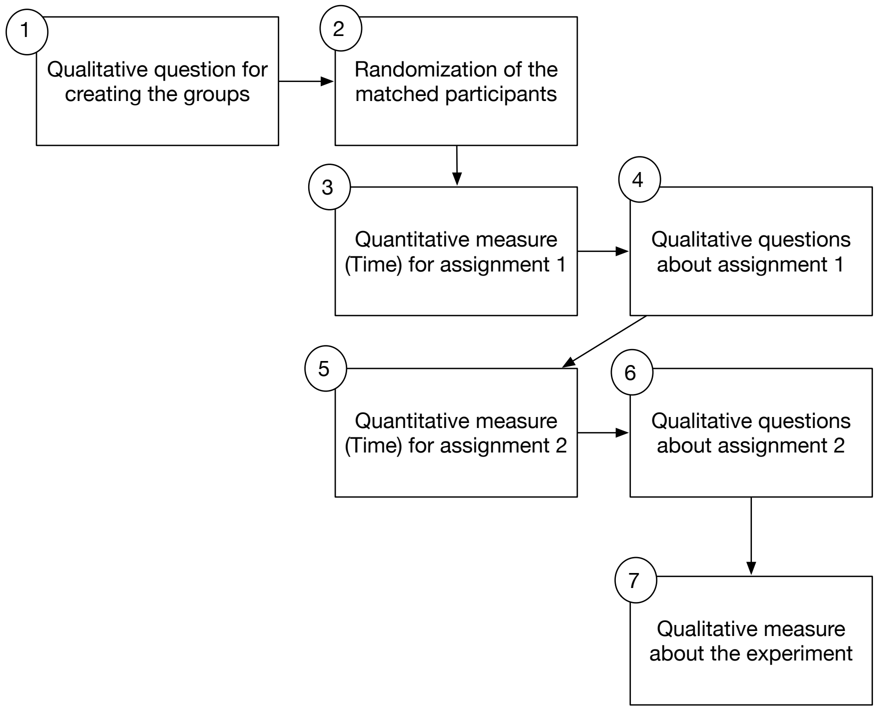

1
Qualitative question for creating the groups
2
Randomization of the matched participants
3
Quantitative measure (Time) for assignment 1
4
Qualitative questions about assignment 1
5
Quantitative measure (Time) for assignment 2
6
Qualitative questions about assignment 2
7
Qualitative measure about the experiment

Figure 7.1: Mixed methods experimental research design used in our user study.

Finally, we detail the research hypothesis (i.e., the best prediction or a tentative solution to a research problem [CJT15]) that our user study aims to investigate. First, we declare the **research problem** that motivates our study as:

Do advanced debugging techniques such as message-oriented breakpoints and rich stepping with visualizing the causality of messages help developers to identify the cause of complex concurrency bugs in actor-based programs?

We derived the IV *debugging techniques*, which represents the cause of the relationship from our research problem. The effect, which represents our DV, is *identifying the cause of concurrency bugs in actor-based programs.* And the treatment condition *advanced debugging techniques* we proposed. To measure the effect in our experiment, we declare as dependent variable the *debugging assignment completion time*, i.e., the time taken by participants to solve the debugging assignment. The complex concurrency bugs we refer here are the ones we classified in section 2.3.

We define now the **research hypothesis** we aim to investigate as the following:

Hypothesis 1. *Advanced debugging techniques such as message-oriented breakpoints and rich stepping with visualizing the causality of messages help developers to identify the cause of complex concurrency bugs in actor-based programs.*

From the research problem, we derived more hypothesizes to test the proposed advanced debugging features:

Hypothesis 2. *Message breakpoints and stepping operations help to reduce the effort when searching the root cause of concurrency bugs.*

Hypothesis 3. *The combination of sequential and message stepping is effective to inspect actor's turn.*

Hypothesis 4. *Visualization of message causality is useful for understanding the program while debugging.*

Hypothesis 5. *The asynchronous stack trace is useful for identifying message ordering problems.*

7.1.2 Experiment Planning

As mentioned before, in a between-participants design, participants are assigned to either a control or an experimental group. The control group used a traditional online debugger found in mainstream languages with state inspection features. On the other hand, the experimental group used Apgar including advanced debugging features for actor-based programs.

For both groups, we log all debugging operations participants used during the assignments, i.e., the breakpoints and the stepping operations. We obtained 28 participants that agreed to participate in the study. We conducted the experiment online, supported with Zoom and VirtualBox, because it was not possible to enable a computer room at the university with all participants due to COVID-19 measures.

Potential participants were asked to fill a time slot to randomize them into the two groups (see details in Section 7.2.1). A maximum of 6 participants was allowed to reserve a time slot. Participants received a day before the experiment an email with instructions to download and setup a VirtualBox[2] image with the corresponding version of the debugger, i.e., control or experimental group. Besides the VirtualBox instructions, we provided participants with a SOMNS cheat sheet[3] we prepared with a summary of the language syntax. Table 7.1 details all the activities made during the experiment.

Activity	Time (min)
(1) Read code of conduct	1
(2) Screen sharing of the SOMNS language tutorial	13
(3) Screen sharing of the Apgar debugger tutorial	EG: 9, CG: 3
(4) Debugging exercise in IntelliJ	5
(5) Debugging assignments	60 (max. 80)
(6) Questionnaire	5 (max. 10)
Total estimated	90 (max. 120)

Table 7.1: Experiment planning.

The first activity in the experiment was to read a code of conduct with some rules for the participants to agree (see Appendix E.1) (activity 1). We prepared two video tutorials, one about the SOMNS language and a second one about the debugging features that participants could find in their debugger version (activities 2 and 3). The experimental and control group had different videos about the debugging features because they were exposed to different experimental conditions, but both videos had a similar length. After viewing the video, we did a debugging exercise with the `PythagorasCalculator` program to illustrate participants the debugging features they have available in the debugger for solving the assignments (activity 4). Later, we gave participants the image location of the file with the assignments to read and perform the experiment (activity 5). Finally, when each participant completed the two assignments, we gave each participant a link to fill a questionnaire (activity 6).

We measure time only for the activity of the debugging assignments, i.e., activity 5. Except for activity 3, in which the experimental group's tutorial was a bit longer than the control group, both groups had the same timing duration. The experiment was estimated for 90 minutes approximately. We gave a margin of 120 minutes maximum for those cases where the number of participants was 5 or 6 because the interaction with

[2]https://www.virtualbox.org

[3]See Appendix B. The version to print is available at `https://github.com/ctrlpz/somns-sample-programs/tree/master/docs`.

the host took more time in those groups. However, the time limit for the debugging assignments was the same for all cases.

7.2 A Between-Subjects Research Design

In this section, we describe how we will apply a between-subjects research design to our study. First, we will explain how we apply the random assignment technique of participants for creating the groups. Second, we will show the debugging assignments used in the experiment. And finally, we describe the posttest we conducted to measure the variables of the study, i.e., the time and participants' perception about the debugger.

7.2.1 Random Assignment of Matched Participants

As we mentioned before, we define two groups of participants from two experimental conditions:

1. a control group, which will employ *traditional debugging techniques* to solve two debugging assignments.
2. a treatment group, which will use our *advanced debugging techniques* in Apgar.

Randomization or random assignment to groups has been defined as *"a probabilistic control technique, that equates experimental groups at the start of an experiment on all extraneous variables, both known or unknown"* [CJT15].

When the group of participants is less than 30, there is no complete assurance of the equivalence between the groups [CJT15]. Christensen et al. advise combining *matching* with *statistical control* along with the randomisation technique, e.g., match pairs of participants and then randomly assign them to treatment and control groups. One drawback here is that a survey needs to be done in advance (i.e., before the experiment) to measure the participants regarding the matching variable. Since we have 28 participants in our study, we follow Christensen's advice.

We defined the matching variable as the *programming experience* of participants in actor-based programs or concurrent programs in general. We ask each participant their experiences programming concurrent applications and programming concurrent applications using actors (e.g., how many years, which concurrency models or actor variants). With the participants' answers, we gave a score in the range of 0 to 3 to each participant. The 0 value represents a *null* programming experience in developing concurrent applications, 1 represents *beginner*, 2 represents *intermediate* and 3 *advanced.* We then create the groups as follow:

1. Order participants according to the matching variable. E.g., rank all participants from lowest to highest programming experience in developing concurrent programs.

2. Match participants for the matching variable. E.g., the two participants with the lowest score are the first set (two, because we have two treatment groups).

3. Assign randomly *each individual* of the set to one of the treatment groups. For the random assignment, we extended a table of random numbers from [CJT15] for our 28 participants. Appendix E.2 details the randomization steps based on a table of random numbers.

4. Repeat the previous step until the set of participants with the highest score is randomly assigned to the groups.

7.2.2 Debugging Assignments

Appendix E.3 shows the two debugging assignments used in the experiment. In assignment 1, we employ Acme Air booking system implementation in SOMns [AMB+18] where we injected a message order violation bug (see Section 2.3.2.1). In assignment 2, participants use an order purchase similar to the one found in literature [SCM09, GBNDM14], which contains a behavioral deadlock concurrency bug (see Section 2.3.1.2).

We gave the participants a description of the program with a conceptual actor diagram. Also, each assignment description contains an expected and observable output. We asked participants to find the root cause of the problem, but they do not need to fix the bug in the code.

During the debugging assignments, participants were not allowed to talk with other participants. However, they were allowed to ask the host[4] any question regarding the SOMns language or how to activate a debugger functionality.

7.2.3 Posttest Design

Our posttest design consists of two measurements. First, we measure the dependent variable of our quantitative study, i.e., the time participants spent to complete each debugging assignment. Later, when participants finished the second assignment, we measured their perception of the debugger in a questionnaire.

The participants notified each assignment *start time* to the host, and the *end time* we got it when the participant sent the line number of the fault to the host. We gave the participants a time limit of maximum 40 minutes for each assignment. Section 7.3.2 will discuss the results of the statistical test regarding the time values. Also, at the end of each assignment, the host asked each participant two questions *"How did you get to the fault?"* and *"Which debugging features you used?"*. This way, we could get additional information about how participants used the debugging features and whether they correctly understood the root cause of the program's bug.

[4]The host is the researcher conducting the experiment, i.e., Carmen.

At the end of the experiment, when each participant finish both assignments, we gave them a questionnaire, i.e., to collect qualitative information about the participants' profile and their perceptions of the debugger. Appendix E.4 shows the exact list included in the questionnaire, which consists of four kinds of questions:

- The first part of the questionnaire (q1 - q8) collects participants' programming experience information.
- The second part of the questionnaire (q9 - q14) collects participants' impressions and data regarding the experiments.
- We also added two questions to give additional comments about the debugger (q15) and the experiment (q16).
- Finally, participants were asked to upload a file generated by the debugger with a log of the operations they used during the experiment (q17).

For question 10, we chose to use a 5-point approval rating scale, aka 5-point Likert scale, because it allows us to observe two key dimensions of attitudes. This rate measures direction (positive or negative toward the attitudinal object) and strength or intensity of attitude [CJT15] regarding the debugging features participants used during the experiment. Section 7.3.3 will discuss the results for the qualitative study.

7.3 Results

In this section, we present and discuss the results of the experiment. We start by describing the participants' profiles. Later, we interpret the quantitative and qualitative results concerning the debugging assignments.

7.3.1 Participants Profile

Our study observed 28 participants. Of the 28 participants, 19 had a Master degree and 8 hold a PhD, 1 participant only had a bachelor's degree. Both groups are approximately balanced in the number of participants from 6 to 10 years of experience developing software. Appendix E.5.1 shows charts with the details of the participants' workplace and scholar degree.

As mentioned before, the posttest questionnaire requested information about the participants' profiles. Figure 7.2 shows a summary of the participants' experience regarding programming using actor model variants. Overall, groups are balanced in actor knowledge because the same number of participants, i.e., 11, reported having experience with the actor model in each group. However, the experimental group has more participants with experience in Communicating Event-Loops languages (e.g., AmbientTalk and SOMns) than the control group. In our population, most participants in both groups are more knowledgeable of the process actor model variant. In particular, the language participants have programming expertise include Akka, Elixir, Erlang, and Pony. In both control and experimental groups, 3 participants did not have any experience with actor-based programs.

The posttest questionnaire also asked debugging techniques participants have used. Unfortunately, less than half of the participants in each group had experience with debuggers when developing actor-based programs, as shown in Figure 7.3. Nevertheless, we can see similar proportions for both groups when using a debugger.

In terms of debugging tools, most participants have experience with JetBrains IDEs debuggers (see details in Figure E.8 in the appendix). This fact is positive because those programming environments are rather similar to the IntelliJ IDE used in the study.

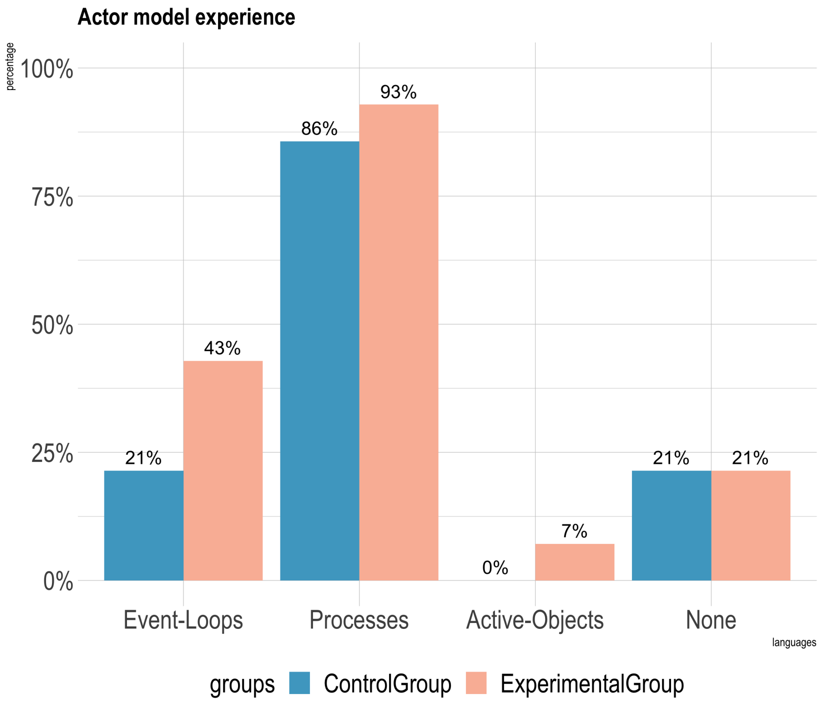


Figure 7.2: Actor model experience.

7.3.2 Quantitative Results

In this section, we interpret the quantitative results referent to the time measures for the debugging assignments.

The number of participants who succeeded in finding the bug's root cause in both assignments was almost equal in the two groups. In the control group, 9 participants

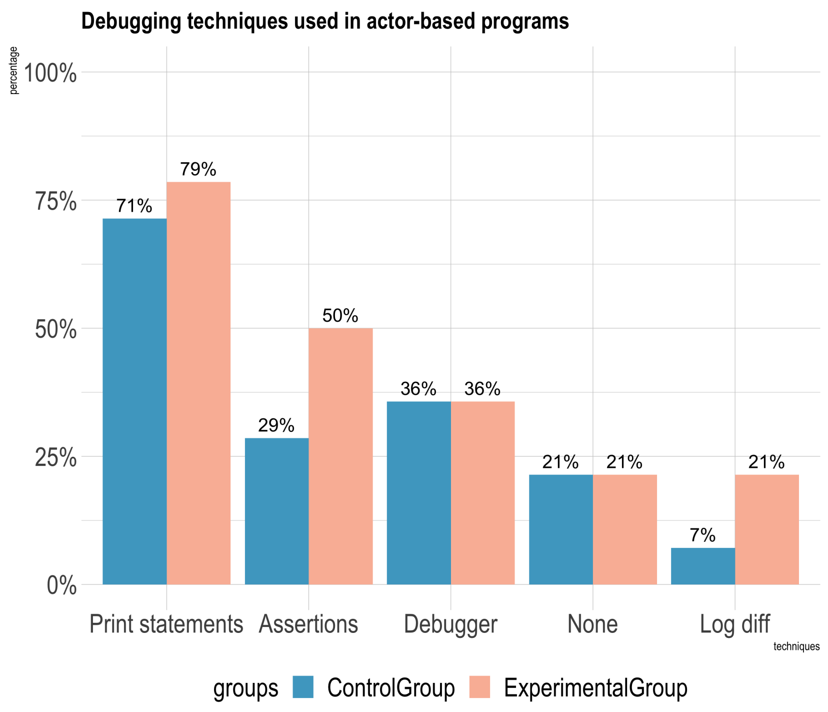


Figure 7.3: Debugging techniques.

solved assignment 1, and 12 participants solved assignment 2. In the experimental group, 9 participants solved assignment 1, and 11 participants solved assignment 2. The difference between the groups is only 1 participant in favor of the control group. Figure 7.4 shows the percentage representation of these data.

Table 7.2 shows the time measures in minutes for each of the debugging assignments of the experiment. Time was adjusted for 5 participants that reported problems with the VM, -10 min the ones that restarted the VM, and -5 min problems of a blocking IDE. We obtain measures of central tendency, i.e., median and mean values for each assignment per group. *Median*[5] is the center point in an ordered set of numbers [CJT15]. *Mean* is the arithmetic average [CJT15].

From the obtained measures with descriptive statistics, we observe that the mean and median values are greater in the control group for assignment 1. Thus the experimental group found faster the root cause of the bug in that assignment. For the second assignment, the experimental group also used less time to complete the assignment. However, there is no much difference between the values with the control group. In the following, we will analyze the results based on participants' expertise levels.

[5]E.g., for an odd number of numbers the median is the middle number, 1,2,3,4,5 = 3. E.g., for an even number of numbers the median is the average of the two centermost numbers, 1,2,3,4 = 2.5.

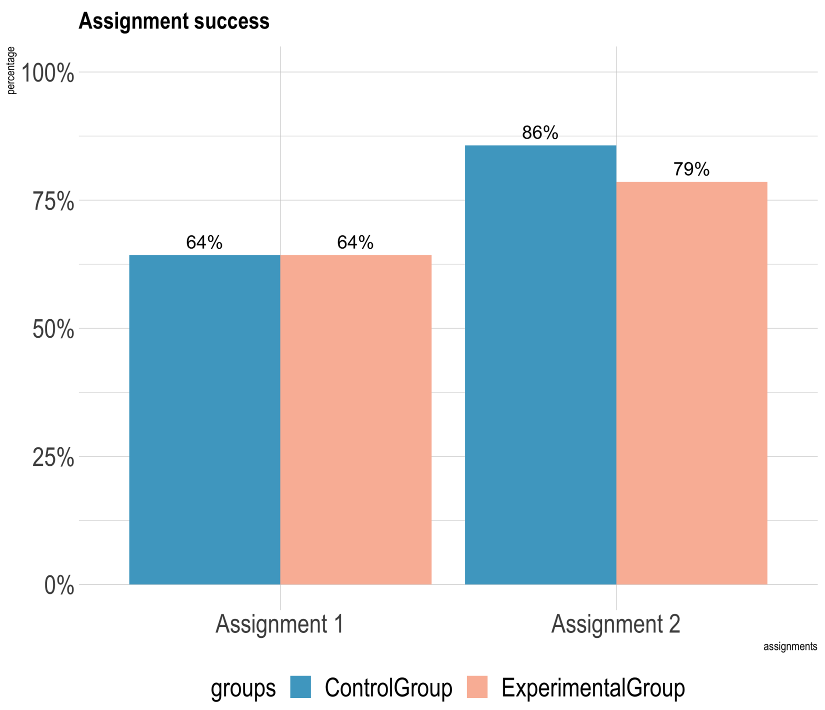


Figure 7.4: Assignment success.

7.3.2.1 Correlation of Time and Participants Expertise

To analyze the correlation of participants' expertise and the time values, we use a violin plot. Violin plots allow us to visualize the distribution of a numeric variable for one or several groups[6]. Figure 7.5 shows the experiment results for the cumulative time, i.e., for both assignments. The black bullet represents the mean, and the black line in the boxplot represents the median. We use dots in the box plot to identify small sample sizes and bimodal distributions. The dots allows interpreting the correlation between the time and participants' experience on concurrent programming.

Overall, experimental group scored lower times. Median value of experimental group ($\mu_{EG} = 20$) is less than the control group ($\mu_{CG} = 24$). Mean value for the experimental group ($\sigma_{EG} = 23$) is also less than the mean value for the control group ($\sigma_{CG} = 24$). The total number of participants for the control group that solved the assignments is 21, whereas the number of participants for the experimental group is 20.

[6]We considered them over box plots because box plots hide data distribution, e.g., there is no visual difference between a normal vs. bimodal distribution with a box plot. In contrast, a violin plot can show differences using the density information [Hol18].

Group	Control group	Experimental group
Assignment 1	20	18
	37	36
	23	23
	28	13
	14	36
	30	19
	30	17
	40	18
	25	35
Median	28	19
Mean	27	24
Assignment 2	19	20
	17	23
	21	18
	11	28
	25	23
	19	18
	18	15
	24	25
	30	18
	20	19
	36	30
	25	
Median	21	20
Mean	22	22
Total median	24	20
Total mean	24	23

Table 7.2: Times for solving the debugging assignments (in minutes).

We now analyze the times with respect to the participants' expertise. Figure 7.5 shows different colors for each expertise level. We observe that expert participants (score = 2 and score = 3) of the experimental group solved the assignment in less time compared to expert participants of the control group. Specifically, the mean value for expert participants of the control group is 24, whereas the mean value for the experimental group is 21. Hence, we can infer that participants' expertise was important to solve the assignments in less time in the experimental group. From the figure, we can derive that participants with more expertise in concurrent programming with actors used Apgar more efficiently than expert participants that used traditional debugging techniques.

Now we will analyze the time for each assignment independently. Figure 7.6 shows times of participants for solving assignment 1 and assignment 2. We can see in assignment 1 that expert participants of the experimental group solved the assignment in less time than the rest of the participants. Conversely, in the control group, expert participants took more time to solve this assignment. Median value of the experimental group ($\mu_{CG} = 19$) is less than the control group ($\mu_{EG} = 28$). Mean values are $\sigma_{CG} = 27$ and

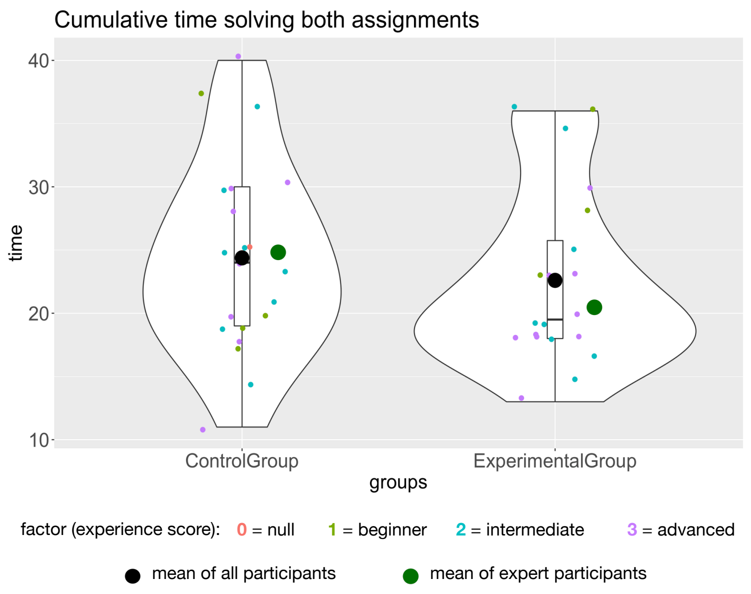
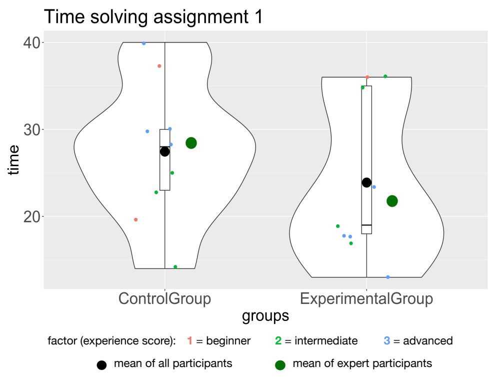
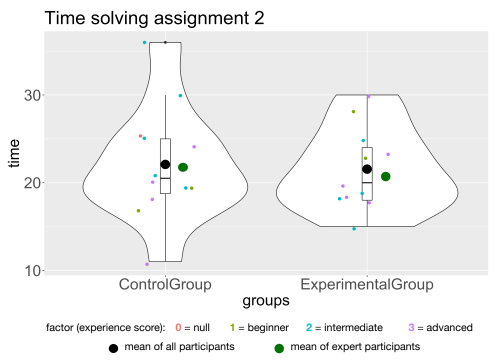


Figure 7.5: Violin plot with cumulative time for the control and experimental group.

$\sigma_{EG} = 24$ for control and experimental group respectively. In the figure we also show the mean values for expert participants which are 27 for the control group and 22 for the experimental group.



(a): Assignment 1



(b): Assignment 2

Figure 7.6: Violin plot with time values for the control and experimental group in each assignment.

In the case of assignment 2, intermediate and expert participants of the experimental group, solved the assignment in the same time approximately. For the control group, time is more spread. Median value of both groups is almost the same, $\mu_{CG} = 21$ and

$\mu_{EG} = 20$. Mean values are $\sigma_{CG} = 22$ and $\sigma_{EG} = 22$ for control and experimental group respectively. In the figure we also show the mean values for expert participants which are 22 for the control group and 21 for the experimental group.

From the results shown in Figure 7.6, we conclude that most of expert participants solved assignment 1 in less time for the experimental group. On the other hand, in assignment 2, results are similar between both groups. There is no much difference regarding the participants' expertise in that case. Considering these results, we argue that knowledge and experience about programming with actors is required to benefit from the advanced techniques we proposed in our proof of concept debugger.

7.3.2.2 Statistical Significance of the Results

After measuring the time, we checked if the time difference between both groups is significant. In particular, we formulate the following null (H_0) and alternative (H_1) hypothesis:

H_0 : There is no difference of assignment completion time between the groups.

H_1 : There is a difference of assignment completion time between the groups.

First, we run the Shapiro-Wilk test on the times data. The null hypothesis of this test is that the *sample distribution is normal.* If the test is significant (i.e., p-value $<$ 0,05), the distribution is non-normal. We obtained p-value = 0,02678 which is significant, and then this distribution is not normal for both assignments, i.e., for the cumulative times of both assignments. We then used an independent 2-group Mann-Whitney U Test non-parametric test to compute the data significance.

Running the Mann-Whitney U Test for the time data of each assignment resulted in p-values greater than the alpha level (0,05). For assignment 1 we obtained p-value = 0,2687. For assignment 2 we obtained p-value = 0,8045. For the cumulative time of both assignments, we obtained p-value = 0,2659. Then, because p-value $>$ 0,05 the experiment failed to reject the null hypothesis H_0. This means that we do not have statistical evidence that the difference in time between the groups is not due to chance instead of the independent variable (i.e., IV = debugging techniques). Since results might be caused by chance, we cannot make generalizations about the opposite of our hypothesis either. Section 7.4 analyzes threats to validity to contextualize the described statistical result.

7.3.3 Qualitative Results

In this section, we will interpret qualitative results related to participants' perception about the debugger and the experiment. As mentioned before, part of the posttest included a 5-point Likert scale about the *debugging assignments* (see Section 7.3.3.1) and the *debugging features* (see Section 7.3.3.2) participants had used in the experiment (see question 10 in Appendix E.4).

7.3.3.1 Debugging Assignments

We now analyze the results for the debugging assignments. We denote with letters A and B the statements regarding the debugging assignments:

A: The debugging assignments were difficult.

B: The debugging assignments are representative of common bugs I have seen in actor-based programs.

