A Principled Approach towards Debugging Communicating Event-Loops

Carmen Torres Lopez
Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium
{ctorresl,egonzale}@vub.be

Christophe Scholliers
Universiteit Gent
Ghent, Belgium
christophe.scholliers@ugent.be

Stefan Marr
Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
{stefan.marr,hanspeter.moessenboec
k}@jku.at

Abstract

Since the multicore revolution, software systems are more and more inherently concurrent. Debugging such concurrent software systems is still hard, but in the recent years new tools and techniques are being proposed. For such novel debugging techniques, the main question is how to make sure that the proposed techniques are sufficiently expressive.

In this paper, we explore a formal foundation that allows researchers to identify debugging techniques and assess how complete their features are in the context of message-passing concurrency.

In particular, we describe a principled approach for defining the operational semantics of a debugger. Subsequently, we apply this technique to derive the operational semantics for a communicating event-loop debugger. We show that our technique scales for defining the semantics of a wide set of novel breakpoints recently proposed by systems such as REME-D and Kómpos. To the best of our knowledge, this is the first formal semantics for debugging asynchronous message passing-based concurrency models.

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

Keywords Debugging, Concurrency, Actors, Breakpoint, Stepping, Operational Semantics

ACM Reference format:
Carmen Torres Lopez, Elisa Gonzalez Boix, Christophe Scholliers, Stefan Marr, and Hanspeter Mössenböck. 2017. A Principled Approach towards Debugging Communicating Event-Loops. In *Proceedings of 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, Vancouver, Canada, October 23, 2017 (AGERE'17),* 9 pages.
https://doi.org/10.1145/3141834.3141839

1 Introduction

A recent field study on state-of-the-art in debugging [17] has shown that debugging parallel applications is especially hard and may require new specialized tools and methods. This study confirms that one of the main difficulties of fixing bugs in parallel and concurrent software is the large distance between the root cause of a bug and the observed failure. However, traditional debuggers available in modern IDEs are designed for sequential programs in which statements execute one after the other possibly modifying some memory state. This makes them barely useful for concurrent and parallel programs. Designing a debugger for concurrent and parallel programs requires rethinking features like breakpoint and stepping semantics, inspecting program changes, etc. This task is challenging because the design space is huge and includes many parameters such as the non-determinism of concurrent processes, lack of global clock and the probe effect [15].

In the past years, several new and promising debugging tools for concurrent programs have emerged including Causeway [21], REME-D [2], Graft [18], BigDebug [7] and others [8, 11]. Typically those tools provide domain-specific debugging features specially designed for the specific characteristics of parallelism and concurrency. For example, the REME-D debugger explored the design space of breakpoints and stepping semantics for an online debugger for distributed actor-based programs [2]. BigDebug, on the other hand, has proposed the concept of *simulated breakpoints* for parallel Big Data programs, which do not stop program execution but they store all necessary information so that developers can later replay execution from that point on [18].

Once a novel debugging technique is designed, the question is how to validate that the technique or tool is useful and aids the hard task of finding the root cause of bugs in concurrent software. A review of recent debugging publications shows that the usual validation methodology is to conduct a user study to assess usability of the tool and its debugging features, e.g. in [2, 19]. However, conclusions from those studies heavily rely on the insights and experience of programmers involved in the experiments. Sometimes it is possible to conduct performance benchmarking to assess the feasibility of the approach (such as in [18]), but this does not provide insights on the usability and suitability of the approach. An alternative approach is to use formal semantics to prove the correctness or suitability of the proposed debugging features. A formal foundation provides developers with a principled approach to identify the design space and assess how complete the set of debugging features is.

In this paper, we design a semantic framework to debug concurrent programs employing the communicating event-loop (CEL) model. We start from the operational semantics for a concurrent programming language model which features communicating event-loop concurrency, namely AmbientTalk [22]. We then propose a novel operational semantics for a debugger for communicating event-loop programs inspired by the work of da Silva [4] on defining the relational semantics of sequential debuggers. Based on this semantics, we model the breakpoint catalog present in AmbientTalk's debugger (i.e. REME-D).

To the best of our knowledge this is the first formal semantics for debugging asynchronous message passing-based concurrency models. Previous semantic frameworks proposed for debugging have focused on a sequential functional programming language [1], or message-passing programming languages employing synchronous communication models [5, 6, 12].

2 Debugging Communicating Event-Loop Programs

This section briefly recapitulates the basic ideas around communicating event-loop concurrency, and debugging approaches for communicating event-loop languages. Finally, we discuss breakpoint semantics explored in prior online debuggers developed for those languages [2, 14], which we will model in our semantics framework for debugging presented later in section 4.3.