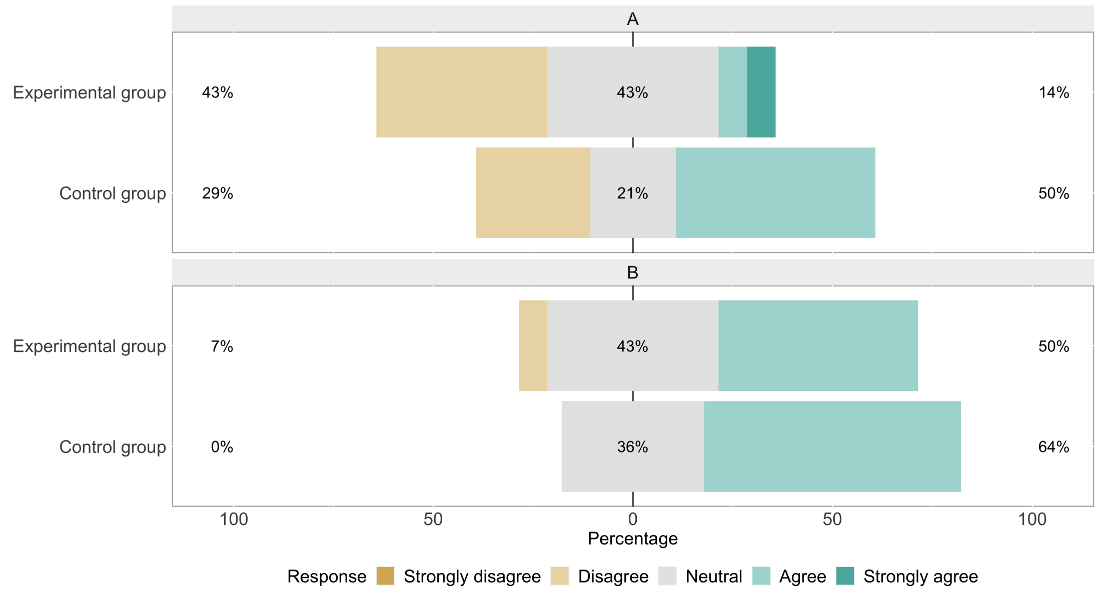


Figure 7.7: Comparison of participants' evaluation of both experimental and control groups about the debugging assignments. The percentage values shown on the left represent the sum of negative answers for the statement, i.e., *strongly disagree* and *disagree*. The values in gray slots represent the neutral answers. Finally, the percentage values on the right represent positive answers to the question, i.e., *agree* and *strongly agree*.

Figure 7.7 shows a stacked barplot of the results to analyze Likert-type items. Each box denoted with a letter represents a question (i.e., a statement), and each box consists of two rows of answers, one for the experimental group and the second one for the control group. The percentage values shown on the left represent the sum of negative answers for the statement, i.e., *strongly disagree* and *disagree*. The values in gray slots represent the neutral answers. Finally, the percentage values on the right represent positive answers to the question, i.e., *agree* and *strongly agree*.

We can observe that 50% of participants of the control group found more difficult the debugging assignments. In contrast, only 14% of the experimental group were positive in this statement. Then, participants of the experimental group found the assignments less difficult, although 43% were neutral.

Regarding if the bugs introduced in the assignments were common bugs that participants have seen in actor-based programs, we could observe that half of the participants of the experimental and control group agreed, in 50% and 64% for the experimental and control group, respectively.

7.3.3.2 Debugging Features

Participants of the study assessed the following statements regarding the debugging features:

C: Message breakpoints and stepping operations help to reduce the effort when searching the root cause of concurrency bugs.

D: The combination of sequential and message stepping is effective to inspect actor's turn.

E: Visualization of message causality is useful for understanding the program while debugging.

F: The asynchronous stack trace is useful for identifying message ordering problems.

G: The plugin debugging views are useful to inspect actor's state.

H: The debugging techniques used in the experiment assist developers not only to discover program faults but to comprehend program's behavior.

Figure 7.8 shows the Likert representation of participants' answers about the debugging features they have seen during the experiment. The control group participants did not answer statements C, D, E, and F, because the mentioned advanced features was not present in their version of the debugger as it only included sequential debugging features. This is denoted in the figure by scoring 100% neutral for those four statements.

Most experimental group participants (71%) reported that message breakpoints and stepping operations help reduce the effort when searching the root cause of concurrency bugs (statement C). Also, 64% participants agreed with the effectiveness of combining sequential and message stepping (statement D). However, only 29% of participants that experienced the message causality visualization considered it useful for understanding the program (statement E). The same percentage of participants considered the asynchronous stack trace useful for identifying message ordering problems (statement F). In section 7.4 we will discuss participants' comments about these two features.

As for the usefulness of the plugin debugging views (statement G), most of the experimental group participants (86%) were positive. On the other hand, most control group participants (71%) were also positive about their views.

Finally, less than half of the control group (43%) expressed positively regarding the value of the debugging techniques they used to comprehend the program's behavior

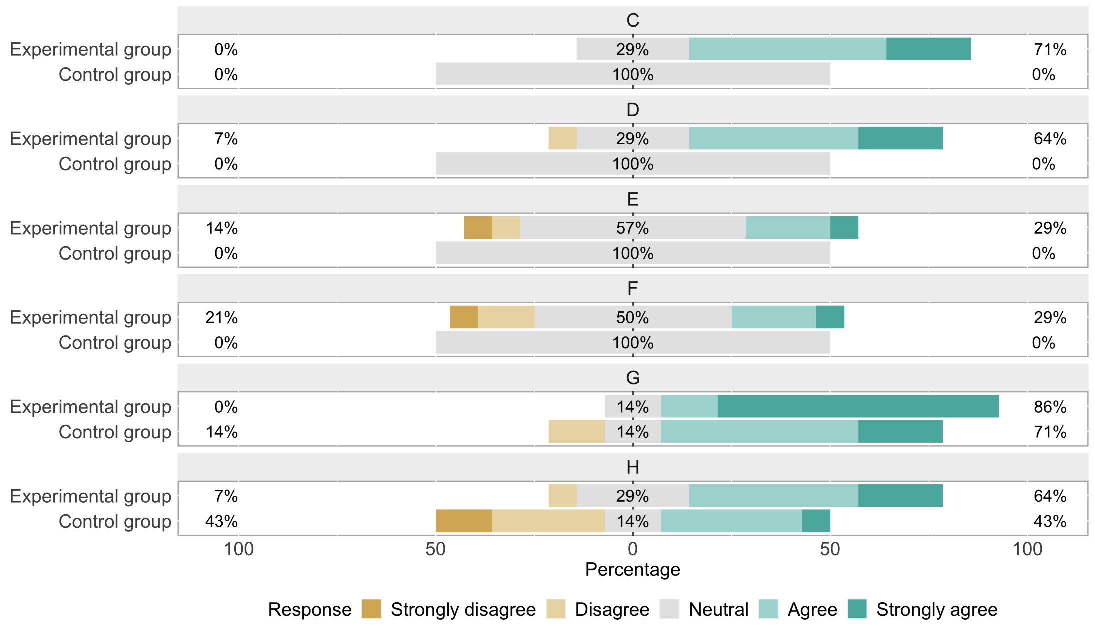


Figure 7.8: Comparison of participants' evaluation of the experimental group about the debugging features. Percentages work similar as in Figure 7.7.

(statement H). Meanwhile, most experimental group participants (64%) agreed that the debugging features they used could help developers better to understand the debugged program.

7.3.3.2.1 Correlation of participants expertise with the statements We now analyze the statements on our advanced debugging features (i.e., statements C to H). Figure 7.9 depicts a violin plot that shows a correlation between participants' expertise and their perception about the advanced debugging features. We now discuss how participants with less or higher scores rate each statement.

In statement C, intermediate and advanced participants mostly agree that message and stepping operations help to reduce the effort when searching the root of concurrency bugs. The median value for this statement is agree (4).

In statement D, intermediate and advanced participants mostly agree on the combination of sequential and message stepping to be effective to inspect actor's turn. The median value for this statement is agree (4).

In statement E, we can see that experts' assessments about the visualization vary between neutral, disagree, strongly disagree, and strongly agree. The majority of the rest of the participants have a neutral position. Then, the median value for this statement is neutral (3).

In statement F, advanced participants are neutral or disagree about the possibility that the asynchronous stack visualization is useful for identifying message ordering problems. The median value for this statement is also neutral (3).

In statement G, intermediate and advanced participants agree that the plugin views are useful to inspect actor state. The median value for this statement is strongly agree (5).

In statement H, intermediate and expert assessments agree and strongly agree regarding whether the shown debugging features help comprehend the program's behavior. The median value for this statement is agree (4).

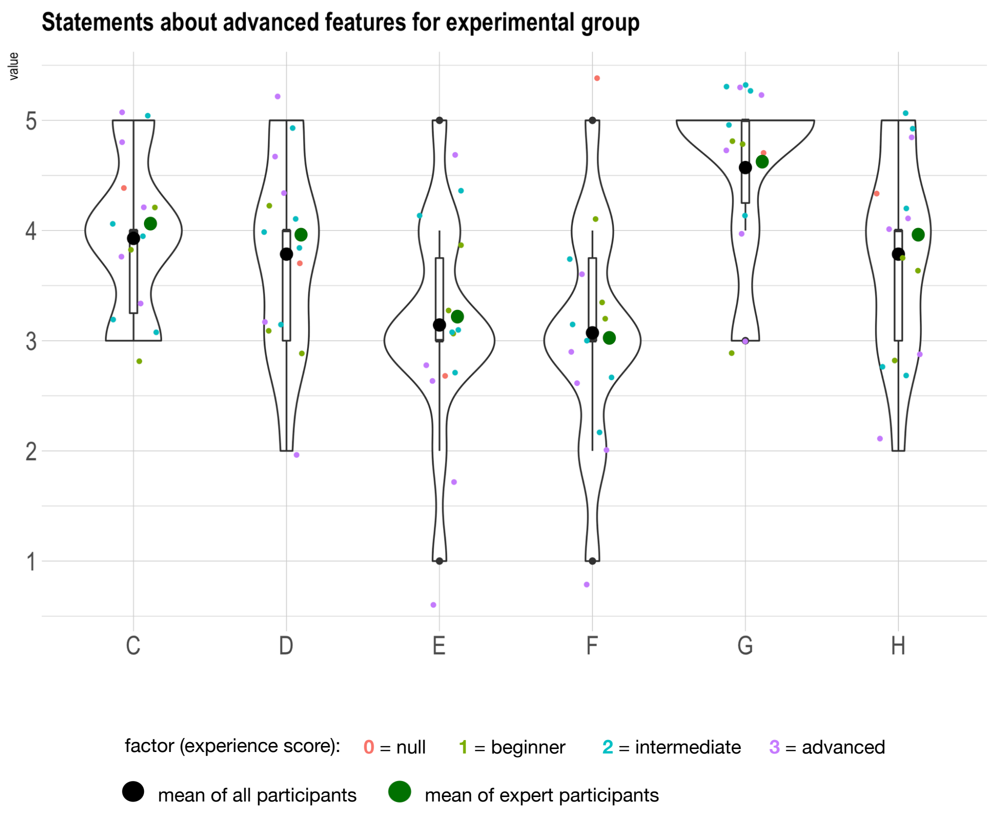


Figure 7.9: Violin plot that shows a correlation between participants expertise and their perception about the advanced debugging features. The 5-point Likert scale is represented in the Y-axis, where 1 = strongly disagree, 2 = disagree, 3 = neutral, 4 = agree and 5 = strongly agree. The X-axis represents the statements about the experiment.

From the analysis shown in Figure 7.9 we conclude that expert participants agreed with the effectiveness of message breakpoints and stepping operations to inspect actor's turn. Furthermore, expert participants gave higher rates to the combination of sequential and message-based stepping. Contrary, the expert's assessment varies for visualizing message causality and having an asynchronous stack trace. We think that more research on visualization techniques is needed. Finally, experts' perception about the usefulness of the advanced debugging views and the advanced debugging techniques was high (agree and strongly agree).

7.3.3.3 Observations from Recorded Debugging Operations

As mentioned before, we automatically recorded the debugger operations performed by the users. In this section we analyze the logs corresponding to the participants of the experimental group. We count the debugging operations that participants used from these logs, i.e., a breakpoint or a stepping operation (we do not measure the frequency but if each participant used the debugging operation at least once).

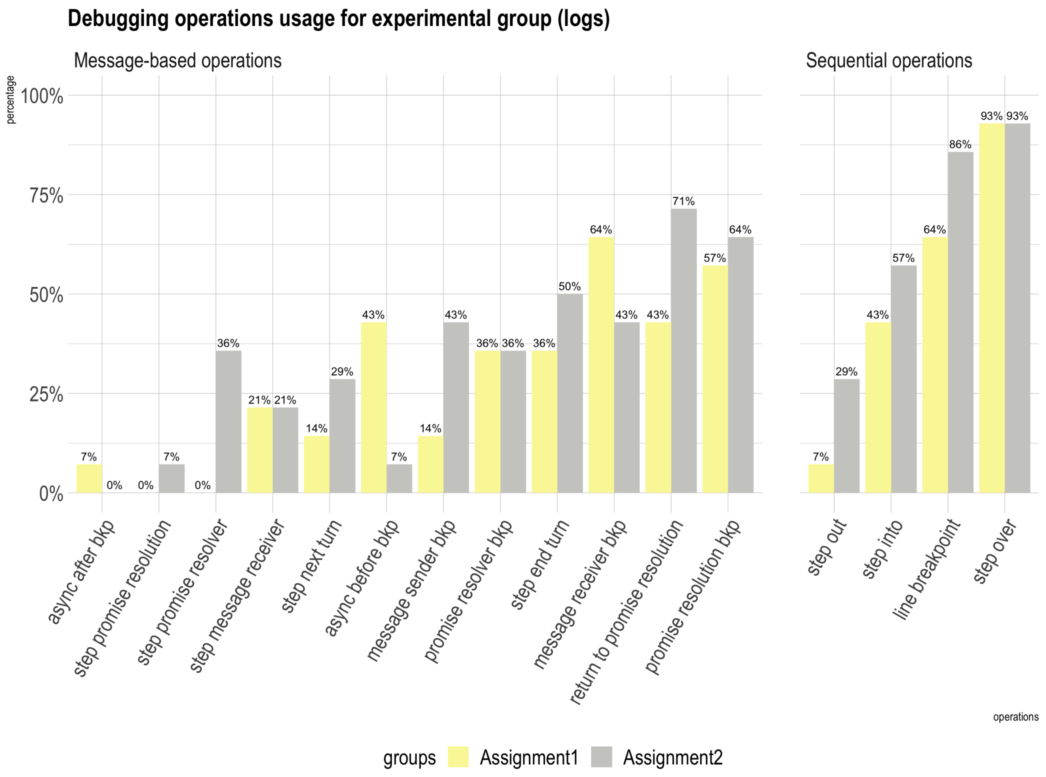


Figure 7.10: Debugging operations used for successful and unsuccessful assignments in the experimental group, grouped by sequential operations and message-based operations.

Figure 7.10 summarizes the two types of debugging operations provided in the debugger, sequential and message-based for all participants' assignments, i.e., participants that found the bug, and we also count the ones that did not find it. Percentage represent the usage of the operation by each participant, which is computed from the total of participants of the group (i.e., 14). For example, only the 7% of the group used asynchronous after breakpoint in assignment 1. We observe that sequential operations such as line breakpoint and step over were the most used operations by the participants in both assignments. We also observe that the usage of operations from assignment 1 to assignment 2 considerably increases. We believe that there is a learning process by the participants regarding the breakpoints and stepping operations since we observed growth in 13 of the 16 debugging operations available.

Figure 7.11 shows the cumulative frequency of operations used during the experiments by the experimental group. The figure also shows that participants used more debugging operations for the second assignment in the two types of operations, sequential and message-based.

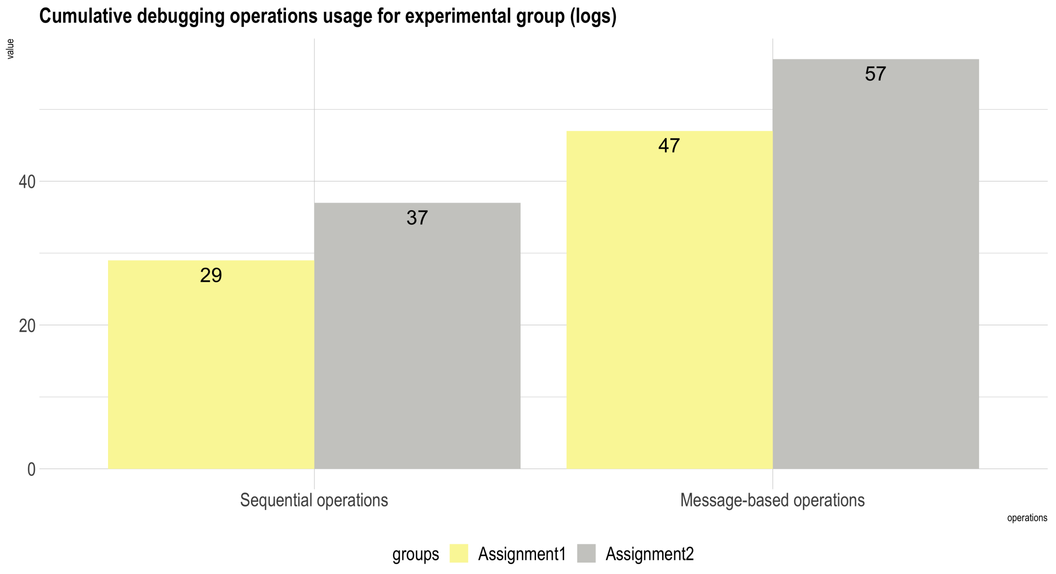


Figure 7.11: Cumulative usage of the debugging operations.

We believe that learning we observe in Figure 7.10, and Figure 7.11 is positive because participants were able to learn the sequential and message-based debugging operations that were new for them. Less than half of the participants (35,7%) in both groups have used a debugger before (see Figure 7.3). We argue that our tool could assist more than half of the participants in debugging an actor-based program for the first time with the new advanced techniques.

7.3.3.3.1 Correlation of debugging operations usage with participants programming experience We now turn our attention to the correlation between the debugging operations usage per years of experience. As shown in Figure 7.12 5 out of 7 participants in the group of 6 to 10 years learned more operations from assignment 1 to assignment 2. In the rest of the groups, however, there are fewer participants that showed this learning. Thus, we conclude that in the experiment's population, participants with more than 5 years of experience learned better the use of debugging operations. In contrast, it is unexpected that participants with more than 10 years of experience do not show this learning compared to the group of 6 to 10 years.

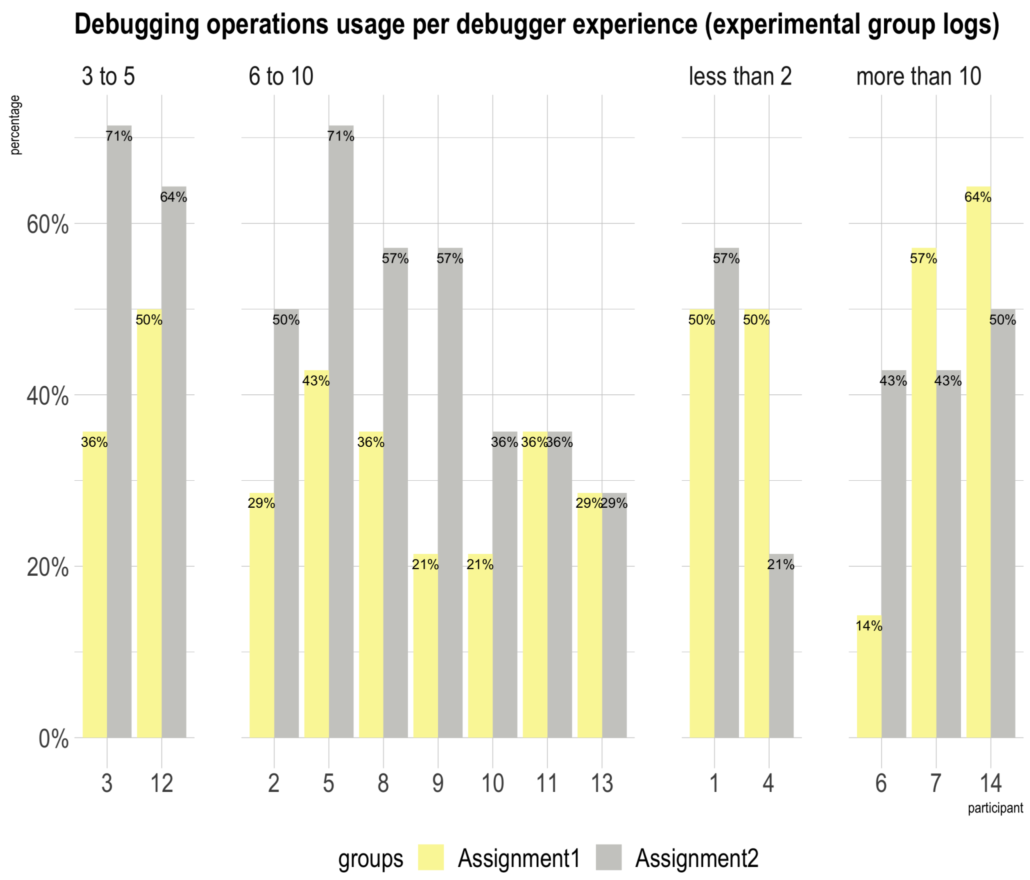


Figure 7.12: Debugging operations correlation with years of experience.

7.3.3.4 Time Pressure

We now discuss the perception of participants with respect to time pressure. As shown in Figure 7.13 the experimental group participants felt more time pressure than the control group when solving the debugging assignments.

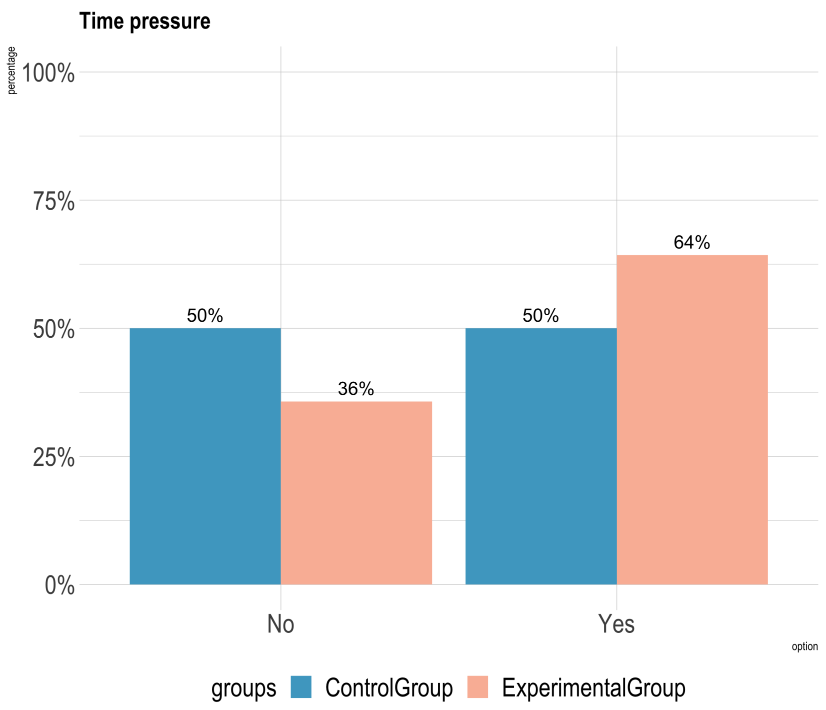


Figure 7.13: Participants perception about the time pressure.

From the comments of question 15 and 16 of Appendix E.4 we obtained that some experimental group participants (2 out of 14) explicitly said the amount of time was not sufficient for them to perform the experiment effectively, e.g., *"In general I feel I was not able to use the debugger effectively. I spent a lot of time "fiddling around with it", trying to get to grips with it. I feel I needed more intro/guidance, and certainly more experience, with the debugger to be exploit it fully. I think it is a really functional tool, but it did not really help that much in the short time available, since I'm not used to work with it."* After finishing the second assignment, that participant said via chat to the host about the debugger that *"...learn to use it in a short amount of time was difficult"*. A second participant said in the questionnaire that *"I thought the current offerings were pretty complete... in the short amount of time, I was not able to get too familiar to them, and I feel like they probably would have helped me solve the tasks quicker."* One participant was an expert in programming with actors, and the second one intermediate.

From those comments, we conclude that time was not sufficient for all participants to solve the assignments effectively. Considering these results in combination with the learning that we observed when we analyzed the debugging operations usage (see Figure 7.10), we believe that more time with the tool seems to be needed to get familiar with the debugger.

7.3.3.5 Observations from Comments

Finally, we analyze participants' comments, about the experiment and about the debugging features presented in the debugger (from questions 15 and 16 in Appendix E.4 respectively).

In the questionnaire, participants were asked to comment about the experiment (question 16). Overall, 71,4% of participants of the experimental group and 50% of the control group gave comments about the experiment. Both groups expressed impressions mainly about their unfamiliarity with the tool or the language in that question.

On the positive side (from question 15 and question 16), 2 participants of the control group declared they would like to try the debugger version of the experimental group, e.g., *"...I would definitely like to try the experimental group set-up, if that would be possible", and "I would be interested in the experimental features I did not get to see in the control group".* In the experimental group, 3 participants referred to some of the advanced features as useful explicitly, e.g., *"regarding the combination of sequential message / stepping: I do think this is useful...", "...I thought the current offerings were pretty complete" and "I don't think there is anything to add".*

On the other hand, participants specified some issues they faced during the experiment, which are summarized in Table 7.3. As mentioned before, participants of both groups did remarks related to their lack of experience with the tool and the language. As we can observe in the table, more than half of the participants of the experimental group (8 out of 14) declared their unfamiliarity with the debugger tool. Thus, unsurprisingly, one of the key issues of the experiment was that participants was not familiar with the tool.

Regarding the interaction with the VirtualBox image, there were participants in both groups that notice the VM slow and unresponsive. This issue was seen more frequently for participants of the experimental group. Performance issues may be related due to virtualization on the participant's machine, but unfortunately, due to COVID-19, we could not use a more controlled hardware and software environment, i.e., a room equipped with the same sort of computers configured by us.

We believe our prototype debugger was stable since there was only one comment from a participant of the experimental group reporting implementation issues, i.e., *"I don't think all Variables are always displayed in the Variables tab".*

7.3.3.5.1 Advanced debugging support suggested by the control group Question 15 of the questionnaire asked participants for other features they would like to see in the tool. Table 7.5 show a list of suggestions from both control and experimental groups, respectively.

Table 7.4 shows that most of the features participants of the control group wanted in the debugger are already supported by Apgar. The most frequent comments were related to the need to show visualizations for message exchange between the actors, and

Issue	Participants control group	Percentage	Participants experimental group	Percentage
Not sufficient amount of time	0	0%	2	14,3%
Lack of experience with the tool	1	7,1%	8	57,1%
Lack of experience with the language	3	21,4%	1	7,1%
Unresponsive/slow VM	1	7,1%	5	35,7%
Review variables information	0	0%	1	7,1%

Table 7.3: Summary of issues found in explicit comments by participants of both groups about the experiment. The number in each cell indicates the number of participants that referred to that issue. These numbers were collected from question 16 of the questionnaire and comments participants gave during and after solving each assignment (via Zoom chat messages in private to the host of the experiment).

visualizations for the actor state visualization. Next were the comments related to the advanced features that were disabled, e.g., *"... analyzing concurrent code with a sequential debugger is difficult..."* and *"... I would expect when I step into a message, whether it's a callback or a method to go inside this."*. One participant explicitly mentioned the need for *"A message trace or stack trace"*, which is a debugging feature that we already supported in the experimental version of the debugger, i.e., an asynchronous stack trace.

Finally, three comments gave suggestions for features which we are currently experimenting (i.e., show unresolved promises as described in section Section 5.1.5) or plan in future work (i.e., reverse debugging as we will discuss in section Section 9.4).

7.3.3.5.2 Advanced debugging support suggested by the experimental group

We now analyze the suggestions given by the experimental group. More than half of the participants of the experimental group (11 out of 14) gave feedback in question 15 of the questionnaire. Most of the comments are related to improvements of the already supported debugging features, such as improving the relation between the sentbox to the asynchronous stack and merging mailbox and sentbox. Similar to the control group participants, we also got comments on showing information of unresolved promises and replay message execution.

Finally, 3 participants mentioned that in the turns view they could not find the information they were looking for while it was there, e.g., *"The turn view can help to find the message ordering, but, as far as I can see, didn't help me to find out what exactly was sent..." and "...I also did not understand what the effect of "selecting" a message is"*.

	Debugging feature	Number of comments	Percentage
Features supported in the experimental version	Message exchanged visualization	5	35,7%
	Actor state visualization	5	35,7%
	Async stack trace	1	7,1%
	Features that were disabled, e.g. message receiver and promise resolution breakpoints	4	28,6%
Not supported features	Back in time debugging	1	7,1%
	Showing unresolved promises	1	7,1%
	Showing previous debugging steps	1	7,1%

Table 7.4: Summary of comments about the debugger by participants of the control group. The number in each cell indicates the number of participants that referred to that issue. These numbers were collected from answers to question 15 of the questionnaire. The percentage is calculated based on the number of comments and the total participants in the control group (i.e., 14).

From those comments we conclude that some participants actually did not understand some debugging features.

	Debugging feature	Number of comments	Percentage
New features	Show unresolved promises	1	7,1%
	Replay a message execution	1	7,1%
Improvements to the supported features	Improve visualization of turns	3	21,4%
	Design of the tabs, higher coupling between them	1	7,1%
	Highlight active/hit breakpoint	1	7,1%
	Relate sentbox to the async stack	1	7,1%
	Merge mailbox and sentbox	1	7,1%

Table 7.5: Summary of comments about the debugger by participants of the experimental group. The percentage is calculated based on the number of comments and the total participants in the experimental group (i.e., 14).

7.3.4 Overview of the Results

On the one hand, in the measures made for the quantitative study, we have obtained differences between the groups in favor of the experimental group. In particular, we observed that expert participants who used the advanced debugging techniques we proposed in Apgar found the bugs in less time than participants who used traditional debugging techniques. However, we obtained that the differences between the groups are not statistically significant, i.e., the study failed to reject the null hypothesis. This means we cannot generalize the results in which the advanced debugging techniques help (or not help) to reduce the time of finding the root cause of the bug.

On the other hand, in the qualitative study, we have obtained positive impressions from the participants regarding advanced features, e.g., message breakpoints and the combination of sequential and message stepping. Interestingly, we obtained those rates from most intermediate and expert participants of the experimental group. On the positive side, most intermediate and expert participants found the plugin views useful to inspect actor state and the debugging techniques helpful for program comprehension. Unfortunately, more than half of the participants were neutral about the features of turn visualization, and half of the participants were neutral about the asynchronous stack trace.

From the recording of the debugging operations, we obtain the insight that participants of the experimental group learned the majority of the debugging operations they have available, from the first assignment to the second one. This is positive because that shows that the debugging features are not difficult to learn. The learning process is noticeable for participants with programming experience from 6 to 10 years. From participants' comments in the questionnaire, we conclude that some issues could negatively influence their execution in the assignments, e.g., some participants emphasized that the lack of experience in the language and the tool was the main problem in the task. Positively, most of the comments by participants of the control group pointed out the need for debugging features we already support in Apgar.

Although we cannot generalize our user study results, we consider we obtained positive assessments from participants regarding the advanced debugging techniques we proposed, not only to identify the root cause of concurrency bugs but also to help developers understand the program's behavior.

7.4 Threats to Validity in Mixed Methods Experimental Research

This section mentions the different threats that could invalidate the results we have obtained in the user study and how we addressed them. Creswell et al. [CC17] mention three threats we need to consider when applying a mixed methods experimental design. Table 7.6 shows these threats and a general strategy to minimize them.

Here we explain the strategies we used to minimize each of the mentioned threats in our research study.

1. **Internal and external validity** In our study, our priority is to determine if the relationship between the IV and DV is causal, which in our case corresponds to ensure correctness for possible internal threats [CJT15]. External validity for cause-and-effect relationships is often needed in research studies conducted by different researchers in different settings with different kinds of people. However, our population is not large (i.e., 28 participants < 30). Thus, we focus on addressing *internal threats* about our quantitative study (see in Section 7.4.1).

Validity threat	General strategy to minimize the threat
1. Not addressing threats to internal and external validity in an experiment design	Address internal and external threats noted in the literature about experimental designs
2. Not specifying why and where the qualitative component is embedded in the experiment	Provide an explicit rationale for collecting qualitative data and its use in the experimental design
3. Introducing bias in the experimental design when qualitative data are collected during the experiment	Consider strategies such as unobtrusive data collection during the experiment that do not introduce bias that might alter the experimental outcomes

Table 7.6: Threats in mixed methods experimental design, from [CC17].

2. **Embedded qualitative component** We give arguments about selecting our research design consisting of quantitative and qualitative measures and how we apply it in our study in Section 7.1.1. Because we collected qualitative data in a posttest questionnaire, we additionally analyzed some validity threats that could occur and possible strategies to minimize their effect (see Appendix E.6).

3. **Bias when collecting qualitative data** We did not perform qualitative questions during the experiment, but only when participants finished each assignment. Questions were only related to the assignment they just finished, e.g., *"How did they found the bug?"* and *"Which debugging operations they used the most?"*. Thus, we did not give any information that could bias the next assignment or the questionnaire. Participants worked individually during the whole experiment.

In the following section, we explain how we minimize each threat to internal validity.

7.4.1 Internal Validity Threats

As mentioned before, quantitative studies are subject to *internal validity*, i.e., it is important to validate the conclusions made for the cause-and-effect relationship between the variables of a quantitative study [CJT15]. Ensuring internal validity means that no other variable apart from the independent variable can influence the dependent variable. Therefore, we need to control all other variables (i.e., extraneous variables) that can influence participants during the experiment and affect the results. In our study, we consider the debugging techniques the *only* variable that can influence the participants to identify the root cause of concurrency bugs.

Christensen et al. [CJT15] identified 8 threats to internal validity. As we mentioned in Section 7.2.1 we used the control technique of random assignment of participants to the groups. Having a control group and applying randomization minimizes all the 8 threats mentioned by Christensen et al. Table 7.7 describes each of the validity threats to internal validity.

In the history threat, Christensen et al. [CJT15] mentions that could be a particular threat named *differential history* in which one group experiences the history event and the other group does not, which is not solved with a control group. Because we run the experiment in two weeks, we minimize this threat regarding possible time differences by controlling the exact time in which participants watch the tutorial videos about the language and the debugger. The host screen shared the videos for all participants at the same time. Later, the host guided each participant through the different experiment steps using private communication via chat messages. Besides, each time slot where we run the experiment was meant only for one group of participants, i.e., participants of the control group or of the experimental group. This way, the host only controlled participants using only one version of the debugger.

On the other hand, we are aware that the time-lapse in the research study should be short to avoid history becoming a rival explanation. However, because of the current situation with the COVID-19 outbreak was not possible to do the experiment for all participants in the same location. As we mentioned in Section 7.1.2, at the beginning of the experiment, the host read a code of conduct with rules for the participants to compromise to follow them during the experiment and after they finish it, at least during the experiment's time period (see Appendix E.1).

Validity threat	Description	Strategy to minimize the threat
1. History	Any event that can produce the outcome, other than the treatment condition, that occurs during the study before posttest measurement	control group
2. Maturation	Any physical or mental change that occurs with the passage of time and affects dependent variable scores	control group
3. Instrumentation	Changes from pretest to posttest in the assessment or measurement of the dependent variable	control group
4. Testing	Changes in a person's score on the second administration of a test resulting from having previously taken the test	control group
5. Regression artefact	Effects that appear to be due to the treatment but are due to regression to the mean	control group
6. Attrition	Loss of participants because they do not show up or they drop out of the research study	control group + pretest [SCC01]
7. Selection	Production of nonequivalent groups because a different selection procedure operates across the groups	randomization
8. Additive and interactive effects	Differences between groups is produced because of the combined effect of two or more threats to internal validity	randomization

Table 7.7: Threats to internal validity, from [CJT15].

Besides the threats to validity shown in Table 7.7 we identify other factors that could affect our study's internal validity, summarized in Table 7.8. There we mention a possible solution strategy to minimize the influence of these threats in the results.

Validity threat	Strategy to minimize the threat
1. Participants lack of knowledge on SOMns language	Tutorial about SOMns + SOMns cheat sheet
2. No familiarity of the participants with the debugged programs	Selection of new programs and give a description for each assignment
3. Debugging assignments are difficult	Avoid the use of complex constructs from SOMns language and add question in posttest questionnaire
4. Introductory tutorials	Do small debugging exercise, but participants can choose any feature they found most appropriate
5. Time pressure	Add question in the posttest questionnaire
6. The programs in the assignment might not be representative	Add question in the posttest questionnaire

Table 7.8: Additional threats to internal validity.

Here we discuss the threats of table 7.8 with the results we obtained:

1. **Lack of language knowledge** We created a video tutorial with the main syntax elements of SOMns. Also, we sent to participants in advance a SOMns cheat sheet by email. Although we provided the mentioned documentation, some participants emphasized in their comments that their lack of experience with the language and the debugger influenced their performance when solving the assignment.

2. **Unfamiliarity with the programs** Programs that have not been seen before by the participants should equate better the groups and reduce the threat in which some participants could know more about the program in advance. To minimize threat 2 we also give a description and a conceptual diagram of the program in the assignment (as shown in Appendix E.3). Some participants declared as useful the assignment's description.

3. **Difficulty of debugging assignments** We added a question in the posttest questionnaire to measure participants' perception about the assignments. We obtained that 50% of the participants from the control group found the assignments difficult, whereas only 14% of participants from the experimental group agreed with this idea. Then, only a 32% of the total of participants confirmed they found the assignments difficult. i.e., 9 out of 28 participants.

4. **Introductory tutorials** We gave both groups a tutorial about the debugger, and we used `PythagorasCalculator` program to show the debugging features, which are different for each group. Moreover, participants freely chose the debugging features they used during the experiment. As such, we consider results should not be biased by any of these materials. Besides, tutorials had similar time length for each group.

5. **Time pressure** During the experiment, 64,3% of participants of the experimental group perceived time pressure (threat 5). We think a possible reason for this is

that the number of new debugging features and debugging views was bigger than the ones experienced by the control group. Because 50% of participants of the control group perceived time pressure, we think that indeed the time limit for the experiment might have influenced the results. Furthermore, some participants expressed that learning the language and the tool took them part of the available time (see Section 7.3.3.5).

6. **Actor-based programs representative** We collected answers that show that participants' assessments were positive for the two actor-based programs used in the experiment. Participants of the experimental group agreed in 50%, and participants of the control group agreed in 64% that programs in the assignments could be representative of real-world actor-based programs. Nevertheless, we conclude that no generalization about the results can be made at the moment. Further studies with real-world applications and real-world bugs are still needed.

Furthermore, Christensen et al. mention the *statistical conclusion validity* as another validity type in quantitative studies. Statistical validity allows inferring that the independent and dependent variables covary, i.e., variations in the first variable affects the dependent variable [CJT15]. In our case, the lack of sufficient participants can threaten the statistical conclusion validity. We solved this threat using the Shapiro-Wilk test to know if the distribution was normal, and then we use Mann-Whitney U test as a non-parametric test to compute the significance of the results.

7.5 Discussion

We now discuss our results regarding our general hypothesis declared in section 7.1.1. We also analyze the results we obtained for our specific hypotheses related to the proposed advanced debugging techniques.

Although we did get differences in the time measures between the groups in favor of the experimental group, the results we obtained are not statistically significant in response to *hypothesis 1*. This means that we cannot conclude that the advanced debugging techniques reduce the effort to identify the cause of complex concurrency bugs in actor-based programs because results might be caused by chance. Furthermore, regarding the success of the assignments, we obtained almost similar results in both groups. The control group got 1 participant more that solved the assignments compared with the experimental group.

From the qualitative assessment measured with the posttest at the end of each assignment, we conclude that the time we gave to participants may not be sufficient, mainly because of their lack of experience with the tool and the SOMns language. More than half of the experimental group participants expressed that a previous interaction with the tool would make them perform the experiments better (see Table 7.3).

Although the quantitative result was not as we expected, we think we obtained positive insights from the experiment. We could see a learning of the debugging operations from assignment 1 to assignment 2. There was an attempt and learning from the participants for using the new advanced debugging techniques despite the fact they have not previous experience with the tool.

Interpreting the participants' perception about the advanced debugging features, we can conclude that we got positive results for *hypothesis 2* and *hypothesis 3*. More than half of the experimental group participants (71%) agree that message breakpoints and stepping operations can help reduce the effort when searching the root cause of concurrency bugs. Besides, most of the participants of the experimental group (64%) found effective combining both types of stepping, sequential and message-based, to inspect actor's turn.

From the answers to qualitative questions at the end of each assignment, we got that some participants (28,6%) of the experimental group read the code or use only sequential features instead of using the advanced debugging techniques (3 participants in assignment 1 and 1 participant in assignment 2). One factor that could have influenced this behavior could be the time pressure, which participants of the experimental group felt more than participants of the control group. However, from the rating results for statement G and H of question 10 of the questionnaire, we can say that participants of the experimental group had more positive assessment about the plugin views (86%) and the debugging techniques (64%) than participants of the control group (71% and 43%, respectively).

In response to *hypothesis 4*, i.e., statement E of question 10 in the questionnaire, we got that some participants who tried to use the turns view found it difficult to understand the drawings. They mention that they were not very clear or a bit cluttered. This issue can be a reason for the low rating (29%) obtained in the statement related to the usefulness of the visualization of message causality. Another factor is that some participants did not find the need to use some of the advanced views to identify the bugs. Nevertheless, although participants consider the information not yet visualized clearly enough, it was useful for some of them.

Finally, responses to *hypothesis 5*, i.e., statement F of question 10 in the questionnaire, had an equal rating result (29%) to hypothesis 4. On the positive side, in the comments, one participant of the control group suggested an asynchronous stack trace as a feature they would like to see in a debugger for actor-based programs. On the other hand, another participant proposed to relate the sentbox to the asynchronous stack in the experimental group. Although not all participants used this feature, maybe because of unfamiliarity, we believe that the asynchronous stack is a relevant debugging feature for actor-based debugging.

7.6 Conclusion

In this chapter, we have described a user study design to evaluate the advanced debugging techniques implemented in Apgar, our proof of concept debugger for actor-based programs. Our study investigates if the use of advanced debugging techniques helps identify concurrency bugs in actor-based programs. To accomplish this goal, we used a *mixed methods experimental design*, where we focused on an experimental approach, i.e., *a posttest only control group design* with embedded qualitative measures.

We conducted an online experiment for two groups of participants. The control group solved the debugging tasks using traditional debugging techniques. Meanwhile, the experimental group solved the debugging assignments using advanced debugging techniques. From the results interpreted, validated, and discussed in the previous sections, we arrive at the following four conclusions.

First, we cannot ensure that the advanced debugging techniques help to identify the cause of complex concurrency bugs in actor-based programs because time measurement was not significant statistically.

Second, the participant's perception of the debugging features is positive. We can say participants valued message breakpoints and stepping operations and the combination of sequential and message stepping. Quantifying the usage of debugging operations helped us see the operations' learning through the two assignments by the participants. Overall, the experimental group participants gave a more positive assessment regarding using the plugin views for inspecting the actor's turn. Most experimental group participants also confirmed that the debugging techniques could help developers identify the root cause of concurrency bugs and program comprehension. Thus we can conclude the advanced debugging features may be helpful in the context of developing actor-based programming.

Third, further research is needed to improve and evaluate visualization for features such as message causality and asynchronous stack traces.

Finally, from the data correlation between the participants' expertise in concurrent programming and the quantitative measures (e.g., the time) and qualitative measures (e.g., statements about the experiment), we conclude that Apgar can be considered as a valuable tool for expert developers.

Chapter 8

Online Debugging Techniques Probe-Effect Free

Online debugging techniques allow to explore the space of non-deterministic concurrency bugs, but only in *one* path of the program's execution. Besides, concurrent programs suffer from the probe-effect. To solve these issues, we explore a debugging technique that allows programmers to observe *all* possible states of a concurrent program at runtime and is probe-effect free. We call this technique multiverse debugging.

In this chapter, we first define multiverse debugging, and we apply this technique to a small language. Second, we apply multiverse debugging for a language based on the Communicating Event-Loops actor model. We showcase a debugging session example using our proof of concept of a multiverse debugger Voyager. Later, we describe Voyager calculus, a formal operational semantics for debugging actor-based programs. Finally, we explain a proof of non-interference for multiverse debugging and we compare our approach to existing related work.

8.1 Multiverse Debugging

As we mentioned in Section 3.2 debugging tools have been categorized as online and offline. Despite the presence of online debuggers in modern IDEs, a recent study showed that debugging parallel applications remains very problematic [PSTH16] 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* [Gai86] (see Section 2.6). Apgar (see Section 5.2) is an example of an online debugger that only explores one path of execution. Thus, in the presence of a *heisenbug*, it has the disadvantage that developers using the debugger will need to run more than once the program to be able to observe that bug again.

In addition to debugging techniques, another technique that has tried to solve the mentioned issues without executing the program is static analyses, e.g., static analysis to verify the boundedness of actor mailboxes [FGN$^+$03], model checkers for concurrent programs written in Erlang [CGS13a] or Scala [LDMA09], type systems to ensure type safety on reciprocal communication channels [THK94]. These techniques detect synchronization errors such as deadlocks [BLR02, FF00], incorrect ordering of locks [BPP14], and incorrect interleaving of messages in actor systems [CGS13a]. However, they often put severe restrictions on the way programs are organized (e.g., on how promises are used in actor-based programs [GGL$^+$13]). 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 that triggered the bug.

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 breakpoint-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 that 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.

8.1.1 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.

We think 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. Related work has distinguished concurrency concepts of different models in the context of debugging [MLA$^+$17]. We participated in that work exploring concurrency concepts relevant for debugging actor-based programs (see Table 5.8). 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.

In the next section, we apply our multiverse debugging recipe for a small language, and in Section 8.3 for a language based on the Communicating Event-Loops concurrency model.

8.1.2 A Multiverse Debugger for a Small Language

In this section, we apply the multiverse debugging recipe to debug ambiguous programs written in λ_{amb}. In section 8.1.2.1, we specify the base language semantics (step 1) and section 8.1.2.2 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 8.3.1. There, we will specify the semantics of a base language in which to write actor-based concurrent programs (step 1), and section 8.3.2 describes the semantics of a multiverse debugger on top of it, including multiverse breakpoints and stepping commands (step 2).

8.1.2.1 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 [McC61] 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 8.1 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.

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

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

Figure 8.1: 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. 8.2. 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 `2` while in the bottom universe, it chose `1`. 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.

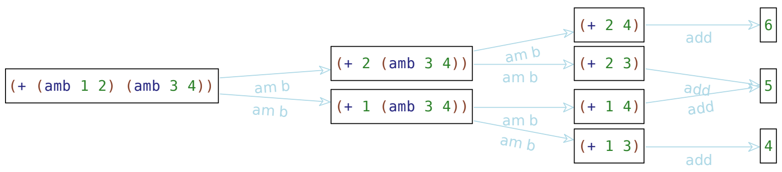


Figure 8.2: Multiverse evaluation graph of a λ_{amb} program.

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

Having 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 8.3 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.

$state$	$::=$	STEP | PAUSED	Debugger State
e_d	$::=$	$(state,\ e)$	Debugger Configuration

(STEP)

$$\frac{e \ \rightarrow_{amb}\ e'}{(\text{STEP},\ e) \rightarrow_{debug} (\text{PAUSED},\ e')}$$

(RESUME)

$$\frac{}{(\text{PAUSED},\ e) \rightarrow_{debug} (\text{STEP},\ e)}$$

Figure 8.3: 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. 8.2 in D_{amb}. The resulting session is shown in fig. 8.4. 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 the 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.

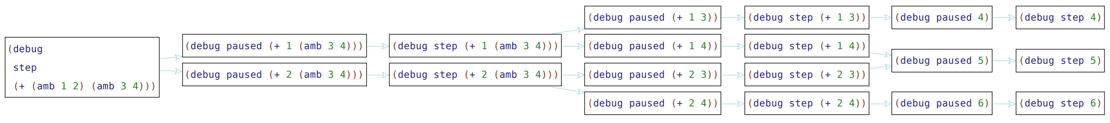


Figure 8.4: Multiverse debugging graph of a λ_{amb} program.

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 the debugger which are not observed in the base-level semantics. This means that debugging the program does not introduce 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.*

8.1.3 Challenges

The main challenge of our approach is the growth in the number of states; the number of possible states increases exponentially with 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 [Val98]. 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 that 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 having a debugging tool that 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 [CGJ$^+$01, CS13, LPD$^+$14, LHA18]. 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 8.4.1 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 [BCCZ99]. 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 number of states exposed to the programmer.

8.2 Voyager, a Proof of Concept Multiverse Debugger for Actors

Multiverse debugging borrows from breakpoint-based debuggers two features to enable an interactive exploration of the program's state, i.e., breakpoints and stepping operations. As we mentioned before (see Section 5.1), a *breakpoint* defines a point of interest in a program in which to pause execution for further inspection. Furthermore, developers can follow a program's execution through different points of interest using *stepping* operations (see Section 5.1.2). We have applied the semantics of breakpoints and stepping operations described in Section 5.1.1 and Section 5.1.2 to our multiverse debugger.

At the moment of this research, the SOMns language does not have an implementation of its operational semantics. Thus, we have employed another Communicating Event-Loops language for which already exists an operational semantics, namely AmbientTalk. AmbientTalk is a CEL language for distributed programming in mobile peer-to-peer networks. Its operational semantics is known as Featherweight AmbientTalk i.e., AT^f [VCGBS$^+$14].

In this section, we show a proof of concept of a multiverse debugger for AmbientTalk programs, named *Voyager*. Voyager[1] features a web-based frontend and a backend implemented on PLT-Redex language [2]. Target programs are written with AmbientTalk semantics 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.

In the next section, we showcase some debugging operations in Voyager through a program written in AmbientTalk. Later, we give an overview of a debugging session in Voyager.

8.2.1 Debugging a Sample Program

We now show a debugging session in Voyager for the SOMns program depicted in Listing 8.1. Appendix F shows the PLT-Redex program written in AmbientTalk semantics, but for simplicity, we show it in SOMns syntax.

The program shows an interaction between 3 actors: a `math` actor (created in Line 42, and two client actors, `client1` (created in Line 44) and `client2` (created in Line 45). The `math` actor (Lines 5 to 16) 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 `startWithResult` message and `start` message to actors `client1` and `client2`, respectively (Lines 47 and 48). 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 [LMBM18], which is common for actor-based programs. It is possible that `client1` gets the result of doubling `33` instead of doubling `12`. Table 8.1 shows all possible interleavings that the program exhibits. It also depicts the message sender for each message. Line 27 would print the result value, which is `24` for the correct interleavings and `66` for the faulty one.

[1]Voyager frontend is a web application implemented by our colleagues of the University of Ghent [SLM$^+$19], which is built on top of the debugging operational semantics we implemented in PLT-Redex, i.e., Voyager backend [LSM$^+$19].

[2]`https://redex.racket-lang.org`

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

```
class DoubleApplication usingPlatform: platform = Value (
| private actors = platform actors. |
)(

  public class Math = (
    | private result ::= 0. |
  )(

    public double: x = (
      ^ result:: x + x
    )

    public getResult = (
      ^ result
    )
  )

  public class Client new: math name: name = (
    | private math = math.
      private name = name.
    |
  )(

    public startWithResult: x = (
      math <-: double: x.
      ^ math <-: getResult whenResolved: [:res |
        (name +' '+res) println.
        ].
    )

    public start: x = (
      ^ math <-: double: x.
    )

  )

 public main: args = (
   | completionPP client1 client2 p1 p2 math |

   completionPP:: actors createPromisePair.

   math:: (actors createActorFromValue: Math) <-: new.

   client1:: (actors createActorFromValue: Client) <-: new: math name: 'c1'.
   client2:: (actors createActorFromValue: Client) <-: new: math name: 'c2'.

   p1:: client1 <-: startWithResult: 12.
   p2:: client2 <-: start: 33.

   p1, p2 whenResolved: [:r |
      completionPP resolve: #ok.
   ].

  ^ completionPP promise
 )
)
```

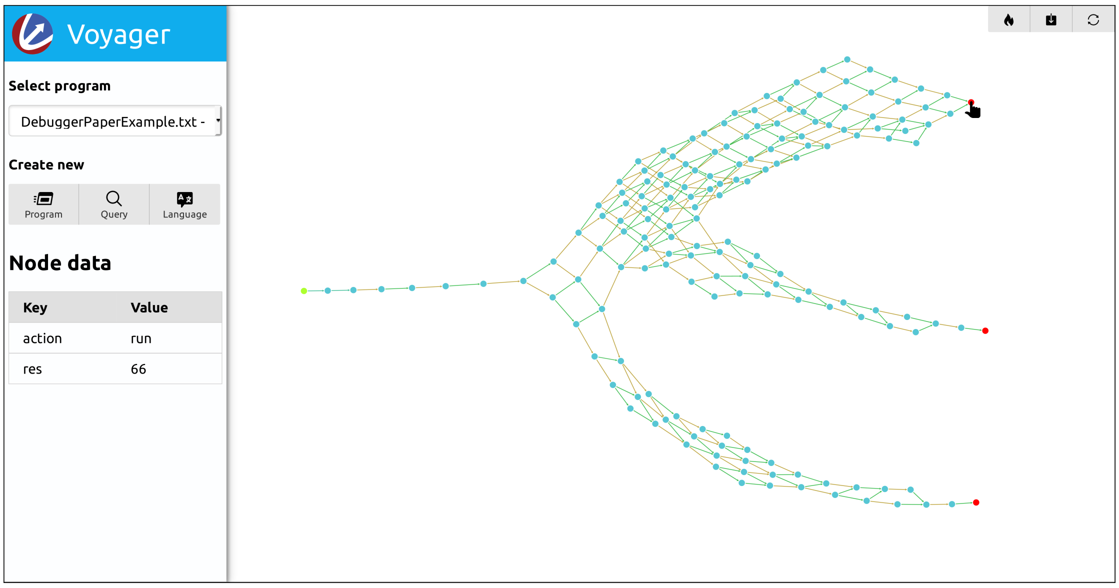
Listing 8.1: Double program containing a bad message interleaving bug in SOMns.

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

Table 8.1: Message interleavings for the program shown in Listing 8.1.

8.2.2 Overview of a Debugging Session

Taking the faulty interleaving of our example program of Listing 8.1 as an 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.



Node data

Key	Value
action	run
res	66

Figure 8.5: Overview of the Voyager tool.

Figure 8.5 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 the 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 (see Section 8.2.1), there are three possible end states. "Node Data" shows the information for the end state under the cursor. The selected state corresponds

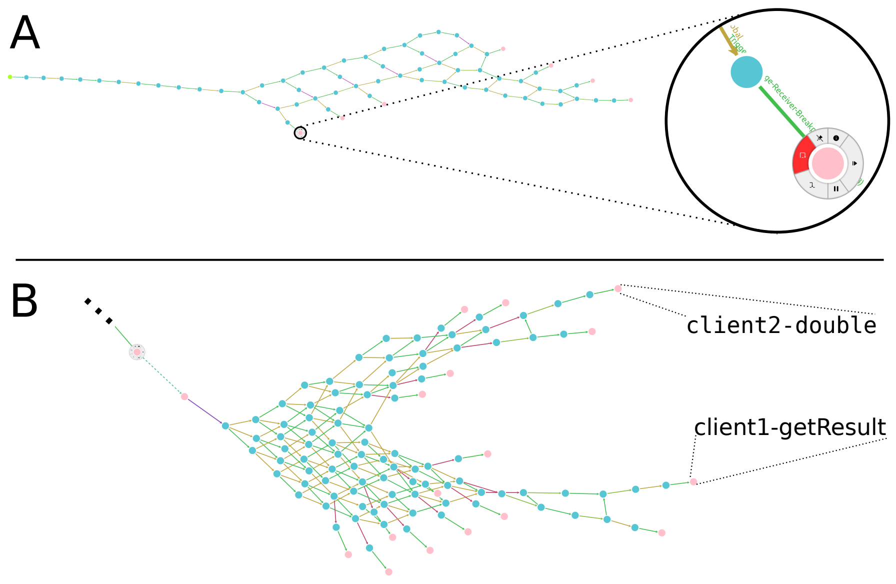


Figure 8.6: A debugging session in Voyager for the program displayed in listing 8.1.