2.1 Communicating Event-Loop in a Nutshell

The communicating event-loop model is a non-blocking concurrency actor model proposed by Miller et al. for the E language [16]. In this model an actor or *vat* is defined by a thread of control, a heap of objects, a stack, and a mailbox. Van Cutsem et al. [23] summarized three main properties of this model. The first property states that actors execute sequentially messages from their mailbox, i.e. messages are processed one by one. An important concept here is the notion of a *turn*, which consists in the processing of one message by the actor until completion. The second property states that actors have the exclusive access to their mutable state. This means that objects are owned by actors and only the owner can access it. The third property states that actors communicate using asynchronous messages, i.e. an actor never waits for another actor to finish a computation because communication between actors is non-blocking. These three properties makes communicating event-loop languages free from any kind of low-level data races by design.

Communicating event-loop languages typically support futures or promises, which are placeholders for the later return value of an asynchronous message. Their values can be accessed only using callbacks, which are scheduled as separate turns on the actor, to avoid exposing the low-level data race of a promise resolution.

2.2 Debugging Communicating Event-Loop Languages

The first debugging tool for communicating event-loop languages is Causeway [21], a message-oriented distributed debugger for the E language. Causeway is a post-mortem debugging tool which records a trace of events (i.e. message sends and receptions) generated during the application's execution, which is then loaded by the debugger UI. Causeway provides message and process order views to browse the recorded event history, allowing developers to establish a partial order of the event history based on the *happened before* relation [10]. This relation shows how message sends and reception events potentially affect each other, helping developers to identify potential places that caused a bug and as such, offering a similar functionality as stack traces in sequential debuggers.

Post-mortem debugging techniques, however, have been widely criticized since manually inspecting traces becomes cumbersome and difficult [15]. Many debugging efforts have thus focused on breakpointed-based online debuggers which allow programers to control program execution and inspecting program state by performing step-by-step execution.

The first online debugging tool for communicating event-loop languages is REME-D [2], a reflective epidemic message-oriented debugger for the AmbientTalk language [22]: a distributed language for developing mobile peer-to-peer applications which uses communicating event-loops as concurrency model. REME-D is a breakpointed-based debugger which introduced the first catalog of message-oriented breakpoints for event-loop programs. It also adapts Causeway's event histories based on the happened before relation to a breakpoint-based debugger by allowing developers to browse causal links for messages in the current execution context. REME-D also includes specific features for debugging distributed systems with changing communication topologies such as a decentralized debugging session, and open debugging sessions (i.e. the debugger can attach to a running application and devices can be incorporated dynamically into a debugging session at run time).

More recently, Kómpos [14] has explored message-oriented debugging for SOMns [13], a Newspeak implementation [3], which also uses communicating event-loops as its concurrency model. Compared to REME-D, it does not focuses on distributed applications but on concurrent ones which enables the debugger to provide a consistent global view of the system easily. Furthermore, SOMns is focused on an efficient implementation to minimize interference of the debugger with the running application.

The work in communicating event-loop debuggers has been inspired by prior work in debuggers for message passing communication models like MPI. The most relevant work related to MPI applicable to communicating event-loop model is Wismüller's *message breakpoints* [24]. A message breakpoint stops all receiver processes of the next message sent by a process. The combination of a message breakpoint with a traditional breakpoint on the send statements provides similar semantics to REME-D's step-into-message-send command. REME-D's breakpoint catalog transcends Wismüller's message breakpoints since it also provides breakpoint semantics for future type message passing.

2.3 Breakpoints for Communicating Event-Loop Languages

REME-D proposed the first catalog of message-oriented breakpoints for event-loop programs [2]. The catalog included three dimensions to define message-oriented breakpoints: (1) the place where the execution of the program should be suspended i.e. sender or receiver, (2) the moment when the execution of the program is suspended for the processing of a message i.e. before or afterwards, and (3) how a breakpoint can be defined in the program code e.g. using line numbers, or predicates. That catalog served as inspiration to Kómpos, whose breakpoints are very similar.

In addition to breakpoints, debuggers usually provide support for stepping between relevant execution points. We consider stepping as the transition from one breakpoint to another one. In each breakpoint location, a possible stepping operation can be thus performed. In the context of event-loops, the stepping operations allow developers to reason and control a program at the level of messages and turns.