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 calculus, this is called a *message receiver breakpoint*; its semantics are shown in Section 8.3.2. With this breakpoint activated, we run the program again in Voyager.

Figure 8.6(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 8.3.2). 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 8.6(b) shows the resulting graph after applying that stepping command. The first pink node in Figure 8.6(b) corresponds to the node shown with the magnifying glass in Figure 8.6(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.

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 Figure 8.6(b), we see that there are two possible kinds of universes the base level program can evolve to. Inspecting the actor's inbox[4] 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.

Furthermore, in order to steer the state exploration, our colleagues from University of Ghent added query facilities to Voyager, based on computing the shortest path to all end states [LSM+19]. Figure 8.7 (a) shows the original graph for the program. Figure 8.7(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.

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.

8.3 Voyager Calculus, an Implementation of a Multiverse Debugger for Actor-based Programs

In this section, we describe how we implemented a multiverse debugger for actor-based programs. First, we used the operational semantics of the AmbientTalk language (step 1

[4]We consider the inbox as a synonym of the mailbox, i.e., the queue that stores the messages the actor receives.

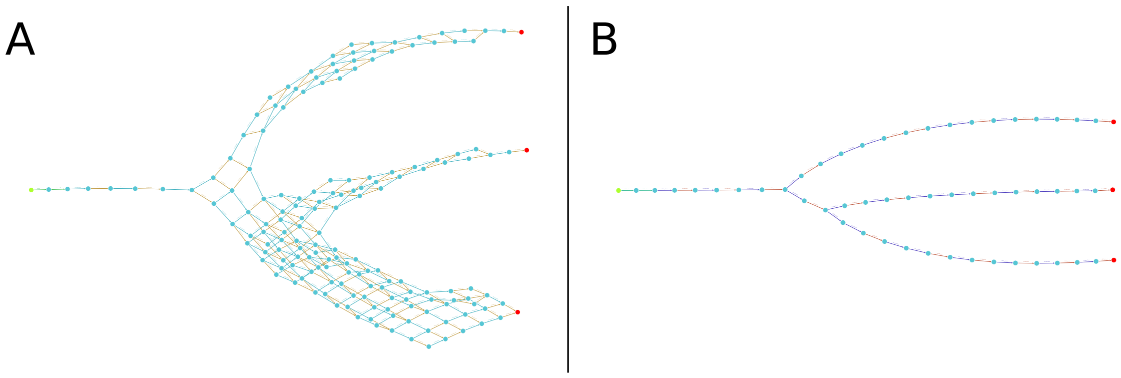


Figure 8.7: Application of a shortest-path-to-end-states query to the program displayed in Listing 8.1.

of our multiverse debugging recipe, section 8.1.1)). Then, we defined Voyager calculus, a debugger configuration and an operational semantics for our multiverse debugger (step 2 of our multiverse debugging recipe).

8.3.1 Syntax and Operational Semantics of the AmbientTalk Language

As we mentioned earlier, we employ the semantics of the AmbientTalk language, i.e., the *Featherweight AmbientTalk* (AT^f) semantics [VCGBS$^+$14] which formalizes common features of the CEL model such as actors, objects, blocks, promises (non-blocking futures in AmbientTalk terminology) and asynchronous message sending. Non-determinism in the CEL model is exhibited in the order in which actors process messages. Promises also introduce additional non-determinism as messages are sent to promises, while promises 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 [IPW01], a minimal core calculus for Java and GJ [BOSW98], only has 10 evaluation rules, we believe that the AmbientTalk semantics should 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. [VCGBS$^+$14] for the complete semantics. In a nutshell, AT^f specifies that actors evaluate messages as expressions to obtain a result value.[5] 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 are no intermediate execution steps. This is useful because it is possible to get all possible states of the evaluation of a non-deterministic program.

In the following description, we keep the AmbientTalk terms of future and inbox queue. We consider these terms indistinctly from promise and mailbox.

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

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

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

Figure 8.8: Semantic entities of the AT^f calculus.

Figure 8.8 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 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 $e \leftarrow m(\overline{e})$.

We needed to extend the AmbientTalk semantics shown in Figure 8.8 for Voyager. In particular, Figure 8.9 shows how we extended two elements of the semantics.

- 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 8.9: Extended semantic entities in AT^f for debugging in Voyager calculus.

8.3.2 Syntax and Operational Semantics of the Voyager Debugger

In this section, we 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 Communicating Event-Loops programs, and we then detail the elements of the *debugger configuration* of Voyager (step 2a of the recipe in Section 8.1.1) and the key mechanisms for supporting *breakpoints and stepping operations* as the one described in the previous section (step 2b of the recipe in Section 8.1.1).

8.3.2.1 Syntax

The Voyager debugger keeps track of both the state of the underlying target 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 described in Chapter 5, 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 to 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.

$d \in$ **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 | 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 8.10: Semantic entities of the Voyager calculus.

Figure 8.10 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. $\mathcal{C}\langle c, \mathit{step}\ n\rangle$.
- 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* or *pause* 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 (cf. Table 5.1).
- 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.

8.3.2.2 Operational Semantics

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 (see 8.3.2.2.1)
2. Reduction rules for breakpoints (see 8.3.2.2.2), including rules needed to model breakpoints which require trigger breakpoints for their functioning.
3. Bookkeeping reduction rules (see 8.3.2.2.3), 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 (see 8.3.2.2.4), 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 (see 8.3.2.2.5), i.e., rules that will resume and pause the program's execution.

8.3.2.2.1 Connection with the Base Level Language Recall that the semantics of a multiverse debugger is defined in terms of the underlying base language semantics, $\textsc{at}^f$ in the case of Voyager. Two reductions rules (shown below) manage the connection of Voyager's semantics with $\textsc{at}^f$: 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 controls all the actor transition 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 $\overset{*}{\rightarrow}_a$. This reduction relation applies one or more single-step local reductions, 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 $\textsc{at}^f$, 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)

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

(CEL-STEP-LOCAL)

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

8.3.2.2.2 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*, defined in Table 5.1, and we explained in the debugging session shown in section 8.2.

It halts the execution of an actor before it processes a message, which is 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 controls 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 to 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, i.e., the actor id $\iota_{a'}$ 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 controls the semantics of the trigger breakpoint added for a message receiver breakpoint. Back in the example debugging session, Figure 8.6 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 change their 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. Then, to enable the message receiver breakpoint, we need two operations, first saving the information needed when the message is about to be sent and triggering the breakpoint. These operations are declared in the reduction rules SAVE-MRB and TRIGGER-MRB.

(TRIGGER-MRB)

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

If the user would like to define a message sender breakpoint, i.e., to halt the program before the message is sent, there is no need to save additional information, as we did for the message-receiver breakpoint. The information we need is available at the current evaluation of the actor configuration K. To trigger a message sender breakpoint, there needs to be an actor in the configuration K that is about to send a message. The id of the message send operator ι_i needs to be the same as the identifier of the breakpoint corresponding to the message sender. When this is the case, the breakpoint is removed from the list of pending breakpoints B_p. Note that a breakpoint at the sender side of the message sent reduces the debugger to a *pause* state just after the message is created. The sender actor change its state from *run* to *pause*. The reduction rule that expresses the semantics for a message sender breakpoint is TRIGGER-MSB.

(TRIGGER-MSB)

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

8.3.2.2.3 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 NOT-APPLICABLE-BREAKPOINT which should be triggered when the breakpoint at the head of the list is not applicable.

(NOT-APPLICABLE-BREAKPOINT[TRIGGER-MSB,SAVE-MRB,TRIGGER-MRB])

$$\frac{not - applicable - breakpoint}{\mathcal{D}\langle \mathcal{B}\langle t_{ub}, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle t_{ub}, \iota_i\rangle, \text{run}, C, A_s, K\rangle}$$

The reduction rule ADD-NEW-ACTOR is declared for the creation of new actors. This rule basically updates the A_s map when an actor is created, with a pair for the actor state, which consists of the new actor id ι_{new} and *run* state.

(ADD-NEW-ACTOR)

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

8.3.2.2.4 Reduction Rules for Stepping Operations Similar to the formalization of the message sender breakpoint, some stepping commands need to be encoded with several reduction rules. For example, the step to next turn command employed in debugging session in section 8.2, is formalized 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 (see CEL-STEP-LOCAL in 8.3.2.2.1).

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. Note that other breakpoints and stepping commands can be encoded in a similar way as we have shown for the message receiver breakpoint and step to next turn.

(PREPARE-STEP-NEXT-TURN)

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

(TRIGGER-STEP-NEXT-TURN)

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

8.3.2.2.5 Reduction Rules for Basic Debugging Commands Finally, we show below the rules for 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)

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

(PAUSE-EXECUTION)

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

8.4 Discussion

There are a number of design decisions and limitations worth discussing. First, in our early prototype of the debugger semantics [TLGBS+17], 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 [TLGBS$^+$17], while in the final version reported here, we are using a multi-step relation. As previously mentioned, in the actor model, the only point where non-determinism matters are 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 a 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 points. We believe that with the current hardware evolution of multicore machines, the size of programs that 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. Of course, applying multiverse debugging to industrial-strength languages will also require further work. But the goal would be that a traditional breakpoint-based debugger can be a foundation for such multiverse debugging. Recently, the operational semantics we proposed here was extended with remote debugging constructs to facilitate debugging of WebAssembly programs for Arduino microcontrollers [SLM$^+$19].

8.4.1 Static Analysis and Multiverse Debugging

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. In Section 3.3.4, we provide an overview of such techniques, with a focus on actor-based approaches. Here we compare them to multiverse debugging.

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 (see Section 8.1.3), 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 [BCCZ99]. 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 counterexample guided abstraction refinement [CGJ+01]. In short, the main difference of model checking with our approach is that the user knows in advance what is the property to be checked for a given program state. Our approach addresses the case when the user does not have any information regarding the bug, but the user knows there is a bug in the program.

Like multiverse debugging, the technique of 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 execution 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.

8.5 Proof of Non-Interference for a Multiverse Debugger

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-hand side 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-hand side 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 into 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 the running or in the paused state. By assumption, when the debugger configuration is in a pause state, the commands C will un-pause (i.e., resume) 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 a 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.

8.6 Related Formal Specifications for Debugging

We now compare Voyager to existing formalization efforts of debugging techniques.

The first formal specification for debuggers was proposed by Da Silva [da 92]. 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. [BS95] 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. [FT01] 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 that they represent using Petri nets. In a later work, Ferrari et al. [FGST08] 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 a few additional insights into the program behavior.

In the context of algorithmic debugging, Luo et al. [LC06] proposed a formal model of tracing for functional programs. The authors proved the 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. [CMMRT18] 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. [LLL14] 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. [GLM14] 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. [LNPV18] proposed a causal consistent reversible debugger for Erlang processes. More concretely, they use a reversible semantics for Erlang [NPV16], 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. [VMV17] 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.

8.7 Conclusion

In this chapter, 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 thesis, 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.

Chapter 9

Conclusion and Future Work

This dissertation presented novel interactive debugging techniques for identifying the root cause of concurrency bugs in actor-based programs. This chapter concludes the dissertation by revisiting our research goals and our contributions. We also discuss some limitations of our approach, and we mention paths for future research.

9.1 Research Goals Revisited

We have stated our research goals in Section 1.3, here we discuss to which extend they have been achieved.

We investigate which kinds of concurrency bugs appear in actor-based programs Chapter 2 and Chapter 3 approached this goal. We did a study of bugs reported in the literature, and we created a taxonomy of concurrency bugs for actor-based applications. Our taxonomy consists of two groups of categories, lack of progress issues and message protocol violations. In the first group, we found communication deadlocks, behavioral deadlocks, and livelocks. In the second group, we found message order violations, bad message interleavings, and memory inconsistencies. Based on this taxonomy, we analyzed the state of the art of techniques to handle concurrency bugs, including static techniques, testing and debugging techniques. The insights from that study was used to improve debugging support for actor-based programs.

We investigate debugging techniques for actor-based programs to find the root cause of concurrency bugs Chapter 5 and Chapter 6 approached this goal. We proposed a set of advanced online debugging techniques based on catalogs of breakpoints, and stepping operations at the level of messages, which can be combined with sequential operations. We designed several visualizations to improve the understanding of actor-based programs, including (1) a graph visualization based on space-time diagrams that shows the happened-before relation of messages in an actor-based program, (2) actor mailbox visualization, and (3) an asynchronous stack trace which combines information at message level (e.g., asynchronous messages sent, and promise resolutions) and sequential method activations. Finally, we extended the Kómpos protocol with new debugging messages for online debugging and new trace events to enable the implementation of the proposed visualisations. We prototyped our novel techniques in Apgar, a message-oriented debugger for the SOMns language and we evaluated them through a user study. We concluded that the online debugging techniques proposed, combined with our trace-based visualizations, may be helpful in the context of actor-based programming to identify concurrency bugs.

We investigate interactive debugging techniques which do not suffer from probe-effect Chapter 8 approached this goal. We proposed multiverse debugging as a novel technique that allows developers explore all the possible states of an actor-based program. We defined the basic recipe for defining the semantics of a multiverse debugger in terms of the base language semantics. We implemented Voyager calculus as a proof of concept debugger of a multiverse debugger for actor-based programs. We proved that the debugger does not interfere with the program behavior and vice versa. We concluded that multiverse debugging establishes a foundation for building online debugging tools which are probe-effect free for actor-based programs.

9.2 Restating the Contributions

In this section, we summarize and restate our contributions.

- We have proposed **a taxonomy of concurrency bugs for actor-based programs** (see Chapter 2). To the best of our knowledge, this is the *first taxonomy of concurrency bugs proposed for actor-based programs.* The taxonomy consists of two main categories: lack of progress issues and message protocol violations. We defined three subcategories of lack of progress issues, i.e., communication deadlock, behavioral deadlock, and livelock. In the category of message protocol violations, we also defined three subcategories, i.e., message order violation, bad message interleaving, and memory inconsistency. Also, we created a catalog of 24 bugs that we found in the literature of tools to find concurrency bugs in actor-based programs. In our study, we classified those bugs according to our taxonomy, we specify the bug patterns that cause the failure in the program, and we described the observable behavior of each of them. This taxonomy not only has provided a new understanding of the kinds of bugs that can be seen in actor-based programs, but it has planted the seeds for further studies of concurrency bugs in programs written with mainstream actor languages. We conclude that our taxonomy will give developers the proper knowledge to build better tools for identifying and solve concurrency bugs.

- We have designed and implemented **interactive debugging techniques for actor-based programs** (see Chapter 5 and Chapter 6). First, we designed a breakpoint catalog that allows developers to pause the program's execution at four different halting locations in the context of an asynchronous message send. Second, we designed a stepping catalog that allows developers to inspect actor state interactively at the level of messages, promises, turns and can be *combined with sequential stepping.* Third, we designed visualizations to enhance the inspection of the actor state and the inspection of message causality information in the debugged program. Fourth, we designed an asynchronous stack trace to show control and data flow information. Finally, we extended the Kómpos protocol to enable the implementation of online debugging features (such as pause and resume activities) and novel visualizations. We conclude that the combination of message-oriented breakpoints and stepping operations with sequential operations, together with the novel visualizations for causality and asynchronous stack traces, can assist developers in identifying lack of progress issues and message protocol violations. Since the extensions made to the Kómpos protocol are concurrency agnostic, we consider that they enable the implementation of similar visualizations for other concurrency models.

- We conducted **a user study to evaluate the proposed advanced debugging techniques for actor-based programs** (see Chapter 7). Our study gathered 28 participants (in two groups) who solved debugging assignments using our novel debugging techniques. Although we cannot generalize our user study results (because time measurements were not statistically significant), we obtained positive assessments regarding participants' perception of the debugging techniques. From our results, we can conclude that the advanced debugging techniques proposed may be helpful in the context of developing actor-based programming. Participants valued message breakpoints and stepping operations and the combination of sequential and message-oriented stepping. Using the data correlation between participants' expertise and the quantitative and qualitative measures, we derived that the expertise of developers of actor-based programs is important to use advanced debugging techniques. To the best of our knowledge, our user study is the first one conducted using a *mixed methods experimental research design* approach for validating debugging techniques.

- We introduced **multiverse debugging as a novel online probe-effect free technique for debugging actor-based programs** (see Chapter 8). This is the first online debugging approach that tackles the non-determinism problem in concurrent programs. Multiverse debugging enables developers to interactively explore the space of *all* non-deterministic execution paths of a concurrent program. We provided a recipe for building multiverse debuggers based on a debugger configuration and its operational semantics in terms of the base language semantics. Besides, we have proved that observing the program behavior with the debugger does not affect the target program and vice versa, i.e., our approach is *probe-effect free*. We conclude that this technique can simplify the reproduction and inspection of bugs because it removes chance and probability from the equation of hitting the problematic scheduling of messages while offering online features for exploring all possible paths interactively.

9.3 Discussion

In this section, we discuss some limitations of the debugging techniques proposed in this dissertation.

Concurrency Bugs and Apgar To provide better debugging support for identifying lack of progress issues and message protocol violations, we have proposed a set of interactive debugging techniques in our research. Apgar addresses concurrency bugs that can be seen in the CEL concurrency model, i.e., behavioral deadlocks, livelocks, bad message interleavings, and message order violations. Since CEL does not feature blocking operations, further experiments are needed to validate if Apgar can help to find the root cause of bugs such as communication deadlocks. It may still be difficult in Apgar to debug applications that suffer from bugs caused by timing issues. For this kind of time-dependent bugs, we think multiverse debugging could be helpful because it can show wrong schedulings of concurrent operations. Furthermore, while Apgar incorporates offline features such as visualization of message causality, bugs in which the distance between the root cause and the failure is long can benefit from other techniques such as reverse debugging (see Section 9.4).

Apgar Frontend From the results we obtained in our user study, we identified some technical limitations in the current Apgar prototype, which may have influenced the perception of the features users used during the experiments. For example, we would like to explore alternative graphic frameworks for the turns visualization to search for better layouts that avoid the message's arrows overlapping. In the asynchronous stack trace implementation, a code revision needs to be done regarding the data flow of promises resolution and the actors' id shown by frames. Regarding the visualization of the asynchronous stack trace, we consider it relevant to incorporate the variables state for the asynchronous frames, i.e., asynchronous messages sent and promise resolution frames. Besides, we do not explicitly show a list of unresolved promises. We only show the state for a promise in the Variables view and in the Mailbox view. This is a limitation we would like to address also as future work, starting with the insights we obtained from our experiences applying interrogative debugging techniques. Moreover, for future work, we would like to test our proof of concept debugger in more complex actor-based programs.

User Study Experiments As we concluded in Chapter 7 we cannot generalize our results because time measurements were not significant statistically. For future user study experiments, we would consider gathering participants only with scores 2 and 3, i.e., with intermediate and advanced knowledge in concurrent programming, mainly with experience with the actor model. We consider that executing the experiments with expert participants will provide better time results because from the correlations of the expertise of participants of our study and the quantitative and qualitative measures, we obtained better results in the assessments of Apgar debugging features made by expert participants. Also, one or two days before the experiment, we recommend giving participants a lab session demonstration about the debugging features, in which they can interact with the tool in advance. We think this is important because, as they referred in their comments, some participants expressed their lack of experience with the tool makes them perform the assignments in more time.

Practical Multiverse Debugging The main open research question of our multiverse debugging technique is how to make it practical for complex concurrent applications. Research is needed to guide the exploration of the state graph, e.g., novel stepping semantics that works at the level of universes. First, we would like to extend the Voyager calculus with the whole catalog of breakpoints and stepping operations from Section 5.1.1 and Section 5.1.2. Besides, we think it would be interesting to include in the semantics tracking the happened-before relation of messages. This can be useful, for example, for implementing advanced stepping on the level of the turns, e.g., stepping to the end of the turn after a message sender breakpoint is triggered. Second, we foresee the investigation of techniques to reduce the exploration of the state space, e.g., to allow developers to filter out non-deterministic paths non-related directly to the fault. Furthermore, in order to make this debugging technique practical, it needs to be applied to a concrete language implementation beyond a PLT-Redex-based formalism. In particular, we aim to investigate techniques that allow us to integrate multiverse debugging in an IDE visualization. Also, it will be interesting to consider using the semantics to prove other properties of the debugging operations, e.g., to prove the correctness of the stepping commands.

9.4 Future Work

Here we present ideas derived from our research that we think can be extended further.

Language and Concurrency-agnostic Debugging APIs In this work, we experimented with the implementation of a debugger as part of a self-optimizing AST-interpreter on the Truffle/GraalVM platform. As we mentioned in Section 4.4 currently, Truffle Debug API is language-agnostic, but it does not provide support for concurrency. Our work has overcome this by using Truffle Instrumentation API to create wrapper nodes and implementing additional stepping strategies based on node metadata. Wrapper nodes allow access and update the state of breakpoints defined for certain nodes in the program AST. And stepping strategies use syntactic tags and source coordinates to tell Truffle sequential debugger until which AST node resume execution. We think that the implementation strategies we have followed to enable breakpoints and stepping operations at the level of messages could be used as a foundation to support language-agnostic tools for concurrent languages build on top of Truffle. However, further investigation is needed to abstract breakpoints and stepping semantics from different concurrency models and integrate them in Truffle Debug API. Our work and the work on the Kómpos concurrency-agnostic protocol [MLA$^+$17], could serve as starting point to leverage Truffle debugging support for concurrent languages.

Offline Debugging We aim as future work to incorporate offline features in Apgar such as reverse debugging. As we mentioned before (Section 3.3.2), recent works have explored reverse debugging in the context of actor-based programs [BMM$^+$16, VBMM18, MOM18]. Our catalogs of breakpoints and stepping operations have not been designed for backward step-by-step execution. We would like to investigate support for reverse debugging in combination with our catalogs and novel visualizations. Considering our stepping catalog, an idea could be to step backward from the points' locations proposed in Figure 5.1, i.e., from point 4 to 3, from 3 to 2, and from 2 to 1, and ideally continue forward when requested by the developer. To achieve this, we will need to investigate how possible it is to obtain state information from the snapshots and enable developers' interaction with the debugger in an offline mode without losing the previous online debugging session. Creating snapshots has already been investigated for SOMns programs [AMGBM19] which could be integrated with Medeor and Apgar. However, it remains to be seen how to enable efficiently and seamlessly backward and forward step-by-step execution. Replaying snapshots will require a decoupling of the two debugging sessions, i.e., online and offline. In particular, we will need to investigate, if additional Truffle instrumentation is needed at the level of AST nodes to enable the offline debugging session and allowing interactive backward stepping. Furthermore, enabling the actor's state inspection and modification in the past requires invalidating (subsets of) snapshots and re-executing parts of the program.

9.5 Concluding Remarks

In the last two decades, computers have changed our world at a fast pace. There has been indeed a revolution for concurrent software, as Herb Sutter foresaw 16 years ago. However, tools support for concurrent software has emerged at a slow pace. This dissertation aims to push forward the development of debugging tools for identifying concurrency bugs in actor-based programs with novel techniques.

We first have contributed a taxonomy of concurrency bugs that aims to give developers a better understanding of the issues that can occur when programming actor-based applications.

We have then designed and implemented advanced debugging techniques based on the combination of online and offline techniques that allows developers to interactively explore an actor-based program execution. Evaluation of our proof of concept debugger shows that the proposed debugging techniques may be beneficial to aid expert developers in finding concurrency bugs and comprehend the program behavior.

Finally, we have proposed a new debugging technique based on a formal operational semantics that focuses on exploring all non-deterministic paths of an actor-based program execution interactively. Proof of non-interference for our operational semantics shows that any evaluation step in the execution of our debugger is equivalent to an evaluation step in the target program, and the debugger observes any step of the target program without interference.

Although our tools are still research prototypes, we consider our contributions the foundation for future debugging tools for actor-based applications.

Appendices

Appendix A

Catalog of Bugs Found in Actor-based Programs

A.1 Catalog of bugs found in actor-based programs

Bug Type	Id	Bug Pattern	Observable Behavior	Source Reporting the Bug	Language
Message order violation	bug-1	incorrect execution order of two processes when registering a name for a pid in the Process Registry	runtime exception	Fig. 1 in [CS10]	Erlang
Memory inconsistency	bug-2	insert and write in tables of Erlang Term Storage with public access	inconsistency of values in the tables	Fig. 2 in [CS10]	Erlang
Memory inconsistency	bug-3	insert and write in tables (dirty operations in Mnesia database)	inconsistency of values in the tables	Fig. 2 in [CS10]	Erlang
Communication deadlock	bug-4	receive statement with no messages	process in waiting state due to an orphan message	Fig. 1 in [CS11b]	Erlang
Memory inconsistency	bug-5	testing insert operations in parallel (Mnesia database)	exception or inconsistent return values	Sec. 5 of [HB11]	Erlang

Memory inconsistency	bug-6	testing open_file in parallel with other operations of dets API (Mnesia database)	inconsistency when visualizing the table's contents	Sec. 5 of [HB11]	Erlang
Memory inconsistency	bug-7	open, close and reopen the file, besides running three processes in parallel (Mnesia database)	integrity checking failed due to premature_eof error	Sec. 5 of [HB11]	Erlang
Memory inconsistency	bug-8	changes in the dets server state	integrity checking failed (Mnesia database)	Sec. 5 of [HB11]	Erlang
Communication deadlock	bug-9	receive statement with no messages	process in waiting state due to an orphan message (server waits for ping requests)	Program 2 and Test code 2 in [GCS11]	Erlang
Communication deadlock	bug-10	message sent to a finished process, the finished process exit without replying	process blocks due to an orphan message	Test code 5 in [GCS11]	Erlang
Message order violation	bug-11	spawned process that terminates before its Pid is register by the parent process	process will crash and exits abnormally due to an orphan message	Fig. 1 in [CGS13b]	Erlang
Bad message interleaving	bug-12	actor execute a third message between two consecutive messages	inconsistent values of variables	Fig. 2 in [LDMA09]	ActorFoundry, Scala
Message order violation	bug-13	incorrect order of execution of two message receives	the program throws an exception because of a null value	Listing 1 in [TPLJ13]	Scala
Message order violation	bug-14	the second message is executed with the value of the first message	actions are performed over the wrong variable	Fig. 4 in [ZBZ11]	JavaScript

Bad message interleaving	bug-15	use of a variable not initialized by other methods before it was defined	out of bounds exception	Fig. 4 in [ZBZ11]	JavaScript
Message order violation	bug-16	race between HTML parsing and user actions	application crash	Fig. 1 in [RVS13]	JavaScript
Message order violation	bug-17	race between execution of a script and rendering of an input text box	inconsistency in the value of the variable (storing text the user entered)	Fig. 2 in [PVSD12]	JavaScript
Message order violation	bug-18	race between creation of HTML element and using the element	throw an exception that can lead the application to crash	Fig. 3 in [PVSD12]	JavaScript
Message order violation	bug-19	invocation of a function before parsing of the same function	application crash	Fig. 4 in [PVSD12]	JavaScript
Message order violation	bug-20	iframe's load event fires before the script executes	event handler will never run	Fig. 5 in [PVSD12]	JavaScript
Bad message interleaving	bug-21	execution of an operation (changing the workspace) between two other operations (starting the file transmission and the completion of the transmission)	exception of variable undefined	Fig. 6 in [HPK14]	JavaScript
Bad message interleaving	bug-22	event handler updates DOM between two input events that manipulate the same DOM element	error because of a null value	Fig.3 in [HPK14]	JavaScript
Message order violation	bug-23	user input invokes a function before it has been defined/loaded	application crashes (due to unexpected turn termination)	Fig. 2 in [HPK14]	JavaScript

Bad message interleaving	bug-24	interleaving of callbacks	undefined or overwritten variable	Fig.2 in [CDG+19]	Node.js

Table A.1: Catalog of bugs found in actor-based programs

Appendix B

SOMNS Cheat Sheet

This section includes a summary of the main elements of the SOMNS programming language given to the participants of our user study.

Collections

Array

```
(* Array is passed by far reference to
    other actors *)
numbersArray1:: Array new: 10.
(* TransferArray is passed by copy to other
     actors *)
numbersArray2:: TransferArray new: 10.
(* ValueArray denotes an immutable array *)
numbersArray3:: ValueArray new: 10 withAll:
     [:i | i*i].

(* all types of arrays have the same API *)
1 to: 10 do:[:i | numbersArray1 at: i put:
    i.].
numbersArray1 at: 1   ⟶   1
numbersArray1 size   ⟶   10
```

Vector

```
studentsVector:: Vector new: 10.
studentsVector append: 'Joe'.
(studentsVector at: 1) println.   ⟶   Joe
(* iterating *)
studentsVector do: [:s | ('Student ' + s)
    println.].
studentsVector doIndexes: [:i |
  ('Student '+ (studentsArray at: i))
    println.].
```

Dictionary

```
dictionary := Dictionary new: 10.
dictionary at: 'somns' put: 80.
dictionary containsKey: 'somns'   ⟶   true
dictionary at: 'somns'   ⟶   80
```

4. Concurrency

Actor Definition

```
(* createActorFromValue message creates an
 actor  from Math value; it returns a far
reference to the actor Math *)
mathFarRef:: (actors createActorFromValue:
Math).
(* new message creates a new instance of
the Math actor *)
mathActor:: mathFarRef <-: new.
```

Implicit Promises

```
result:: mathActor <-: division: 27 and: 5.
(* Registering a callback for a promise;
    whenResolved: is applied when the result
     is available, onError: when an error
    happens; onError: is optional*)
result whenResolved:[:div |
  ('Division result: '+ div) println.
] onError:[:error |
  ('DivisionZeroError' + error) println. ].
```

Promise Group

```
 squareA:: mathActor <-: square: sideA.
 squareB:: mathActor <-: square: sideB.
 (* registers a promise for a group of
   promises stored in a table *)
 squareA, squareB whenResolved:[
   :squaresVector | ... ].
  ,   ⟶   concat. operator returns a table
```

Explicit Promises

```
(* explicit promise creation *)
promisePair:: actors createPromisePair.
(* resolves the promise with a value *)
promisePair resolve: perimeter.
(* resolves the promise with an error *)
promisePair error: e.
(* accessing the promise object *)
promisePair promise
(* accessing the resolver object *)
promisePair resolver
```

References

1. SOMns: https://github.com/smarr/SOMns
2. Setup guide: https://somns.readthedocs.io/
3. Sample programs: https://github.com/ctrlpz/somns-sample-programs
4. The standard language library is accessible in the SDK of the project opened in IntelliJ: *core-lib*.

This cheat sheet has been adapted from the Smalltalk one at http://sdmeta.gforge.inria.fr/Teaching/0809Turino/st-cheatsheet.pdf

SOMns Cheat Sheet

Software Languages Lab
Vrije Universiteit Brussel
November 2020

1. The SOMns IntelliJ plugin

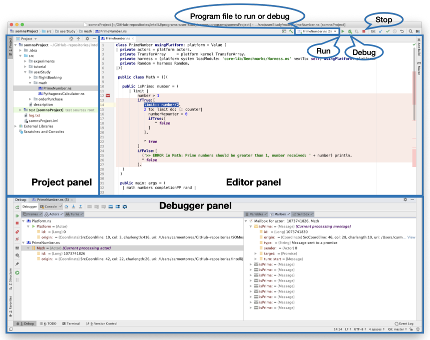


Figure 1: The SOMns IntelliJ plugin

Run it: (CTRL+FN+SHIFT+F10) Evaluate selected .ns file.

Debug it: (CTRL+FN+SHIFT+F9) Evaluate selected .ns file step-by-step with the integrated debugger.

Stop it: (CMD+FN+F2) Stop program's execution, in run or debug mode.

2. The SOMns Language

- Class-based OO inspired by Smalltalk: everything is an object. Everything happens by sending messages.
- Communicating Event-Loops actor model.
- Messages between objects within the same actor are sent synchronously and return a promise.
- Messages between objects in different actors are sent asynchronously.

Keywords

- `self`, the receiver.
- `super`, the receiver, method lookup starts in superclass.
- `nil`, the unique instance of the class `Nil`.
- `true`, the unique instance of the class `True`.
- `false`, the unique instance of the class `False`.

Literals

- Integer: `123`
- Double: `123.4`
- Boolean: `true`, `false`
- String: `'abc'`
- Symbol: `#ok`
- Array:

```
obj:: Object new.
array:: { nil. false. #rr. obj }.
(array at: 1)   ⟶   nil.
(array at: 2)   ⟶   false.
(array at: 3)   ⟶   rr.
(array at: 4)   ⟶   instance of Object.
```

Message Sends

1. *Unary messages* take no argument.
   `25 sqrt` sends the message `sqrt` to the object `25`.
2. *Binary messages* take exactly one argument.
   `3 + 4` sends message `+` with argument `4` to the object `3`. Binary selectors are built from one or more of the characters `+`, `-`, `*`, `=`, `<`, `>`, etc.
3. *Keyword messages* take one or more arguments.
   `2.0 pow: 6.0` sends the message named `pow:` with argument `6` to the object `2`.

Unary messages are sent first, then binary messages and finally keyword messages:

```
2.0 pow: 2 + 16 sqrt   ⟶   64
```

Messages are sent left to right. Use parentheses to change the order:

```
1 + 2 * 3   ⟶   9
1 + (2 * 3)   ⟶   7
```

Syntax

- Comments
  `(* Comments are enclosed in parentheses and asterisks *)`
- Temporary variables
  `| var1 var2 |`
- Mutable variable declaration
  `var ::= aStatement`
- Immutable variable declaration
  `var = aStatement`
- Variable assignment
  `var:: aStatement`
- Statements
  `aStatement1. aStatement2`
- Synchronous messages
  `receiver message` (unary msg)
  `receiver + argument` (binary msg)
  `receiver message: argument` (keyword msg)
  `receiver message: arg1 with: arg2`
- Asynchronous messages
  `receiver <-: message` (unary msg)
  `receiver <-: message: arg` (keyword msg)
  `receiver <-: message: arg1 with: arg2`
- Blocks
  `[aStatement1. aStatement2]`
  `[:argument1| aStatement1. aStatement2]`
  `[:arg1 :arg2| | temp1 temp2 | statement]`
- Return statement
  `^ aStatement`
- Main class definition

```
public class MainClassName usingPlatform:
     platform = Value (
  | slots |
)(
 (* classes definitions and method
    definitions *)

  public main: args = ( ^ (* returns an
    integer as error code or a promise
    for program completion *) )
)
```

- Class definition

```
public class ClassName new: parameter1
    param2: parameter2 = (
  | slots |
)( body )
```

- Method definition

```
messageSelectorAndArgumentNames = (
(* comment stating purpose of message *)
      | temporary variable names |
      statements )
```

3. Standard Classes

Logical Expressions

```
true not   ⟶   false
1 = 2 or: [ 2 = 1 ]   ⟶   false
1 < 2 and: [ 2 > 1 ]   ⟶   true
```

Conditionals

```
1 = 2 ifTrue: [ '1 is equal to 2' println.]
1 = 2 ifFalse: [ '1 is not equal to 2'
   println.]
```

Loops

```
(* conditional iteration *)
[ student notNil ] whileFalse: [ 'student
   nil' ]
[ student notNil ] whileTrue: [ (student
   name) println.]

(* fixed iteration *)
sum:: 0.
100 timesRepeat: [
  sum:: sum + 1. ].

(* another fixed iteration *)
1 to: 100 do: [ :index | index println. ].
```

Blocks (anonymous functions)

```
[ 1 + 2 ] value   ⟶   3
[ :x | x + 2 ] value: 1   ⟶   3
[ :x :y| x + y ] value: 1 value: 2   ⟶   3
```

Appendix C

Sample Programs in SOMNs

C.1 Prime number

The program creates one Math actor and one Platform actor. The Platform actor randomly computes 10 numbers, which are sent to the Math actor, to check if they are prime numbers or not. The Platform actor prints the result. Figure C.1 shows a conceptual representation of this program.

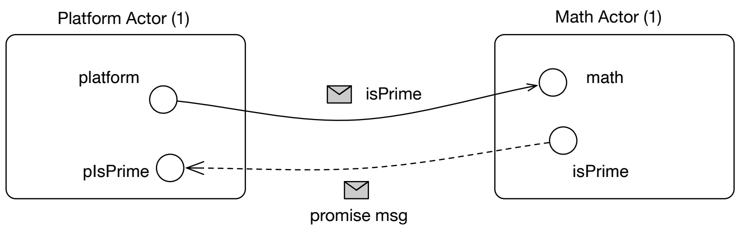


Figure C.1: Conceptual diagram of the prime number program. For simplicity the `isPrime` message is represented as a message sent to a far reference in the diagram, however, in the implementation shown in listing C.1 this message is sent to promise `math`, the instance created for the Math actor.

```
class PrimeNumber usingPlatform: platform = Value (
| private actors = platform actors.
  private TransferArray     = platform kernel TransferArray.
  private harness = (platform system loadModule:
  'core-lib/Benchmarks/Harness.ns' nextTo: self) usingPlatform: platform.
  private Random = harness Random.
|)(

 public class Math = ()(

   public isPrime: number = (
     | limit |
       number > 1
       ifTrue:[
          limit:: number/2.
          2 to: limit do: [: counter|
            number%counter = 0
            ifTrue:[
              ^ false
            ]
          ].

          ^ true
        ]
        ifFalse:[
          ('ERROR in Math: Prime numbers should be greater than 1,
          number received: ' + number) println.
          ^ false
        ].
    )
  )

  public main: args = (
    | math numbers completionPP rand |

    completionPP:: actors createPromisePair.
    numbers:: TransferArray new: 10.

    1 to: 10 do:[:i |
      rand:: Random new: i + 73425.
      numbers at: i put: (1 + (rand next % 100)).
    ].

    math:: (actors createActorFromValue: Math) <-: new.

    numbers do:[:n |
      | pIsPrime pWR |
      pIsPrime:: math <-: isPrime: n.
      pWR:: pIsPrime whenResolved:[:isPrime |
        isPrime ifTrue:[ ('Number '+ n + ' is prime') println.].
        isPrime ifFalse:[ ('Number '+ n + ' is not prime') println.].

        n = (numbers at: (numbers size))
        ifTrue:[
          completionPP resolve: true.
        ]
      ].

      pWR <-: println.
    ].

    completionPP promise <-: println.

    ^ completionPP promise
  )
)
```

Listing C.1: Implementation of a prime number program in SOMns.

C.2 Instant messenger

The program creates three actors, one instance of `Platform` class and two more actors instances of the `InstantMessenger` class. Each instant messenger actor will host one user and their list of chat buddies. Each user can send simple text messages to a buddy. The instant messenger should only display the message in its own chat window after it has received an acknowledgement from the remote user that the message was successfully received. In short, the protocol is: when two messengers discover one another, they ask one another user names. In this solution both messengers sends a 'Hello' message to each other and acknowledge its reception (`sendMessage` method). A second method should allow the messenger to accept a text message from a remote user (`receive` method). It will print the message to the screen and send an acknowledgement message to the sender user. Figure C.2 shows a conceptual representation of this program.

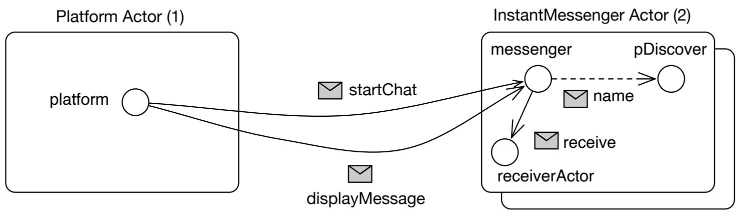


Figure C.2: Conceptual diagram of the Instant messenger program. For simplicity of the diagram we only show one promise message, i.e., `name`. Besides, the `startChat` message is represented as a message sent to a far reference in the diagram, however, in the implementation shown in listing C.2 this message is sent to promise `messenger1` and `messenger2`, which are instances created for the InstantMessenger actor.

```
class InstantMessengerApplication usingPlatform: platform = Value (
| private actors = platform actors.
  private Array = platform kernel Array.
  private Dictionary = platform collections Dictionary.
|)(
  class TextMessage new: content sender: senderName = Value (
  | public content = content.
    public sender = senderName.
  |)()

  public class InstantMessenger new: name  total: size = (
  | private buddyMap = Dictionary new: size.
    public name = name.
    private textMessage ::= nil.
  |)(

    public startChat: remoteMessenger = (
      | pDiscover pName pSend msg pp array |
      pp:: actors createPromisePair.

      (* returns a far reference of the remote messenger *)
      pDiscover:: self addMessenger: remoteMessenger.

      pName:: pDiscover <-: name.
      pName whenResolved: [:remoteName |
        msg:: 'Hello ' + remoteName.
        pSend:: sendMessage: remoteName contentMsg: msg.

        pSend whenResolved: [:result |
          result = #ok
            ifTrue: [
              pp resolver resolve: (TextMessage new: msg sender: name) ]
            ifFalse:[
              pp resolver resolve: nil ] ] ].

      ^ pp promise
    )

    public sendMessage: receiverName contentMsg: content = (
      | receiverActor pReceive |
      textMessage:: TextMessage new: content sender: name.

      receiverActor:: buddyMap at: receiverName.
      pReceive:: receiverActor <-: receive: textMessage.
      pReceive whenResolved:[: r|
        ('Receive message '+ r) println.
      ].

      ^ pReceive
    )

    public displayMessage: msg = (
      (name + ' screen> ' + msg) println
    )

    public addMessenger: messenger = (
      | p pName buddy pWRError |
      p:: actors createPromisePair.

      pName:: messenger <-: name.
      pWRError:: pName whenResolved: [:n |
        buddyMap at: n put: messenger.

        (buddyMap containsKey: n)
          ifTrue: [
            (name + ' updated the far reference to buddy: ' + n) println. ]
          ifFalse: [
            (name + ' discovered a new messenger buddy: ' + n) println ].

        p resolver resolve: (buddyMap at: n).
      ] onError: [:e |
        ('-Error adding the messenger: ' + e) println.
        p resolver resolve: nil.
      ].
      pWRError <-: println.

      ^ p promise
    )

    public receive: textMessage = (
      | sender content |
      content:: textMessage content.
      sender:: textMessage sender.

      self displayMessage: sender + ': ' + content.

      ^ #ok
    )
  )

  public main: args = (
    | completionPP1 completionPP2 users messenger1 messenger2 pResult1
    pResult2 |
    completionPP1:: actors createPromisePair.
    completionPP2:: actors createPromisePair.

    '[INSTANT MESSENGER APPLICATION] Starting' println.

    users:: Array new: 2.
    users at: 1 put: 'Joe'.
    users at: 2 put: 'Marie'.

    messenger1:: (actors createActorFromValue: InstantMessenger) <-: new: (
    users at: 1) total: users size.
    messenger2:: (actors createActorFromValue: InstantMessenger) <-: new: (
    users at: 2) total: users size.

    pResult1:: messenger1 <-: startChat: messenger2.
    pResult1 whenResolved: [:result |
      result ifNotNil: [
        messenger1 <-: displayMessage: result sender + ': ' + result content.
        completionPP1 resolver resolve: 0. (* end application *)
      ] ].

    pResult2:: messenger2 <-: startChat: messenger1.
    pResult2 whenResolved: [:result |
      result ifNotNil:[
        messenger2 <-: displayMessage: result sender + ': ' + result content.
        completionPP2 resolver resolve: 0. (* end application *)
      ] ].

    ^ completionPP1 promise whenResolved: [:result1 |
        completionPP2 promise whenResolved: [:result2 |
          '\n\n[INSTANT MESSENGER APPLICATION] Ending' println ] ].
  )
)
```

Listing C.2: Implementation of an instant messenger application in SOMns.

C.3 Pythagoras calculator

The program creates four actors: one Platform, two Calculator and one Math. Platform actor contains an array of Calculator actors. Each Calculator represents a student assignment, which computes the perimeter of the right triangle knowing only two of its sides. In response to the computePerimeter message the Calculator actor applies the Pythagoras theorem $c = \sqrt{a^2 + b^2}$ to get the longest side of the triangle, i.e., the hypotenuse. The Calculator actor sends square and add messages to the Math actor to compute the sum of the sides square. When the square root of the result is computed by the Calculator actor, the message trianglePerimeter is sent. Program that computes the triangle perimeter given the length of two of its sides. All triangles are considered to be right triangles. Then, we applied the Pythagoras theorem to compute the third side. Figure C.3 shows a conceptual representation of this program.

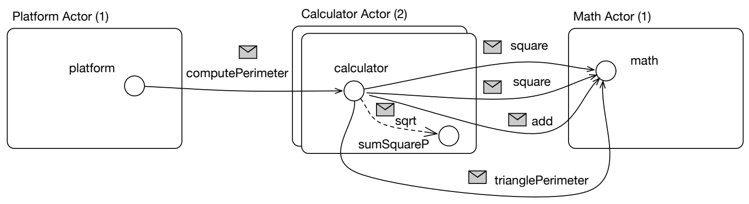


Figure C.3: Conceptual diagram of the Pythagoras calculator program. For simplicity of the diagram we only show one promise message, i.e., `sqrt`. Besides, the `computePerimeter` message is represented as a message sent to a far reference in the diagram, however, in the implementation shown in listing C.3 this message is sent to promise `c`, the instance created for the `Calculator` actor.

```
class PythagorasCalculator usingPlatform: platform = Value (
| private actors = platform actors.
  private Array = platform kernel Array.
  private harness = (platform system loadModule:
  'core-lib/Benchmarks/Harness.ns' nextTo: self) usingPlatform: platform.
  private Random = harness Random.
  private Exception   = platform kernel Exception.
  private numberStudents = 2.
|)(
   public class DivisionZeroError = Exception (
     self signal.
   )(
     public asString = (
       ^ 'DivisionZeroError'.
     )
  )

  public class Math = (
  )(
    public add: x and: y = (
      ^ x + y
    )

    public square: x = (
      ^ x * x
    )

    public trianglePerimeter: a b: b c: c = (
      ^ add: (add: a and: b) and: c.
    )

    public division: x and: y = (
      (y > 0)
      ifTrue:[
        ^ x / y
      ]
      ifFalse:[
        DivisionZeroError signal
      ]
    )
  )

  public class Calculator new: studentId math: math = (
    | private studentId ::= studentId.
      private math = math.
      private rand = Random new: studentId + 73425.
    |
  )(
    public computePerimeter = (
      | sideA sideB squareA squareB perimeterPP |
      perimeterPP:: actors createPromisePair.

      sideA:: 1 + (rand next % numberStudents).
      sideB:: 1 + (rand next % numberStudents).

      squareA:: math <-: square: sideA.
      squareB:: math <-: square: sideB.

      squareA, squareB whenResolved:[:squares |
        | squareSumP hypotenusePromise |
        squareSumP:: math <-: add: (squares at: 1) and: (squares at: 2).
        hypotenusePromise:: squareSumP <-: sqrt.
        hypotenusePromise whenResolved:[:sideC |
          | perimeterPromise |
          perimeterPromise:: (math <-: trianglePerimeter: sideA
                                          b: sideB
                                          c: sideC).
          perimeterPromise whenResolved:[:perimeter |
            ('Student assignment: '+ studentId + ',
            Triangle sides: A = '+sideA+',
            B = '+sideB+ ',
            C = '+sideC + ',
            Perimeter: '+perimeter) println.
            perimeterPP resolve: perimeter.
          ].
        ].
      ].

      ^ perimeterPP promise
    )
  )

  public main: args = (
    | math a b completionPP calculators counter |
    completionPP:: actors createPromisePair.
    math:: (actors createActorFromValue: Math) <-: new.

    calculators:: Array new: numberStudents.
    counter:: 0.

    '[PYTHAGORAS CALCULATOR] Starting...\n' println.

    calculators doIndexes: [:i |
      | c |
      c:: (actors createActorFromValue: Calculator) <-: new: i math: math.
      calculators at: i put: c
    ].

    calculators do: [:c |
      c <-: computePerimeter whenResolved:[:p |
        counter:: counter + 1.
        counter = numberStudents
          ifTrue: [
            completionPP resolve: true.
          ].
      ].
    ].

    ^ completionPP promise whenResolved:[:r |
       '\n[PYTHAGORAS CALCULATOR] Ending.' println.
    ]
  )
)
```

Listing C.3: Implementation of the Pythagoras calculator program in SOMns.

Appendix D

Apgar Implementation Details

D.1 Medeor Implementation Classes

This section details the implementation classes of Medeor, the backend of our proof of concept debugger Apgar.

D.1.1 Debugger Tool in Medeor

To create a debugger tool for Truffle we created the `WebDebugger` class, which extends from `TruffleInstrument` class [1]. As mentioned in Section 4.4.1, `TruffleInstrument` is an interface for Truffle clients that may observe and inject behavior into interpreters written using the Truffle framework, as in our case of the SOMns interpreter. The `WebDebugger` class connects the Truffle debugging features with the debugger frontend using web sockets[2].

The `WebDebugger` class has an instance of the `FrontendConnector` class, which awaits for the web sockets to complete the connection and handles the requests from the debugger frontend. Also, the `FrontendConnector` manages the sending of the protocol messages to the frontend through the web sockets when a suspension occurs. The JSON format is used to encode the messages sent between the debugger frontend and the backend. In the next section, we detail the messages exchanged for debugging.

The `WebSocketHandler` is the class to instantiate a web socket server, for example, Medeor have two web socket instances, implemented as fields in the `FrontendConnector` class, i.e., `messageHandler` and `traceHandler`. The first one to send debugger messages, and the second one to send trace events.

The `Suspension` class controls the interaction between the debugger frontend and the application thread. It provides a mechanism to ask the application thread to perform tasks on behalf of the frontend, e.g., to walk the stack and obtain the relevant data for the debugger. Here we describe the class fields:

- `activityId` identifier of the actor that has been suspended.
- `activity` instance of the suspended actor.
- `activityThread` thread in the fork join pool executing an actor.
- `suspendedEvent` gives access to the state of a guest language execution thread that has been suspended, for example, by a breakpoint or stepping operation.
- `stack` keeps information on the runtime stack of an application thread for requests from the frontend.

[1] `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/TruffleInstrument.html`

[2] WebSockets `https://tools.ietf.org/html/rfc6455`

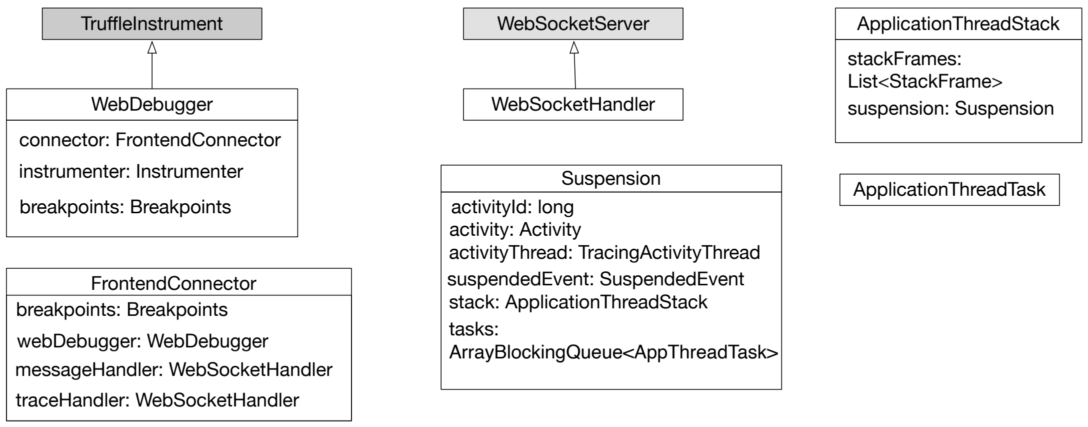

TruffleInstrument
WebDebugger
connector: FrontendConnector
instrumenter: Instrumenter
breakpoints: Breakpoints
FrontendConnector
breakpoints: Breakpoints
webDebugger: WebDebugger
messageHandler: WebSocketHandler
traceHandler: WebSocketHandler
WebSocketServer
WebSocketHandler
Suspension
activityId: long
activity: Activity
activityThread: TracingActivityThread
suspendedEvent: SuspendedEvent
stack: ApplicationThreadStack
tasks: ArrayBlockingQueue<AppThreadTask>
ApplicationThreadStack
stackFrames: List<StackFrame>
suspension: Suspension
ApplicationThreadTask

Figure D.1: Class diagram of the main classes of Medeor. Classes in gray color belongs to the Truffle framework and the WebSocket API.

- `tasks` is a collection of tasks that need to be executed by the application thread while it is in a suspension.

D.1.2 Breakpoints and Stepping

Figure D.2 shows the main implementation classes related to breakpoints in Medeor[3]. On the one hand, instances of `LineBreakpoint` class implement traditional breakpoints defined in line numbers in the frontend. Line breakpoints need only the definition of the `uri`, `line number`, and if the breakpoint is `enabled` or not. On the other hand, instances of the `SectionBreakpoint` class, correspond to the message-oriented breakpoints that we proposed in the breakpoint catalog of Section 5.1.1. As we observed in the breakpoint examples shown in Section 5.1.1 message breakpoints are defined by a source coordinate object that consists of a `line number`, a `column number` and a `char length`. The coordinate also has information about the program file `uri`. A section breakpoint contains also a breakpoint type `bpType`, e.g., message sender, message receiver, etc., and if it is `enabled` or not.

The `Breakpoints` class contains an instance of the Truffle `DebuggerSession` class and two maps of breakpoints, the ones managed by Truffle, i.e., `truffleBreakpoints`, and the ones created using wrapper nodes, i.e., `breakpoints`. The `DebuggerSession` class from Truffle Debug API represents a single debugging session of a debugger.

Moreover, the `BreakpointEnabling` class checks if a breakpoint is active or a stepping target has been reached. This class is used for breakpoints and stepping operations that are not managed by the Truffle framework, i.e., that used wrapper nodes.

As mentioned before, the `BreakpointType` class is needed to specify the type of breakpoint when creating a section breakpoint, i.e., a message-oriented breakpoint. This class is defined by the name of the breakpoint, the applicable node tags, and a stepping type (see Section 5.1.2).

Similarly, in `SteppingType` class, each type of stepping is defined by a name and the possible node tags to which the stepping operation is applicable.

[3]These classes are available at `https://github.com/ctrlpz/SOMns/tree/somns-intellij-4.5/src/tools/debugger/breakpoints` and `https://github.com/ctrlpz/SOMns/tree/somns-intellij-4.5/src/tools/debugger/nodes`

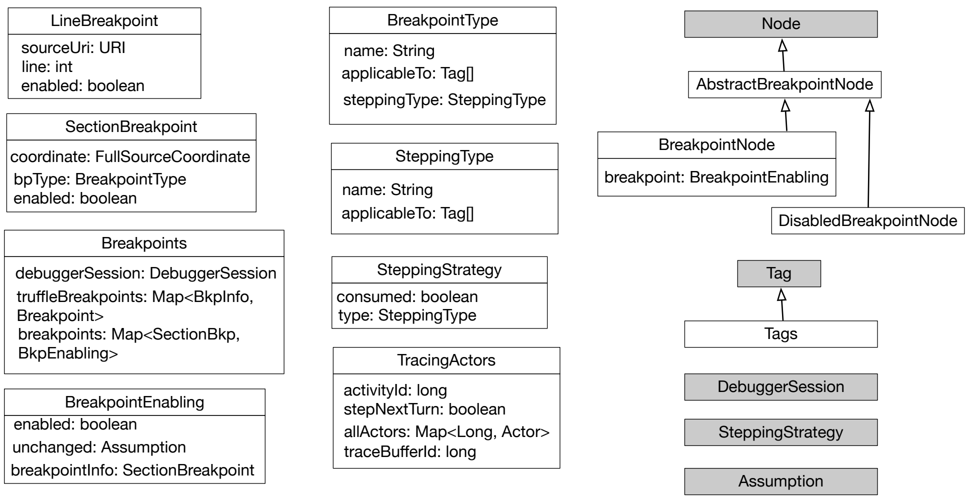


Figure D.2: Main classes related to the breakpoints and stepping operations.

The `SteppingStrategy` class indicates if a stepping command has been consumed or not in the current thread and keeps the stepping type defined for an actor.

The `TracingActors` class which extends from the SOMns `Actor` class, corresponds to an actor in the Kómpos trace (see Section 6.1.3). This class is responsible for checking the flags for debugging commands, e.g., if the message has a flag enable for halting at the receiver side or on promise resolution. `TracingActors` is defined by the following fields:

- `activityId` actor identifier.
- `stepNextTurn` flag to enable a step to next turn command.
- `allActors` map with all the actors created in the program, which is needed when the developer wants to pause actors without declaring breakpoints explicitly in the frontend.
- `traceBufferId` buffer identifier used in the Kómpos trace to record the events of the current actor.

`Assumption` class represents the implementation of a boolean flag that starts always in `true`. Once invalidated, an assumption cannot be valid again. A field instance of this class is declared in `BreakpointEnabling` class, which is needed to know if a breakpoint was enabled or disable.

As mentioned before, wrapper nodes are created using the `AbstractBreakpointNode` class. We have two subclasses, `BreakpointNode` and `DisabledBreakpointNode`. The first one contains the specializations to change the breakpoint state, i.e., enable or disable. The second one is a node that always returns false, which is needed when the Truffle debugger is not enabled.

`Tags` class contains the implementation for all the syntactic tags used in SOMns to annotate nodes for enabling breakpoints, e.g., `ExpressionBreakpoint`. This class extends from the `Tag` class of the Truffle Instrumentation API.

`SteppingStrategy` class from Truffle Debug API provides an implementation of different stepping commands, that allows execution to continue until it reaches another location, e.g., step into, step over, etc. There we implemented two additional stepping strategies, i.e., *step next* and *step end turn*, which are used for the breakpoints that implement wrapper nodes and the step to end turn command (see Section 5.1.2).

D.1.3 Trace Events for Actor State Inspection

We described here how we obtain information to visualize the mailbox for a paused actor showed in Section 5.1.3.1.

An actor of the program is recorded in the Kómpos trace when the actor is created, with an id, name and source location (see Figure 5.14). The activity completion event is recorded by the `KomposTrace` class for other concurrent activities different from actors. To inspect the actor state, we need to record information when a message is about to be sent to that actor (i.e., send operations) and when that message arrives in the actor's mailbox (i.e., message receptions). Furthermore, we also need the information related to messages and promises (i.e., passive entities) and turns (i.e., dynamic scopes).

Passive entities are recorded when they are created in `SPromise` class. Dynamic scopes are recorded when the node corresponding to the body of the invoked method is about to be executed, i.e., in `ReceivedRootNode` class.

From the `SendOperation` event, we obtain the *type* of the message, using the `marker` property, i.e., if it is a message sent to a far reference, a message sent to a promise or if the send operation corresponds to the resolution of a promise. The information about the *target* of the message that we show in Figure 5.2 corresponds to the `targetId`, i.e., the id of the actor or the id of the promise to which the message was sent. As we showed in Figure 5.14 we added other properties to the `SendOperation` event for the mailbox visualization:

- `msgSelector` selector of the message, i.e., the message name.
- `targetSource` source section where the message was sent, i.e., the *origin* property shown in the mailbox of Figure 5.2.
- `targetActivity` this is needed to identify the actor that resolves a promise. For visualizing a message sent to a promise in the actor's mailbox, we need the information of the actor that resolves that promise to show the message as sent to the paused actor. We explain the details of our implementation in Section 6.2.1.1.
- `length` indicates de string length of the resolution value for a promise. This field is recorded for all promise resolution messages (i.e., when the promise is resolved with a value or an error). In the case of chained promises we record only the id of the promise that is used to resolve the chained promise.
- `resolutionValue` corresponds to the value or the error that is used to resolve a promise when a promise is resolved.

Then, every time an eventual message is created in the backend send operation events are recorded in the trace, i.e., in `EventualSendNode.SendNode` class and classes related to promise primitives e.g., `PromisePrims.WhenResolvedPrim` class. We obtain the *turn* information for a message when parsing the events in the frontend (see details in Section 6.2).

For the mailbox visualization shown in Figure 5.2 we need to record also the `MessageReception` event. We use the notion of messages received by the actor to guarantee that the order of messages visualized in the mailbox corresponds to the messages that will be executed by the actor, i.e., messages are shown in order of arrival. In particular, we record message reception at two points in the `Actor` class in the debugger backend, i.e., when the actor appends the received message in its mailbox, and when the actor is about to process the messages of its mailbox. We need both recordings because the actor can be paused and still received messages.

Actors in Medeor execute on a thread of the fork join pool when they have messages in their mailbox, i.e., instances of `ActorProcessingThread` which extends from `TracingActivityThread` class. Each thread executes the events of one actor at a time, and each thread contains one buffer. The recording of trace events in `KomposTrace` happens in the buffer of the current actor thread. Thus, if the actor is not running, e.g., is not processing messages because it is *idle*, this means there is no thread processing the messages for that actor yet, then no message recording is possible. If the actor is paused due to a breakpoint or a stepping, is not processing any messages but new messages can arrive in its mailbox sent by other actors that are running in the program. In this case, the thread in which this paused actor is executed is available and we can record information about these messages:

- `messageId` identifier of the message received.
- `activityId` identifier of the current actor processing the message.

If the actor is not idle, neither paused due to a debugging operation, the actor can receive messages and process them immediately. Then we need to avoid recording the message twice in the trace. We keep a map of messages received by each actor in `KomposTrace` class that we check before saving new message information.

D.1.4 Asynchronous Stack Trace

Figure D.3 shows the main classes related to the asynchronous stack trace support in SOMns.

The `ShadowStackEntryLoad` class represents the node that is declared as a child for every node of the AST where the instrumentation is going to take place, i.e., where the new entry is created to be saved in the shadow stack.

The `StackIterator` class traverses the run time stack and all available calling context information. It manages the iteration for the two available types of stacks, i.e., synchronous stack, and the asynchronous stack. If the stack trace request corresponds to an asynchronous stack, the shadow stack iterator only uses the first frame and then it relies on the shadow stack entry to get the next stack entries.

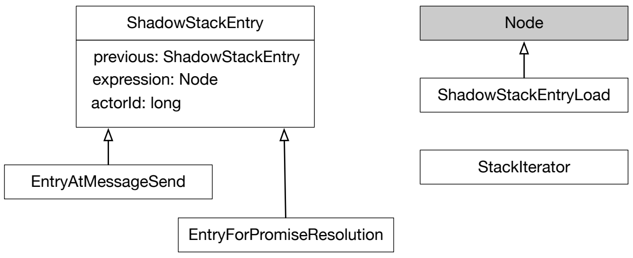


Figure D.3: Main classes related to the asynchronous stack trace support in SOMns. `Node` is the abstract base class for all Truffle nodes.

D.2 Apgar Frontend Implementation Classes

Figure D.4 shows the main classes related to the implementation of the Apgar frontend. Here we describe each of them.

The `SomnsDebugProcess` is the main class of the debugger implementation which defines a debugging session. It provides the debugging capabilities for the SOMns language as an extension to the `XDebugProcess` from IntelliJ Debug API. `SomnsDebugProcess` it is defined by:

- `processHandler` defines the handler to manage the execution process and capture its output.
- `debuggerController` handles the data exchange between the debugger frontend and the backend. Besides, it manages the interaction between the model and the view.
- `connection` creates the connection with the backend through WebSockets clients.
- `tabController` registers the components for the new tabs in the Debug Tab of IntelliJ.

- `lineBreakpoints` map that saves the line breakpoints source location in the editor for sending it to the backend.
- `sectionBreakpoints` map that saves the section breakpoints by their source location in the editor for sending it to the backend.
- `actorDebugList` list of all the actor paused in a debugging session. For each actor, we keep the debugging information of its last suspension, e.g., due to a breakpoint or a stepping operation.
- `scheduleFuture` schedule the task of requesting trace information every 1000ms to the backend.

The `SomnsDebuggerRunner` class extends from the IntelliJ `GenericProgramRunner` class to start the debugging session defined by the implementation in `SomnsDebugProcess` class.

The `SomnsDebuggerSupport` class extends the IntelliJ `DebuggerSupport` class with stepping handlers to support new custom stepping commands, i.e., message-based stepping. The stepping commands are implemented as actions that extend from `XDebuggerSuspendedActionHandler` class of IntelliJ and correspond to buttons in the toolbar of the Debug tab.

The `ClientMessageHandler` and `ClientTraceHandler` classes implement the web sockets clients for the connection with the backend, i.e., for receiving the debugger messages and the trace events, respectively. The `SomnsVMConnection` manages the connection with the debugger backend through web sockets clients.

The `DebuggerController` is the class that handles the data exchanged between the debugger and the interpreter. Additionally, it manages the interaction between the model and the view in the plugin. The `SomnsDebugTabController` class registers the components for the new tabs in the Debugger Tab. There we added new views for showing actors, mailbox, turns, and sentbox. And the class `SomnsDebugViewModel` updates the data received from the Kómpos protocol in the debugger views. We declare five maps that keep updated the data related to actors, send operations, dynamic scopes, passive entities, and received messages in the instance of `SomnsDebugTabController` for visualization.

`TraceParser` class parse the trace events received from the backend through the Kómpos protocol. This class defines the following fields:

- `parseTable` contains the markers for each trace event of the Kómpos protocol.
- `typeCreation` contains the entity types of the Kómpos protocol and its creation marker, i.e., actor, promises, messages, and turns.
- `sendOps` contains the send operations and its markers, i.e., message to a far reference, message to a promise, and promise resolutions.
- `metaModel` the `KomposMetaModel` keeps track of all data send by the backend corresponding to the Kómpos protocol meta model. We check that the markers received for each entity in the buffer match with the markers defined in the meta model.
- `executionData` manages data about the program execution, which is used for processing the trace events for visualization in the UI. Similar as to the Kómpos debugger [MLA$^+$17] we needed to handle data races between the frontend and the backend, e.g., in the `ExecutionData` class, we implement promises for symbol ids, in order to wait for all dependent data elements before a trace event can be used further in the frontend.
- `eventsByActor` keep all events by actorId and bufferId in order to resolve the turn's events after the parsing of the trace. This is needed because events in Kómpos trace can be recorded out-of-order (see Section 6.2.1.2).
- `currentTurn` keeps the identifier of the current executing turn received in the trace. It is reset when the end of the turn is reached.
- `currentBufferId` keeps the identifier of the buffer where the events received were recorded.
- `currentActivityId` keeps the identifier of the current executing actor.

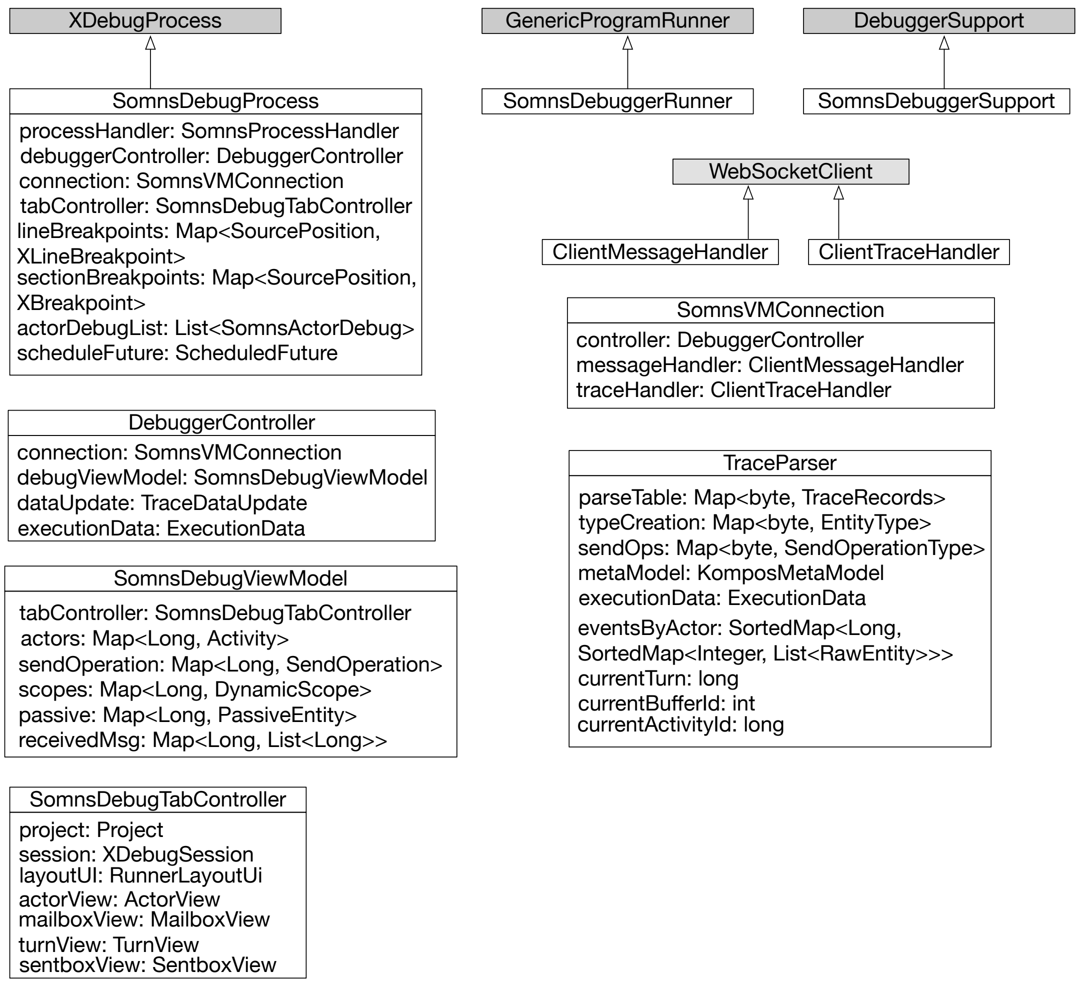

XDebugProcess
SomnsDebugProcess
processHandler: SomnsProcessHandler
debuggerController: DebuggerController
connection: SomnsVMConnection
tabController: SomnsDebugTabController
lineBreakpoints: Map<SourcePosition, XLineBreakpoint>
sectionBreakpoints: Map<SourcePosition, XBreakpoint>
actorDebugList: List<SomnsActorDebug>
scheduleFuture: ScheduledFuture
GenericProgramRunner
SomnsDebuggerRunner
DebuggerSupport
SomnsDebuggerSupport
WebSocketClient
ClientMessageHandler
ClientTraceHandler
SomnsVMConnection
controller: DebuggerController
messageHandler: ClientMessageHandler
traceHandler: ClientTraceHandler
DebuggerController
connection: SomnsVMConnection
debugViewModel: SomnsDebugViewModel
dataUpdate: TraceDataUpdate
executionData: ExecutionData
TraceParser
parseTable: Map<byte, TraceRecords>
typeCreation: Map<byte, EntityType>
sendOps: Map<byte, SendOperationType>
metaModel: KomposMetaModel
executionData: ExecutionData
eventsByActor: SortedMap<Long, SortedMap<Integer, List<RawEntity>>>
currentTurn: long
currentBufferId: int
currentActivityId: long
SomnsDebugViewModel
tabController: SomnsDebugTabController
actors: Map<Long, Activity>
sendOperation: Map<Long, SendOperation>
scopes: Map<Long, DynamicScope>
passive: Map<Long, PassiveEntity>
receivedMsg: Map<Long, List<Long>>
SomnsDebugTabController
project: Project
session: XDebugSession
layoutUI: RunnerLayoutUi
actorView: ActorView
mailboxView: MailboxView
turnView: TurnView
sentboxView: SentboxView

Figure D.4: Class diagram of the main classes of Apgar debugger frontend. In gray color are represented classes from the IntelliJ platform and the WebSocket API.

D.3 Interactions between Apgar Frontend and Backend

Here we explain the interaction of Apgar frontend with Apgar backend for the requests of breakpoint activation, trace information, and asynchronous stack information.

D.3.1 Setup and Breakpoint Activation

First, we explain the setup of a debugging session in Apgar through the sequence diagram of Figure D.5. In an online debugging scenario, we assume the developer has set a breakpoint(s) in the program before starting the debugging session. Once the debugging session starts (1), the web sockets clients are initialized (2), and the method `runSomnsWebDebugger` creates the external process that starts the SOMns interpreter using command line arguments defined in the debug configuration (3). When the SOMns language starts, it enables the Truffle language context (4) and starts the debugger backend connection, i.e., in the `WebDebugger` class (5). The `WebDebugger` creates an instance of the `FrontendConnector` class, which is initialized with the breakpoint defined by the developer in the debugger frontend. In the `FrontendConnector` class, breakpoints are saved (6) according to their type (7) and later installed in the Truffle debugging session (8).

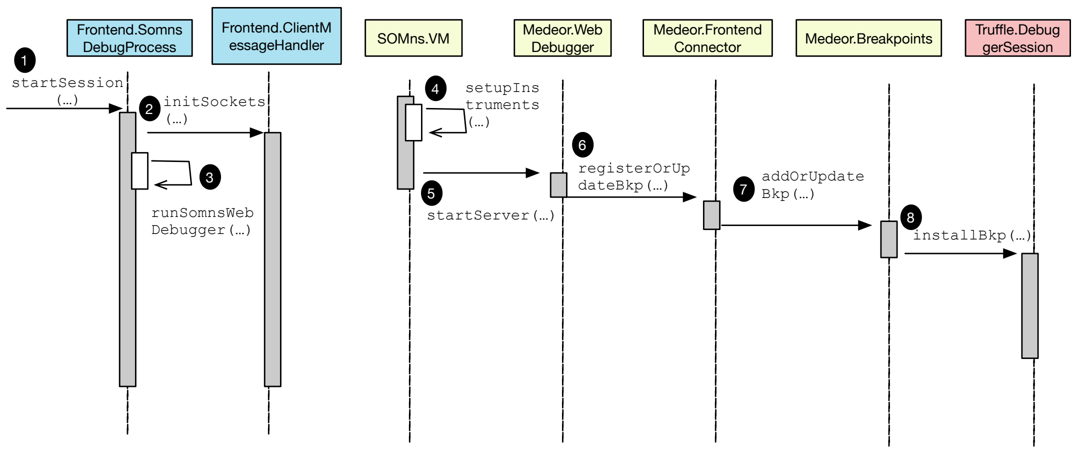


Figure D.5: Sequence diagram corresponding to the setup of an online debugging session in Apgar. Classes in *blue* color denote classes of the debugger frontend, classes in *yellow* color denote classes of the SOMns interpreter and the class in *pink* color belongs to the Truffle Debug API.

Now we explain the class interactions when a breakpoint is reached using the sequence diagram of Figure D.6. The `DebuggerSession`[4] class handles the suspension request of the guest language execution thread, when it receives a notification that the thread has reached an AST location (1). Then, the web debugger, which is implemented as a Truffle `SuspendedCallback` receives the event of the suspension (2) and sends a `StoppedMessage` to the frontend (3). The actual suspension of the current thread is done in the `Suspension` class (4), through a blocking operation that waits for the execution of the next available task, i.e., resuming or sending a stack trace to the frontend. In the frontend, the class `ClientMessageHandler` listens for the messages sent by the interpreter (5). When a `StoppedMessage` is received (6), the controller class sends a notification to the debug process, which does the corresponding data update in the debugger UI (7).

The described interaction of Figure D.6 can occur due to a breakpoint, a stepping operation, or pausing explicitly an actor in the frontend. These actions trigger the sending of a `Stopped` message to the frontend indicating a suspension in the program.

[4]The Truffle Debug API is available at `https://www.graalvm.org/truffle/javadoc/index.html?com/oracle/truffle/api/debug/`

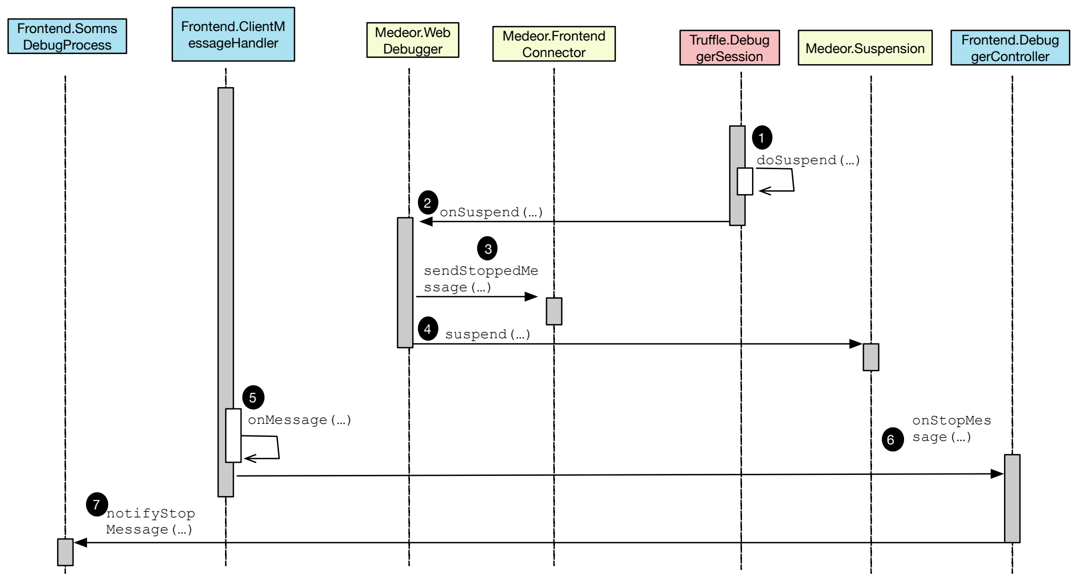


Figure D.6: Sequence diagram of a breakpoint activation.

D.3.2 Trace Information Request

We now detail the debugger interactions in response to a trace information request. Figure D.7 shows a sequence diagram of the classes we implemented to get the trace information in the frontend [5].

[5] This implementation is based on one of the Kómpos debugger, which has an initial version of a process order visualization for actors using the trace.

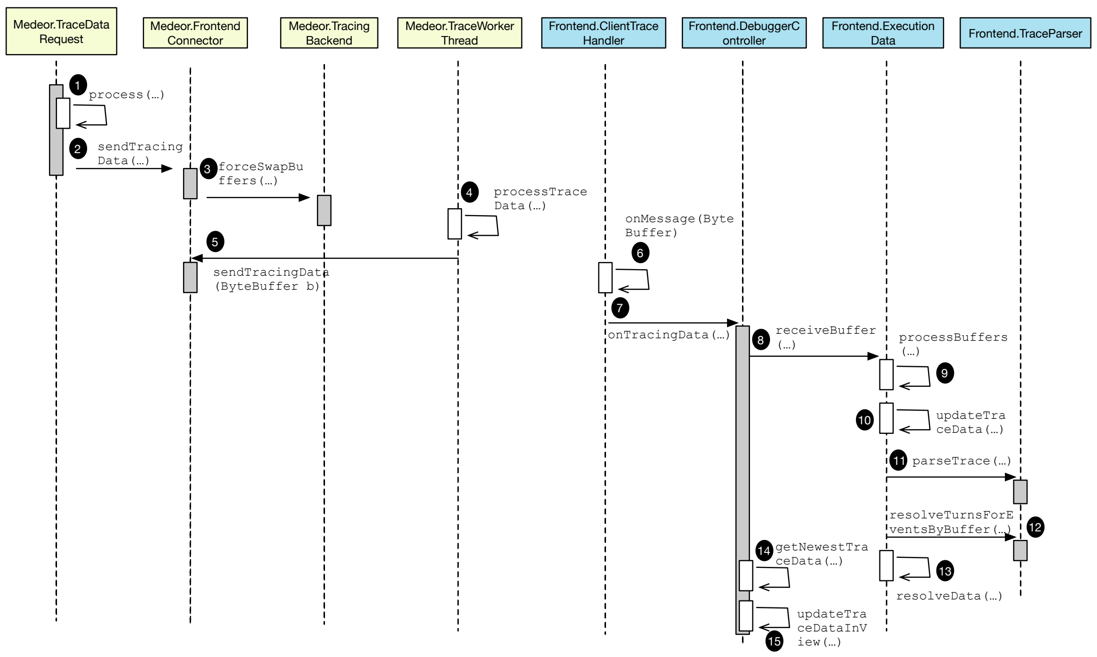


Figure D.7: Sequence diagram for trace data request. Classes in *blue* color denote classes of the debugger frontend, classes in *yellow* color denote classes of the SOMns interpreter. The goal of this interaction is to return the Kómpos protocol trace events recorded for the program being debugged.

First, the debugger frontend requests the trace information to the SOMns interpreter. This request has been implemented with a *pull strategy* for requesting the trace data periodically (i.e., every 1000ms) to the backend. The request task is canceled once the debugging session stops.

Once Medeor receives the `TraceDataRequest` message (1), the `FrontendConnector` requests swapping the buffers (2). *Swapping* the buffers consist of creating new buffers for the current executing threads. One buffer represents a *subtrace* of the events of an actor when it starts executing on a thread [AMB$^+$18][6]. The `TracingBackend` manages the swapping of the buffers for all executing threads, which are instances of `TracingActivityThread` class (3). Instances of the `TraceWorkerThread` class, correspond to the threads that write events in the trace. These threads capture the available data periodically from the available buffers (4), e.g., after the swapping, and send the buffers' data to the debugger (5).

After the debugger frontend receives a trace message (6), the `DebuggerController` passes this information (7) to the `ExecutionData` class (8), which manages the processing of the buffers (9, 10). The `TraceParser` class parses the new buffer data received (11), and afterward, it resolves each parsed event to their corresponding turn (12).

As mentioned in [MLA$^+$17] concurrent debugging interactions between the backend and frontend need to handle data races. Remember from section 6.1 that the web debugger implementation in Medeor has two web socket instances. Hence, the order in which messages are exchanged in these two connections is not guaranteed. To handle these races, we wait for all dependent trace events.The method in step (13) of the diagram resolves this dependent data, using promises to wait, for example, for symbols information, such as file uri ids. After the processing of the buffer information we get the lists of all

[6]Buffers are created once, and recycled after being emptied.

the new events parsed (14), i.e., all the *activities*, *dynamic scopes*, *passive entities*, *send operations* and *received messages*. With this new information, we update the trace-based debugger views in the frontend (15). These views are described in sections 5.2.2.1 and 5.2.2.2.

D.3.3 Stack Trace Information Request

So far, we have shown the debugger frontend interactions and backend using the two web sockets, for example, to send debugger messages (e.g., `Stop` message) and to send trace events. The interaction we show in Figure D.8 responds to another debugger message of the Kómpos protocol, i.e., `StackTraceResponse`, which uses the same web socket connection as for the `Stop` message.

When the `DebuggerController` class in the frontend receives the debugger message `Stopped`, it requests the stack trace information to the backend. The SOMns interpreter receives the `StackTraceRequest` message (1), and then it requests to the `FrontendConnector` the suspension object corresponding to the paused actor (2). With the suspension information, the task `SendStackTrace` is submitted (3, 4) to be processed (5). When the task `SendStackTrace` is processed (6, 7), the `FrontendConnector` requests the creation of a new instance of the message `StackTraceResponse` (8). For creating this response, the stack frames are requested to the `ApplicationThreadStack` (9, 10). To build the stack iterator the `ApplicationThreadStack` asks Truffle `SuspendedEvent` for the list of guest language stack frame objects (11). With this list of frame objects a new suspension iterator is created for the suspended event and the paused actor (12). Here, the interpreter checks if the flag corresponding to the asynchronous stack is enabled. In this case, the suspension iterator uses the stack frames of the *shadow stack*. Otherwise, the `ApplicationThreadStack` returns a list of frames built by Truffle, i.e., a stack without asynchronous information. By default, the asynchronous stack trace is enabled in the debugging configuration in the frontend. For changing the stack type, the developer can specify the following commands: `-sst` for synchronous stack and `-asts -astic` for an asynchronous stack.

Thus, a new response object is created with the stack frames information (13) and send to the frontend (14). At the frontend, the `DebuggerController` class receives the `StackTraceResponse` message and requests the scopes and variables information for each frame (15, 16).

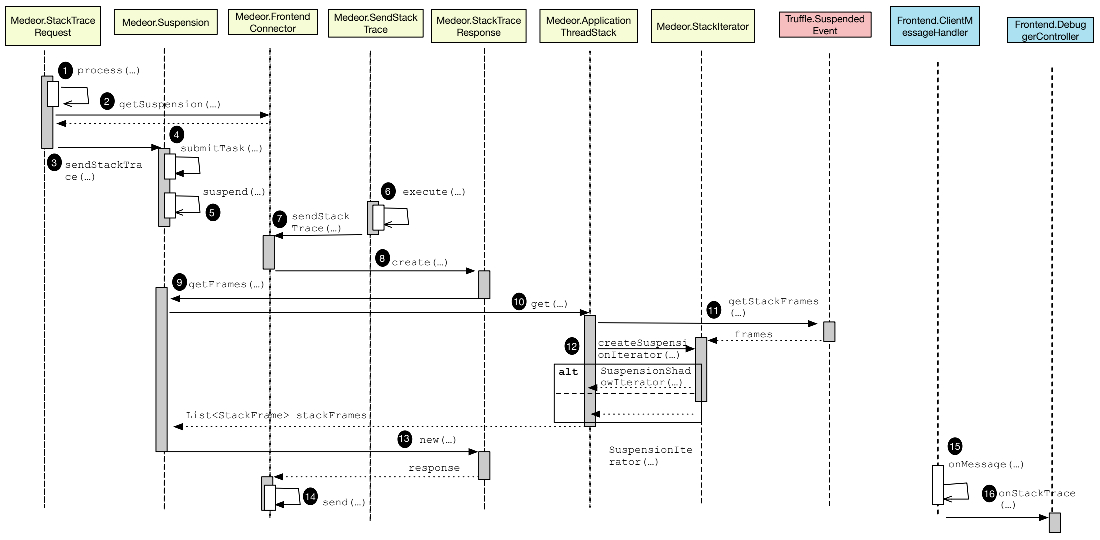


Figure D.8: Sequence diagram for stack trace information request. Classes in *blue* color denote classes of the debugger frontend, classes in *yellow* color denote classes of the SOMns interpreter and the class in *pink* color belongs to the Truffle Debug API.

Appendix E

User Study Material

E.1 Code of conduct for the online experiment

To not threaten the validity of the results participants of the experiment are required to understand and respect the following rules:

- Do not ask questions concerning the assignments out loud during the experiment. If you have a question concerning the assignments, write them to the host (Carmen) in private mode (chat message via Zoom). If you want to talk with the host you can ask her to assign you to a breakout room.
- During the experiment you are not allowed to communicate via chat messages, phone and mail with another participant.
- Use only the provided material. As soon as you have an answer for an assignment, please send it to the host (privately). We plan the experiment to be approximately 90 minutes, but we will measure the time you spend solving the assignments. For each assignment you have a time limit of 30 - 45 minutes approximately.
- Do not leave the Zoom room until you have finish the experiment. We will apply the experiment for groups at different times, then, after you finish the experiment, please do not comment on information about it to other participants.

Do you agree with the rules I have read?

E.2 Steps for random assignment

Here we summarize the steps of a procedure mentioned in [CJT15], to randomly assign (e.g., 28) participants to two groups, i.e., one group to receive the experimental treatment condition and one group to receive the control condition. In the following steps is not described the individual matching technique.

- Step 1. Create a table of random numbers (e.g., 28 rows x 14 columns). Generate a random number for each cell in the range of 0-9.
- Step 2. Number the participants from 0 to 27 (participants' IDs).
- Step 3. Block the list of IDs into columns of two, because the maximum number of participants you have is a two-digit number.
- Step 4. Randomly select the first group of 10 participants by reading down the first two columns (in the table of step 1) until you come to a number less than 28. The read number from the table

will represent the ID of the participant selected for the first group. Continue reading until we have the required amount of participants for the first group.

- Step 5. Do the same procedure as step 4 for the second group (from where we stop the reading for group 1). The second group will be the remaining participants in the table.
- Step 6. After we have obtained the same number of groups as there are treatment conditions, the groups should ideally be randomly assigned to the treatment conditions. In this case, this is accomplished by using only one column of the table of random numbers because there are only two groups of participants. The two groups are numbered 0 and 1. e.g., if the first number encountered that is less than 2 is 0, so group 0 (the first group of participants) is assigned to the first treatment group. Then, group 2 is assigned to the second treatment group.

E.3 Debugging assignments

The purpose of these debugging assignments is to get familiar with the SOMns debugger in IntelliJ IDE and to find concurrency bugs often present in actor-based programs. This plugin provides debugging support for SOMns programs that feature *actors* as concurrent entities. To this end, each program of the debugging assignments contains one error. Using the debugger features, you should try to find the cause of the error.

E.3.1 Assignment 1: FlightBooking

Description: The flight booking program handles new booking of flights. The SOMns implementation corresponds to a minimized version of a web application, in which customers can book flights through a website.

Figure E.1 shows the actors created in the program: Platform, Customer, Website and Database.

The Platform actor starts the program and manages the runtime system of SOMns.

In response to the `bookFlight` message, the Customer actor requests to the Website actor the flight prices (`requestFlightPrices` message). The program generates randomly the flight id and the seat id that the Customer will select for the booking, and with this information, the Customer actor sends the `createBooking` message to the Website actor. Before adding a new booking in the bookings array, the Website actor sends the message `checkSeat` to the Database actor to check if the Customer's seat id is available in the specified flight. The Website actor can change flight prices.

Once the booking is created, the Customer sends the message `pay`, corresponding to the payment of the flight. When the payment is executed, the Customer receives a confirmation of the booking created from the Website actor, through the `confirmBooking` message. Finally, the Customer sends a `done` message to indicate to the Website actor that the Customer finished his/her booking.

The solution is parametrized by the number of customers (3), the number of flights (2) and the number of available seats by flight(2).

Assignment: Running `FlightBooking.ns` should show the output of figure E.2 a), in which all customers have their booking confirmed. However, figure E.2 b) shows an error in the output.
Note: You should comment the timeout in the main method of the program to be able to debug without timeout restrictions (from line 118 - 121). A comment in SOMns is declared like this `(* commented code *)`.

Your task is: Use SOMns debugger to find the cause of the problem. When you have the answer, send a private chat message via Zoom with the line number of the fault to Carmen.

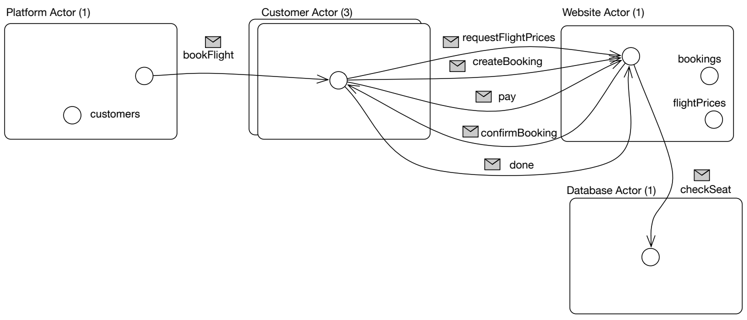
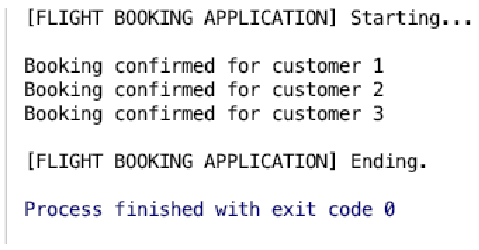


Figure E.1: Conceptual diagram for the flight booking program.



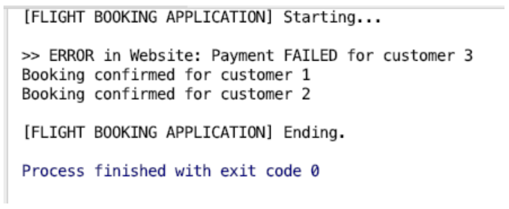
```
[FLIGHT BOOKING APPLICATION] Starting...

Booking confirmed for customer 1
Booking confirmed for customer 2
Booking confirmed for customer 3

[FLIGHT BOOKING APPLICATION] Ending.

Process finished with exit code 0
```

(a): Expected output



```
[FLIGHT BOOKING APPLICATION] Starting...

>> ERROR in Website: Payment FAILED for customer 3
Booking confirmed for customer 1
Booking confirmed for customer 2

[FLIGHT BOOKING APPLICATION] Ending.

Process finished with exit code 0
```

(b): Observable output

Figure E.2: Flight booking program output.

```
class FlightBooking usingPlatform: platform = Value (
| private actors = platform actors.
  private Array = platform kernel Array.
  private TransferArray = platform kernel TransferArray.
  private harness = (platform system loadModule: 'core-lib/Benchmarks/Harness.ns'
     nextTo: self) usingPlatform: platform.
  private Random = harness Random.
  private Pair = platform kernel Pair.

  private numCustomers = 3.
  private numFlights = 2.
  private numSeats = 2. (* number of seats that can be reserved by plane *)
|)(

  public class Database = ()(

    public checkSeat: flightId seat: seatId = (
      ^ seatId = 1 or:[ seatId = 2].
    )
  )

  public class Website new: completionRes db: database = (
  | private completionRes = completionRes.
    private database = database.
    private finishedCustomers ::= 0.
    private flightPrices = TransferArray new: numFlights withAll: 50.
    private bookings = TransferArray new: numCustomers.
    private resolved ::= false.
  |)(

    public requestFlightPrices = (
        ^ flightPrices.
    )

    public createBooking: customerId flight: flightId seat: seatId = (
      | price pSeat completionPP |
      completionPP:: actors createPromisePair.

      pSeat:: database <-: checkSeat: flightId seat: seatId.
      pSeat whenResolved:[: seatAvailable|
        seatAvailable ifTrue:[
          price:: (flightPrices at: flightId).
          bookings at: customerId put: (Pair withKey: flightId andValue: price).
          flightPrices at: flightId put: (price + 10).(* increase flight price *)
          completionPP resolver resolve: price.
        ]
      ].

      ^ completionPP promise
    )

    public pay: customer customerId: customerId flight: flightId
    amount: amount = (
      | customerBooking |
      customerBooking:: bookings at: customerId.
      (flightId = (customerBooking key) and:[amount = (customerBooking value)])
      ifTrue:[
        customer <-: confirmBooking: flightId amount: amount.
      ]
      ifFalse:[
        ('ERROR in Website: Payment FAILED for customer '+ customerId) println.
        resolved ifFalse:[
          completionRes resolve: false.
          resolved:: true ]
      ].
    )

    public done = (
      finishedCustomers:: finishedCustomers + 1.
      finishedCustomers = numCustomers ifTrue:[
        resolved ifFalse:[
          completionRes resolve: true.
          resolved:: true ]
      ]
    )
  )

  public class Customer new: customerId website: web = (
  | private customerId = customerId.
    private web = web.
    private rand = Random new: customerId + 73425.
  |)(
    public bookFlight = (
      | flightId seatId bookingPromise |
        (web <-: requestFlightPrices) whenResolved:[:flights |
          flightId:: 1 + (rand next % numFlights).
          seatId:: 1 + (rand next % numSeats).

          bookingPromise:: web <-: createBooking: customerId
                                   flight: flightId
                                   seat: seatId.
          bookingPromise whenResolved:[: price |
          | paymentPromise |
            paymentPromise:: web <-: pay: self
                                     customerId: customerId
                                     flight: flightId
                                     amount: (flights at: flightId).
          ].
        ].
    )

    public confirmBooking: flightId amount: amount = (
       ('Booking confirmed for customer ' + customerId) println.
       web <-: done
    )
  )

  public main: args = (
    | customers website database payment completionPP timeout |
    timeout:: 3000.

    '[FLIGHT BOOKING APPLICATION] Starting...\n' println.

    completionPP:: actors createPromisePair.
    database:: (actors createActorFromValue: Database) <-: new.
    website:: (actors createActorFromValue: Website) <-: new: completionPP
    resolver db: database.

    customers:: Array new: numCustomers.
    customers doIndexes: [:i |
      | c |
      c:: (actors createActorFromValue: Customer) <-: new: i website: website.
      customers at: i put: c ].

      customers do: [:c | c <-: bookFlight ].

      actors after: timeout do: [
        'Program exit due to TIMEOUT' println.
        completionPP resolve: 1.
      ].

     ^ completionPP promise whenResolved:[: r|
         '\n[FLIGHT BOOKING APPLICATION] Ending.' println.
    ].
  )
)
```

Listing E.1: Flight booking program in SOMns.

E.3.2 Assignment 2: OrderPurchase

Description: The order purchase program handles new purchase of orders. Figure E.3 shows the actors created in the program: Platform, Customer, Website, Store, Account, Shipper and Database.

The Platform actor starts the program and manages the runtime system of SOMns. First, the Platform actor sends a `buy` message to the Customer actor, with all the product items he/she wants to buy. Then, in response to the `buy` message, the Customer actor sends the message `checkoutShoppingCart` to the Website actor.
Before the order is acknowledged, the Website actor must verify three services which are represented by three actors in the program:

- whether the requested items are still in stock (service managed by the Store actor)
- whether the customer has provided valid payment information (service managed by the Account actor)
- whether a shipper is available to ship the order in time (service managed by the Shipper actor)

Consequently, the Website actor sends the messages `productInStock`, `checkCredit` and `canDeliver` to the actors Store, Account and Shipper respectively. Each service sends messages to the Database actor to retrieve and manage the needed information and send a reply to the Website actor. To collect the answer of the three services, a promise group is used.

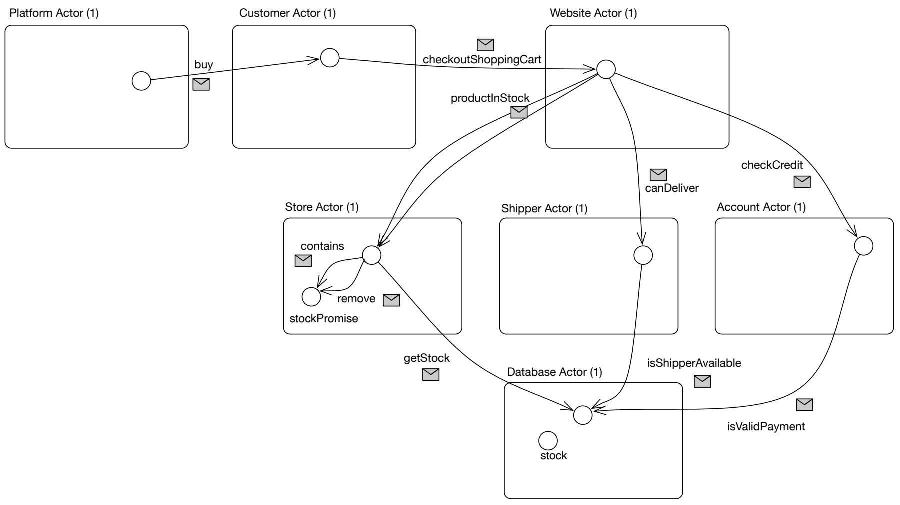
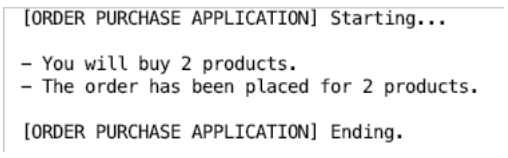


Figure E.3: Conceptual diagram for the order purchase program.

Assignment: Running `OrderPurchase.ns` should show the output of figure E.4 a), in which all requirements are satisfied and the order is placed successfully. However, figure E.4 b) shows a timeout in the output.
Note: You should comment the timeout in the main method of the program to be able to debug without timeout restrictions (from line 159 - 162). A comment in SOMns is declared like this `(* commented code *)`.

Your task is: Use SOMns debugger to find the cause of the problem. When you have the answer, send a private chat message via Zoom with the line number of the fault to Carmen.

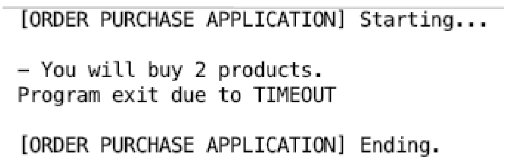


```
[ORDER PURCHASE APPLICATION] Starting...

- You will buy 2 products.
- The order has been placed for 2 products.

[ORDER PURCHASE APPLICATION] Ending.
```

(a): Expected output



```
[ORDER PURCHASE APPLICATION] Starting...

- You will buy 2 products.
Program exit due to TIMEOUT

[ORDER PURCHASE APPLICATION] Ending.
```

(b): Observable output

Figure E.4: Order purchase program output.

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

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

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

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

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

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

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

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

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

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

      ^ completionPP promise
    )
  )

  public class Store = ()(

    public productInStock: item database: database = (
      | stockPromise existPP flag ::= false. |

      existPP:: actors createPromisePair.

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

      ^ existPP promise
    )
  )

  public class Account = ()(

    public checkCredit: customerId database: database = (
      ^ database <-: isValidPayment: customerId
    )
```

```
)

public class Shipper = ()(

  public canDeliver: customerId database: database = (
    ^ database <-: isShipperAvailable: customerId
  )
)

public class Database = (
  | private stock = init. |
)(

  private init = (
    | s |
      s:: Vector new.
      s append: 'hdd'.
      s append: 'ipad'.
      s append: 'phone'.
      s append: 'screen'.
      s append: 'laptop'.
    ^ s
  )

  public getStock = (
    ^ stock
  )

  public isValidPayment: customerId = (
    ^ true
  )

  public isShipperAvailable: customerId = (
    ^ true
  )

)

 public main: args = (
   | customer store account shipper website database items buyPromise
   timeout completionPP |
   timeout:: 3000.

   completionPP:: actors createPromisePair.

   '[ORDER PURCHASE APPLICATION] Starting...\n' println.
   items:: TransferArray new: 2.
   items at: 1 put: 'phone'.
   items at: 2 put: 'laptop'.

   store:: (actors createActorFromValue: Store) <-: new.
   account:: (actors createActorFromValue: Account)  <-: new.
   shipper:: (actors createActorFromValue: Shipper)  <-: new.
   database:: (actors createActorFromValue: Database) <-: new.
   website:: (actors createActorFromValue: Website) <-: new: store
   account: account shipper: shipper db: database.
   customer:: (actors createActorFromValue: Customer) <-: new: 'Joe'
   website: website.

```

```
    buyPromise:: customer <-: buy: items.

    actors after: timeout do: [
     'Program exit due to TIMEOUT' println.
      completionPP resolve: 1.
    ].

    completionPP resolve: buyPromise.

    ^ completionPP promise whenResolved: [:result |
          '\n[ORDER PURCHASE APPLICATION] Ending.' println.
    ]
  )
)
```

Listing E.2: Order Purchase program in SOMns.

E.4 Questionnaire

Dear participant,
As part of our experiment we would like you to fill this questionnaire. Questions are divided into two categories, the first one consists of questions about you and your work as software developer. Second category shows questions regarding your experience with the debugging assignments and the debugger. Thank you for participate in this study!

1. About you

1. **How old are you?** ________
2. **Where do you work/study?** ____________________
3. **What is your scholar degree?** ○ Bachelor. ○ Master. ○ PhD.
4. **How many years have you been developing software?**

   ○ less than 2. ○ 3 to 5. ○ 6 to 10. ○ more than 10.
5. **Which programming languages have you more experience with? (write at most 3 languages)**

   ________________ ________________ ________________
6. **If you have experience in concurrent programming with actors, mention the actor language(s) in which you have more experience.**

   ________________ ________________ ________________
7. **Which one(s) of these debugging techniques have you used when developing actor-based programs?**

   ☐ print statements ☐ log diff ☐ assertions ☐ a debugger
8. **Which debugger(s) have you experience with?**

   ____

2. About the experiment

9. **In which group did you perform the experiment?**
   ○ Control group ○ Experimental group

10. **To what extend do you agree with the following statements? Rate in this scale 1 (Strongly disagree) 2 (Disagree) 3 (Neutral) 4 (Agree) 5 (Strongly agree). If you belong to the control group please answer neutral in statements c-d-e-f.**

(a) The debugging assignments were difficult.	1 2 3 4 5
(b) The debugging assignments are representative of common bugs I have seen in actor-based programs.	1 2 3 4 5
(c) Message breakpoints and stepping operations help to reduce the effort when searching the root cause of concurrency bugs.	1 2 3 4 5
(d) The combination of sequential and message stepping is effective to inspect actor's turn.	1 2 3 4 5
(e) Visualization of message causality is useful for understanding the program while debugging.	1 2 3 4 5
(f) The asynchronous stack trace is useful for identifying message ordering problems.	1 2 3 4 5
(g) The plugin debugging views are useful to inspect actor's state.	1 2 3 4 5
(h) The debugging techniques used in the experiment assist developers not only to discover program faults but to comprehend program behavior.	1 2 3 4 5

11. **Select the assignments you solved:** ☐ **Assignment 1.** ☐ **Assignment 2.**

12. **Did you feel time pressure during the exercises?.**
    ○ Yes. ○ No.

13. **How long did you spend for solving assignment 1? (ask the host for the exact value).**

    ____

14. **How long did you spend for solving assignment 2? (ask the host for the exact value).**

    ____

15. **What other features would you like to see in the Debugger tabs?.**

    ____
    ____
    ____

16. **Please write here any additional comment you have about the experiment.**

    ____
    ____
    ____

17. **Please do the following steps to convert and upload the log file:**
    1. Open the QTerminal that you can find in the desktop.

2. Execute command: `cd IdeaProjects/somnsProject/`
3. Execute command: `enscript -p log.ps log.txt`
4. Execute command: `ps2pdf log.ps log.pdf`
Finally, rename log.pdf file with your name (e.g., log-CarmenTorres.pdf) and upload it here.

E.5 Additional user study results

The following are additional charts corresponding to the experiment results.

E.5.1 Participants profile

Figure E.5 shows that most of the participants of the user study worked at the VUB, for control and experimental group.

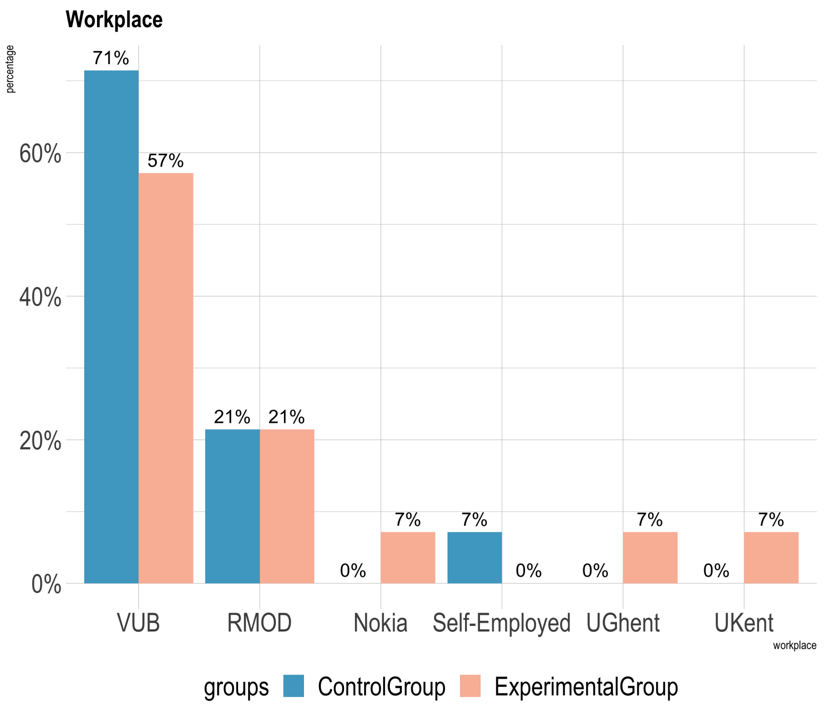


Figure E.5: Participants workplace.

Figure E.6 shows the scholar degree of participants. Most of the part of participants in both groups are master. In the experimental group we can observe more participants with the PhD degree.

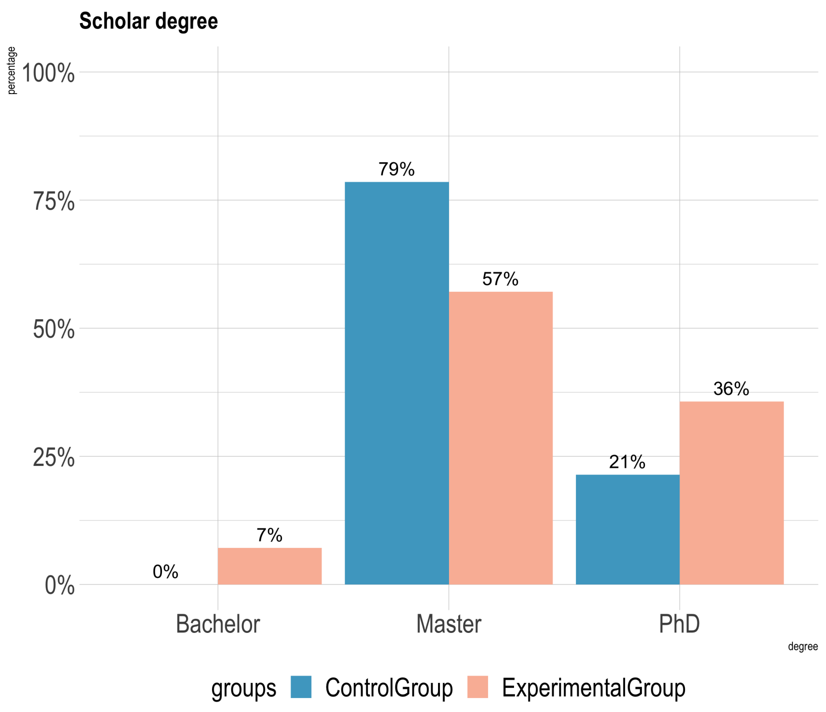


Figure E.6: Participants scholar degree.

Figure E.7 shows that both control and experimental groups are approximately balanced in the number of participants, in particular from 6 to 10 years of experience developing software.

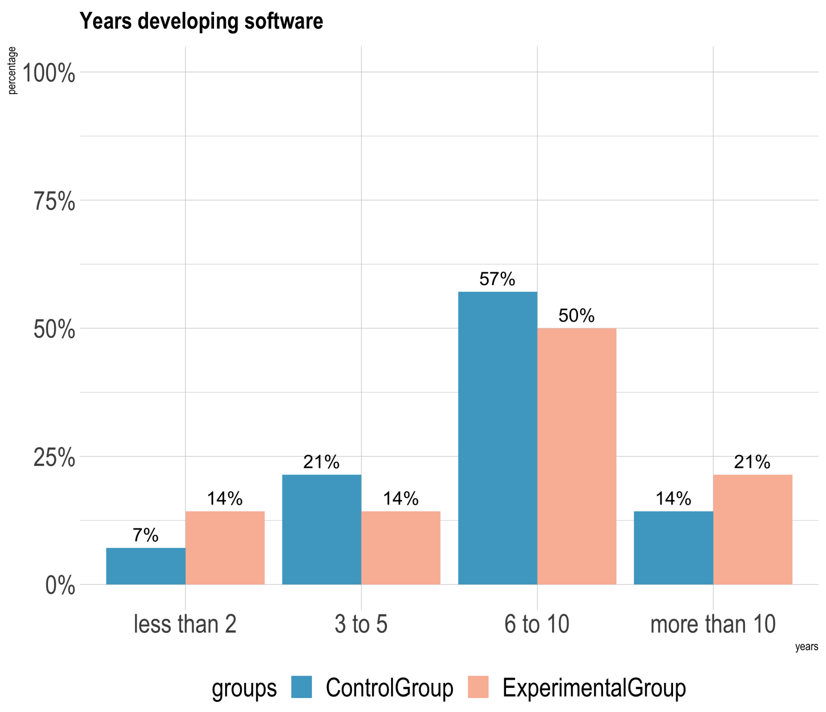


Figure E.7: Years of experience developing software.

Figure E.8 shows that most of the participants in both groups have experience with JetBrains IDEs debuggers.

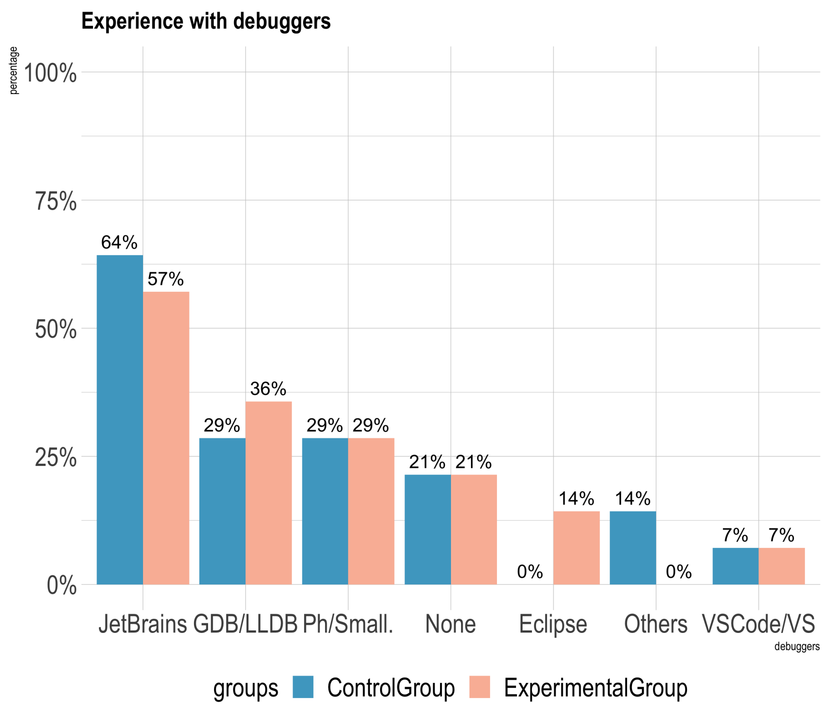


Figure E.8: Debuggers.

E.6 Threats to Validity in the Qualitative Study

Together with the validity threats summarized by Creswell et al. [CC17] for mixed methods experimental studies, we followed also Christensen et al. [CJT15] advise of checking that all components of the research design are conducted appropriately. This idea is considered by the *multiple validities* threat in mixed methods research, i.e., analyze the relevant validity types for both studies. Table E.1 shows threats to validity that we analyze for our qualitative study when collecting data in the posttest questionnaire. The column *Strategy* refers to the strategy we selected to minimize the threat in our study. Regarding threats 1 and 3, we want to add that one researcher conducted the experiment, but two more researchers helped interpret and discuss the data. For threat 2 we added an explicit question in the questionnaire (question 16). For threat 4, we analyzed the results using methods and measures of descriptive statistics and percentage representations for the different related variables.

Validity threat	Description	Strategy	Description
1. Descriptive	Provide an accurate description of a particular phenomenon, situation, or group	Investigator triangulation strategy	Use of multiple investigators to interpret the data
2. Interpretative	Report how people subjectively think and feel about phenomena	Participant feedback, low-inference descriptors	Ask participants their findings and include quite a few quotes from the participants
3. Theoretical	Degree that the theoretical explanation provided by the researcher accurately fits the data	Peer review	Discuss your interpretations, conclusions, and explanations with your colleagues who can provide a different perspective
4. Internal	Describe how each group operates during the experiment and understand how variables are causally related	Researcher as detective	Examine each possible “clue" to draw conclusions about cause and effect. Use descriptive statistics measures.

Table E.1: Threats in qualitative research, from [CJT15].

Appendix F

Debugger Configuration in PLT-Redex

The code in Listing F.1 shows the Double program with a bad message interleaving written in PLT-Redex language. We use the `traces` function of PLT-Redex to explore the sequences of terms of the program in Racket GUI [1]. The program in the listing is represented by K in a debugger configuration in the form $\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K\rangle$, which is described in Section 8.3. In this case, we start the debugging session putting a message receiver breakpoint for the asynchronous message from `client1` actor to the `math` actor, i.e., denoted by `c1-double-to-math`. Here we detail each element of the debugger configuration:

- Line 3 shows the list of pending breakpoints (B_p) with a message receiver breakpoint.
- Line 4 shows the list of checked breakpoints (B_c), which is empty.
- Line 5 shows the state of the debugger (d_s), which is `run`.
- Line 6 shows the list of commands (C), which contains two operations, step to next turn and resume execution.
- Line 7 shows a map of the actors of the program with its state (A_s), in this case `client1` has `run` state.
- Line 8 - Line 26 shows the actor configuration (K), which contains the Double program written in the AmbientTalk operational semantics.

As mentioned in Section 8.3.1, an actor configuration in the AmbientTalk semantics has the form $\mathcal{A}\langle \iota_a, O, Q_{in}, e\rangle$. In this case, the actor with id `client1` has defined the expression to be evaluated e from Line 12 to Line 25, which corresponds to the sample program to compute double numbers (cf. Section 8.2.1).

[1] `https://docs.racket-lang.org/redex/reference.html`

```
(traces dstep
(term
(((Msg-Receiver c1-double-to-math))
  ()
  run
  ((Step-Next-Turn id_new) (Resume-Execution))
  ((client1 run))
  ((actor
    client1
    ()
    ()
    (let (math (actor (field result 0) (method double x (set! (this $ result) (+ x x))) (method result p (this $ result))))
      in
      (let (client2 (actor (method start math (send math double (12) c2-double-to-math))))
        in
        (let (a (send client2 start (math) c1-start-to-c2))
          in
          (let (b  (send math double (33) c1-double-to-math))
            in
            (let (x_f x_r)
              future
              in
              (let (x_l (let (some-var 5) in (object (method apply x ((x_r $ resolve-mu) x)))))
                in
                (let (var (send (let (x_f1 x_r1) future in (let (var (send math result (0 x_r1) c1-result-to-math)) in x_f1)) register-mu (x_l) c1-result-to-math)) in x_f)))))))
     ))
   )
))
```

Listing F.1: PLT-Redex version of the Double program containing a bad message interleaving bug.

Bibliography

[AAGZ15] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Test case generation of actor systems. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *ATVA*, volume 9364 of *Lecture Notes in Computer Science*, pages 259–275. Springer, 2015.

[AAGZ18] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Systematic testing of actor systems. *Softw. Test., Verif. Reliab.*, 28(3), 2018.

[AASE$^+$17] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Wasif Afzal. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Software Quality Journal*, 25(1):49–82, Mar 2017.

[AB20] Ericsson AB. Erlang debugger user's guide. `http://erlang.org/doc/apps/debugger/debugger_chapter.html`, 2020. Online; accessed 5 March 2021.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Softw. Test., Verif. Reliab.*, 13(4):207–227, 2003.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[AMB$^+$18] Dominik Aumayr, Stefan Marr, Clément Béra, Elisa Gonzalez Boix, and Hanspeter Mössenböck. Efficient and deterministic record & replay for actor languages. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery.

[AMGBM19] Dominik Aumayr, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. Asynchronous snapshots of actor systems for latency-sensitive applications. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, pages 157–171, New York, NY, USA, 2019. Association for Computing Machinery.

[AS17] Stavros Aronis and Konstantinos Sagonas. The shared-memory interferences of erlang/otp built-ins. In Natalia Chechina and Scott Lystig Fritchie, editors, *Erlang Workshop*, pages 43–54. ACM, 2017.

[Av16] Akka-viz. A visual debugger for akka actor systems. `https://github.com/lustefaniak/akka-viz`, 2016. Online; accessed 12 April 2021.

[AVWW93] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[AZMT18] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan 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, pages 193–207, Berlin, Heidelberg, 1999. Springer-Verlag.

[BCD$^+$18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.

[BFSK20] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: A comprehensive study on symptoms, root causes, api usages, and differences. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[BFSS10] Maria Brito, Katia R Felizardo, Paulo Souza, and Simone Souza. Concurrent software testing: A systematic review. *On testing software and systems: Short papers*, page 79, 2010.

[BJC$^+$13] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software: Quantify the time and cost saved using reversible debuggers, 2013.

[Blo12] Chromium Blog. Debugging web workers with chrome developer tools. `http://blog.chromium.org/2012/04/debugging-web-workers-with-chrome.html`, April 2012. Online; accessed 5 March 2021.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin 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, pages 211–230, New York, NY, USA, 2002. ACM.

[BM14] Earl T. Barr and Mark Marron. Tardis: Affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 67–82, New York, NY, USA, 2014. Association for Computing Machinery.

[BMM$^+$16] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav 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, pages 1003–1007. ACM, 2016.

[BMP18] F. A. Bianchi, A. Margara, and M. Pezzé. A survey of recent trends in testing concurrent software systems. *IEEE Transactions on Software Engineering*, 44(8):747–783, 2018.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. Association for Computing Machinery.

[BPP14] Thibaut Balabonski, Francois Pottier, and Jonathan Protzenko. Type soundness and race freedom for mezzo. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 253–269, Cham, 2014. Springer International Publishing.

[Bra09] Gilad Bracha. Newspeak programming language draft specification version 0.06. Technical report, Technical report, Ministry of Truth, 2009.

[Bri20] T. Editors of Encyclopaedia. Britannica. Uncertainty principle. `https://www.britannica.com/science/uncertainty-principle`, May 2020. Online; accessed 3 March 2021.

[BS95] Karen L. Bernstein and Eugene W. Stark. Operational semantics of a focusing debugger. *Electronic Notes in Theoretical Computer Science*, 1:13 – 31, 1995. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

[BWBE16] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems. *Commun. ACM*, 59(8):32–37, July 2016.

[CBKB19] Tony Clark, Balbir Barn, Vinay Kulkarni, and Souvik Barat. Making sense of actor behaviour: An algebraic filmstrip pattern and its implementation. In *Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC'19, New York, NY, USA, 2019. Association for Computing Machinery.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[CC17] John W Creswell and Vicki L Plano Clark. *Designing and conducting mixed methods research.* Sage publications, 2017.

[CC19] A. Colak and M. A. Cuvic. An educational tool for visualising actor programs. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 605–610, 2019.

[CDG$^+$19] X. Chang, W. Dou, Y. Gao, J. Wang, J. Wei, and T. Huang. Detecting atomicity violations for event-driven node.js applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 631–642, 2019.

[CFL95] Patrick Coscas, Gilles Fouquier, and Agnes Lanusse. Modelling Actor Programs using Predicate/Transition Nets. In *Proceedings Euromicro Workshop on Parallel and Distributed Processing*, pages 194–200, January 1995.

[CGJ$^+$01] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In Reinhard Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2001.

[CGS13a] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *2013 IEEE Sixth In-*

ternational Conference on Software Testing, Verification and Validation, pages 154–163, March 2013.

[CGS13b] Maria Christakis, Alkis Gotovos, and Konstantinos F. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *ICST*, pages 154–163. IEEE Computer Society, 2013.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

[CJT15] Larry B. Christensen, R. Burke Johnson, and Lisa A. Turner. *Research Methods, Design, and Analysis; 12th Edition, Global Edition.* Pearson / Addison Wesley, 2015.

[CMMRT18] Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit. Declarative debugging of concurrent erlang programs. *Journal of Logical and Algebraic Methods in Programming*, 101:22 – 41, 2018.

[CPS97] J-L. Colaco, M. Pantel, and P. Sallé. *A Set-Constraint-based analysis of Actors*, pages 107–122. Springer, 1997.

[CPS$^+$09] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding Race Conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 149–160. ACM, 2009.

[CS63] Donald T. Cambell and Julian Stanley. *Experimental and quasi-experimental designs for research.* Rand McNally & Company, Chicago, IL, 1963.

[CS10] Maria Christakis and Konstantinos Sagonas. Static Detection of Race Conditions in Erlang. PADL 2010, pages 119–133, January 2010.

[CS11a] Maria Christakis and Konstantinos Sagonas. Detection of Asynchronous Message Passing Errors Using Static Analysis. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages: 13th International Symposium,*, PADL 2011, pages 5–18. Springer, January 2011.

[CS11b] Maria Christakis and Konstantinos Sagonas. Static Detection of Deadlocks in Erlang. Technical report, June 2011.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[da 92] Fabio Q. B. da Silva. *Correctness proofs of compilers and debuggers: an approach based on structural operational semantics.* PhD thesis, University of Edinburgh, UK, 1992. British Library, EThOS.

[DF98] Mads Dam and Lars-ρ ake Fredlund. On the Verification of Open Distributed Systems. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, SAC '98, pages 532–540. ACM, 1998.

[Dij65] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[Dij68] Edsger Wybe Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[DKO13] Emanuele D'Osualdo, Jonathan Kochems, and C. H. Luke Ong. Automatic verification of erlang-style concurrency. In Francesco Logozzo and Manuel Fähndrich, editors, *20th International Symposium on Static Analysis*, SAS 2013, pages 454–476. Springer, June 2013.

[DKVCDM16] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: A taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, pages 31–40. ACM, 2016.

[Doc21] ScalaIDE Documentation. Asynchronous debugger. `http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html`, 2021. Online; accessed 21 March 2021.

[DP02] Fabien Dagnat and Marc Pantel. Static analysis of communications in erlang programs, November 2002.

[Dra13] Iulian Dragos. Stack Retention in Debuggers For Concurrent Programs, July 2013.

[dVSH$^+$18] Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *Art Sci. Eng. Program.*, 2:14, 2018.

[Eng12] J. Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6, 2012.

[fED] Scala IDE for Eclipse Documentation. Asynchronous debugger. `http://scala-ide.org/docs/current-user-doc/features/async-debugger/index.html`. Online; accessed 5 March 2021.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[FGN$^+$03] Lars-Ake Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. A verification tool for erlang. *STTT*, 4(4):405–420, 2003.

[FGST08] Gian Luigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Debugging distributed systems with causal nets. *ECEASST*, 14:1–10, 2008.

[FT01] Gian Luigi Ferrari and Emilio Tuosto. A debugging calculus for mobile ambients. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, SAC '01, page 2, New York, NY, USA, 2001. ACM.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Softw. Pract. Exp.*, 16(3):225–233, 1986.

[GBNDM14] Elisa Gonzalez Boix, Carlos Noguera, and Wolfgang De Meuter. Distributed debugging for mobile networks. *Journal of Systems and Software*, 90:76–90, 2014.

[GCS11] Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. Test-driven development of concurrent programs using concuerror. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 51–61. ACM, 2011.

[GGL$^+$13] Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, pages 394–411, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GLM14] Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-consistent reversible debugging. In Stefania Gnesi and Arend Rensink, editors, *FASE*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014.

[glo90] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[GPT06] Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux. Static safety for an actor dedicated process calculus by abstract interpretation. In Roberto

Gorrieri and Heike Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems*, FMOODS 2006, pages 78–92. Springer, June 2006.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Fifth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986, Los Angeles, California, USA, January 13-15, 1986, Proceedings*, pages 3–12. IEEE Computer Society, 1986.

[Gra21] GraalVM. Getting started with instruments in graalvm. `https://www.graalvm.org/graalvm-as-a-platform/implement-instrument/#simple-tool`, 2021. Online; accessed 15 March 2021.

[Hal15] Philipp Haller. High-level concurrency libraries: Challenges for tool support, October 2015.

[HB11] John M. Hughes and Hans Bolinder. Testing a database for race conditions with quickcheck. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 72–77. ACM, 2011.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[Hei27] W. Heisenberg. Über den anschaulichen inhalt der quantentheoretischen kinematik und mechanik. *Zeitschrift für Physik*, 43(3):172–198, 1927.

[HO09] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, February 2009.

[Hol18] Yan Holtz. Boxplot. `https://www.r-graph-gallery.com/boxplot.html`, 2018. Online; accessed 4 April 2021.

[HPK14] Shin Hong, Yongbae Park, and Moonzoo Kim. Detecting Concurrency Errors in Client-Side Java Script Web Applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, pages 61–70. IEEE, March 2014.

[HPR16] Thomas Haigh, Mark Priestley, and Crispin Rope. *ENIAC in Action: Making and Remaking the Modern Computer.* The MIT Press, 2016.

[HS11] Philipp Haller and Frank Sommers. *Actors in Scala - concurrent programming for the multi-core era.* artima, 2011.

[Huc99] Frank 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, pages 261–272, New York, NY, USA, 1999. ACM.

[HZ18] Brandon Hedden and Xinghui Zhao. A comprehensive study on bugs in actor systems. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[KG76] A. Kay and A. Goldberg. SMALLTALK-72 Instruction Manual. Technical Report SSL 76-6, Xerox Palo Alto Research Center, Palo Alto, California, 1976.

[KM08] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. ICSE '08, pages 301–310, New York, NY, USA, 2008. Association for Computing Machinery.

[KM10] A. J. Ko and Brad A. Myers. Extracting and answering why and why not questions about java program output. *ACM Trans. Softw. Eng. Methodol.*, 20(2):4:1–4:36, 2010.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[LBL$^+$16] Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. On ordering problems in message passing software. In *Proceedings of the 15th International Conference on Modularity*, pages 54–65. ACM, 2016.

[LBM15] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. Accurate and efficient object tracing for java applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 51–62, New York, NY, USA, 2015. Association for Computing Machinery.

[LC06] Yong Luo and Olaf Chitil. Proving the correctness of algorithmic debugging for functional programs. In Henrik Nilsson, editor, *Trends in Functional Programming*, volume 7 of *Trends in Functional Programming*, pages 19–34. Intellect, 2006.

[LDMA09] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul 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, pages 468–479, Washington, DC, USA, 2009. IEEE Computer Society.

[LHA18] Sihan Li, Farah Hariri, and Gul Agha. Targeted test generation for actor systems. In Todd D. Millstein, editor, *ECOOP*, volume 109 of *LIPIcs*, pages 8:1–8:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[Lig21] Lightbend. Cinnamon 2.6 released, now supports vizceral. `https://www.lightbend.com/blog/lightbend-cinnamon-viczeral`, 2021. Online; accessed 12 April 2021.

[LKMA10] Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul 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, pages 363–364. ACM, 2010.

[LLL14] He Li, Jie Luo, and Wei Li. A formal semantics for debugging synchronous message passing-based concurrent programs. *Science China Information Sciences*, 57(12):1–18, Dec 2014.

[LLLG16] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 517–530, New York, NY, USA, 2016. Association for Computing Machinery.

[LM18] Maya Lekova and Benedikt Meurer. Faster async functions and promises. `https://v8.dev/blog/fast-async`, November 2018. Online; accessed 21 March 2021.

[LMBM18] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A study of concurrency bugs and advanced development support for actor-based programs. In Alessandro Ricci and Philipp Haller, editors, *Programming with Actors - State-of-the-Art and Research Perspectives*, volume 10789 of *Lecture Notes in Computer Science*, pages 155–185. Springer, 2018.

[LNPV18] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. CauDEr: A Causal-Consistent Reversible Debugger for Erlang. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming*, volume 10818 of *FLOPS'18*, pages 247–263, Cham, 2018. Springer.

[LPD$^+$14] Kasper Soe Luckow, Corina S. Pasareanu, Matthew B. Dwyer, Antonio Filieri, and Willem Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ASE*, pages 575–586. ACM, 2014.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, March 2008.

[LSM$^+$19] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:30, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[LSX$^+$17] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jin Song Dong. Feedback-based debugging. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *ICSE*, pages 393–403. IEEE / ACM, 2017.

[MAS92] Shakuntala Miriyala, Gul Agha, and Yamina Sami. Visualizing actor programs using predicate transition nets. *Journal of Visual Languages & Computing*, 3(2):195–220, 1992.

[McC61] John 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), pages 225–238, New York, NY, USA, 1961. ACM.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, December 1989.

[Mic21] Microsoft. Debug adapter protocol. `https://github.com/Microsoft/vscode-debugadapter-node/tree/main/protocol`, 2021. Online; accessed 22 March 2021.

[MLA$^+$17] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A concurrency-agnostic protocol for multi-paradigm concurrent debugging tools. *CoRR*, abs/1706.00363, 2017.

[MM15] Stefan Marr and Hanspeter Mössenböck. Optimizing communicating event-loop languages with truffle, October 2015.

[MOM18] Aman Shankar Mathur, Burcu Kulahcioglu Ozkan, and Rupak Majumdar. Idea: An immersive debugger for actors. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*, Erlang 2018, pages 1–12, New York, NY, USA, 2018. Association for Computing Machinery.

[MTS05] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. Concurrency among strangers. In Rocco De Nicola and Davide Sangiorgi, editors, *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer, 2005.

[NA96] Brian Nielsen and Gul Agha. Semantics for an actor-based real-time language. In *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, WPDRTS '96, page 223, USA, 1996. IEEE Computer Society.

[Not17] W3C Working Group Note. Web workers. `https://www.w3.org/TR/workers/`, 2017. Online; accessed 14 February 2017.

[NPV16] Naoki Nishida, Adrián Palacios, and Germán Vidal. A reversible semantics for erlang. In Manuel V. Hermenegildo and Pedro López-García, editors, *LOPSTR*, volume 10184 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2016.

[Pac11] David Pacheco. Postmortem debugging in dynamic environments. `https://queue.acm.org/detail.cfm?id=2039361`, October 2011. Online; accessed 26 April 2021.

[PGB$^+$05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice.* Addison-Wesley Professional, 2005.

[PGR$^+$15] Sushil K. Prasad, Anshul Gupta, Arnold L. Rosenberg, Alan Sussman, and Charles C. Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[PSTH16] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1):83–110, 2016.

[PVSD12] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *ACM SIGPLAN Notices*, volume 47, pages 251–262. ACM, 2012.

[RVS13] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 151–166. ACM, 2013.

[SA06] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In Luciano Baresi and Reiko Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[Sag05] Konstantinos Sagonas. Experience from developing the dialyzer: A static analysis tool detecting defects in erlang applications. In *Proceedings of the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[Sag10] Konstantinos Sagonas. Using static analysis to detect type errors and concurrency defects in erlang programs. In *International Symposium on Functional and Logic Programming*, pages 13–18. Springer, 2010.

[SBSB19] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. Reasoning about the node.js event loop using async graphs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 61–72. IEEE Press, 2019.

[SCC01] W. R. Shadish, T. D. Cook, and Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2 edition, 2001.

[SCM09] Terry Stanley, Tyler Close, and Mark Miller. Causeway: A message-oriented distributed debugger. Technical report, HP Labs, April 2009.

[sig83] *SIGSOFT '83: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, New York, NY, USA, 1983. Association for Computing Machinery.

[SLM$^+$19] Robbert Gurdeep Singh, Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Artifact). *Dagstuhl Artifacts Series*, 5(2):4:1–4:3, 2019.

[SM16] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *ICSE*, pages 796–807. ACM, 2016.

[SNDMDR17] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. Mailbox abstractions for static analysis of actor programs (artifact). *DARTS*, 3(2):11:1–11:2, 2017.

[Sut05] Herb Sutter. The free lunch is over. `http://www.gotw.ca/publications/concurrency-ddj.htm`, March 2005. Online; accessed 25 April 2021.

[SW17] Kazuhiro Shibanai and Takuo Watanabe. Actoverse: A reversible debugger for actors. 2017.

[TGMJ11] Samira Tasharofi, Milos Gligoric, Darko Marinov, and Ralph Johnson. Setac: A Framework for Phased Deterministic Testing Scala Actor Programs, 2011.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[TKL$^+$12] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In Holger Giese and Grigore Rosu, editors, *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*, pages 219–234. Springer, 2012.

[TLGBS$^+$17] Carmen Torres Lopez, Elisa Gonzalez Boix, Christophe Scholliers, Stefan Marr, and Hanspeter 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, pages 41–49. ACM, October 2017.

[TLMMGB16] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. Towards Advanced Debugging Support for Actor Languages: Studying Concurrency Bugs in Actor-based Programs, October 2016.

[TLSZ19] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 865–878, New York, NY, USA, 2019. Association for Computing Machinery.

[TPLJ13] Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph E. Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 114–124, November 2013.

[TV10] Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, Nov 2010.

[Val98] Antti Valmari. *The state explosion problem*, chapter 9, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[VBMM18] John Vilk, Emery D. Berger, James Mickens, and Mark Marron. Mcfly: Time-travel debugging for the web. *CoRR*, abs/1810.11865, 2018.

[VCGBS+14] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(3-4):112–136, 2014.

[Ver20] Louise Van Verre. Interrogative debugging for somns programs, 2020. Bachelor thesis.

[VMG+07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12. IEEE Computer Society, 2007.

[VMV17] Simon Van Mierlo and Hans Vangheluwe. Debugging non-determinism: a petrinets modelling, analysis, and debugging tool. In *CEUR workshop proceedings*, volume 2019, pages 460–462, 2017.

[VRH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*, volume 19. 01 2004.

[WDG+17] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in node.js. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 520–531, 2017.

[web16] akka-visualmailbox. `https://github.com/ouven/akka-visualmailbox`, 2016. Online; accessed 12 April 2021.

[Wis97] Roland Wismüller. Debugging message passing programs using invisible message tags. In Marian Bubak, Jack Dongarra, and Jerzy Waśniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 295–302, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[WWH$^+$17] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. Association for Computing Machinery.

[WWS$^+$12] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. Association for Computing Machinery.

[WWW$^+$13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. Association for Computing Machinery.

[XBLL16] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. "automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *ICSME*, pages 267–278. IEEE Computer Society, 2016.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in abcl/1. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 258–268, New York, NY, USA, 1986. Association for Computing Machinery.

[ZBZ11] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 805–814. ACM, 2011.

[Zel09] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[ZWCZ15] Mingxing Zhang, Yongwei Wu, Kang Chen, and Weimin Zheng. What is wrong with the transmission? a comprehensive study on message passing related bugs. In *ICPP*, pages 410–419. IEEE Computer Society, 2015.

Glossary

actor an actor can be defined as a four-tuple: an execution context, an inbox, an interface and its state. An actor perpetually takes messages from its inbox and processes them in a new execution context with respect to that actor's interface and state. This continues until the inbox is empty after which the actor goes to an idle state until a new message arrives in its inbox [DKVCDM16].

behavior denote the combination of an actor's interface an its state. Some actor systems enable an actor to modify its entire behavior in one single operation. [DKVCDM16].

calling context history of synchronous message calls that lead to a pausing state in a program.

concurrent a concurrent program has multiple logical threads of control. These threads may or may not run in parallel [**?**].

cyclic debugging cyclic debugging is the debugging process of stopping the program's execution, inspect the variables state and resume execution, or stop again in another execution point [MH89].

dependent variable is a presumed effect or outcome. The dependent variable is influenced by one or more independent variables [CJT15].

dynamic deoptimization occurs when specialization fails, i.e., the type of the node is not longer valid, then compiled code is reverted to the AST interpreter [WWW$^+$13].

experimental condition is defined by one level of manipulation of the independent variable, is also referred as treatment condition in experimental research [CJT15].

far reference object that references another object within *another* actor. These objects communicate through asynchronous messages [MTS05].

independent variable is the presumed cause of another variable [CJT15].

interface at any given point in time, an actor's interface defines the list and types of messages it understands. An actor can only process incoming messages that fit this interface. For some actor systems this interface is fixed while other actor systems allow an actor to change its interface, thus allowing it to process different types of messages at different points in time [DKVCDM16].

mailbox stores an ordered set of messages received by an actor [DKVCDM16].

near reference object that references another object within the *same* actor (aka, a direct reference). These objects communicate through synchronous messages [MTS05].

node specialization means that when the nodes are executed, i.e., during interpretation, they are replaced by type-specific nodes, e.g., integer nodes, double nodes [WWW+13].

parallel parallel programming is about using additional computational resources to produce an answer faster [**?**]. A parallel program potentially runs more quickly than a sequential program by executing different parts of the computation simultaneously (in parallel). It may or may not have more than one logical thread of control [**?**].

partial evaluation is the process of creating the initial high-level compiler intermediate representation (IR) for a guest language function from the guest language interpreter methods (code) and the interpreted program (data) [WWH+17].

reverse debugging a debugging technique in which a history of program execution is recorded and then replayed under the user's control, in either the forward or backward direction [glo90].

send context history of asynchronous message calls that lead to a pausing state in a program.

state all the state that is synchronously accessible by an actor (i.e., state that can be read or written without blocking its thread of control). Depending on the implementation, that state can be mutable or immutable, and isolated or shared between actors [DKVCDM16].

turn a turn is defined as the processing of a single message by an actor. In other words, a turn defines the process of an actor taking a message from its inbox and processing that message to completion [DKVCDM16].