As illustration, fig. 1 shows the points of interests for debugging in a communicating event-loop concurrency model. In particular, it shows the points of interest involved in the sending of a message from a sender object A to a receiver object B hosted in another actor.

For regular asynchronous messages, we consider breakpoints either on the sender or the receiver end of a message send (denoted as point 1 and 2 in fig. 1). Both breakpoints halt before executing the corresponding send or receive operation. From point 1 in the program it is then possible to step to the message receiver (point 2).

For future messages, i.e. asynchronous messages returning immediately a future (denoted as object F in fig. 1). These messages add two more points of interest for debugging: the point after the value for the future got determined and before it is sent to resolve the future itself (point 3), and the point before any callbacks that were registered on the future are executed, i.e. before future resolution (point 4). These points are considered as stepping targets as well. In addition, in a

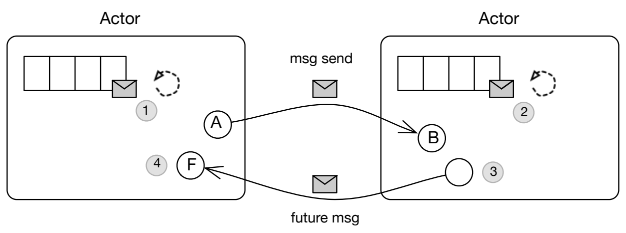

Figure 1. Points of interest for debugging messages exchanged by actors as communicating event-loops. Object A sends a message to Object B. The message returns a future. When the future is resolved a callback is executed. Both objects are located in different actors.

turn, we can step to the resolution using the turn's result value.

Table 1 summarizes all breakpoints and stepping operations considered in this paper. For flexibility, we also explore breakpoints at the beginning and end of each turn, which allows developers to step between turns on a single actor.

3 Formalization of the Concurrent Event-Loop Model

Bernstein and Stark were the first ones to develop the operational semantics for a debugger on top of a programming language definition [1]. Similar to their approach, our semantic framework to debug communicating event-loop programs (cf. Section 4) is defined on top of the operational semantics for the AmbientTalk language defined by Van Cutsem et al. [22]. We briefly introduce the key aspects of the semantics which are necessary to follow the contributions of our work.

AmbientTalk's semantics is based on JCoBox's semantics [20] which was adapted to formalize language features common to event-loop based languages such as actors, objects, blocks, non-blocking functions and asynchronous message sending [22]. In a nutshell, AmbientTalk's semantics considers that actors evaluate messages as expressions, to obtain a result value. Reduction rules of AmbientTalk semantics can be applied in a non-deterministic way, and they correspond to concurrency properties of the communicating event-loop model.

Figure 2 shows the semantic entities for AmbientTalk semantics 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 outbox queue Q_{out} that, for each remote actor ι_a, stores all outgoing messages addressed to objects owned by ι_a. The outbox queue Q_{out} actually stores envelopes l. An envelope is a message and the set of objects that represent the receivers of the message. A network in which actors communicate is identified by ι_n and e is the expression the actor is executing.

Table 1. Overview of breakpoints and stepping operations inspired by REME-D's breakpoint catalog. Their support in REME-D and Kómpos debuggers, and our formalization is indicated in the table.

Message Type	Breakpoints	Stepping	Activation		Execution		Supported		
			Sender	Receiver	Before	After	REME-D	Kómpos	Semantics
Regular Message	Message sender		X		X			X	X
	Message receiver	step-to-msg-rcvr		X	X		X	X	X
Future Message	Future resolver	step-to-future-resolver		X		X		X	X
	Future resolution	step-to-future-resolution, return-from-turn-to-future-resolution	X	X		X	X	X	X
Both	Begin turn	step-to-next-turn		X	X		X	X	X
	End turn	step-end-turn		X		X		X	X

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.e. isolates I. A future is a first-class placeholder for an asynchronously awaited value and 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 $e \leftarrow m(\overline{e})$.

$a \in A \subseteq$ **Actor**	$::=$	$\mathcal{A}\langle \iota_a, O, Q_{in}, Q_{out}, \iota_n, e\rangle$	Actors
Object	$::=$	$O\langle \iota_o, t, F, M\rangle$	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
$Q_{out} \in$ **Outbox**	$::=$	$\iota_a \mapsto \overline{l}$	Outbox queues
$l \in$ **Envelope**	$::=$	$(\mathrm{m}, O_{\mathrm{m}})$	Envelopes
$M \subseteq$ **Method**	$::=$	$m(\overline{x})\{e\}$	Methods
$F \subseteq$ **Field**	$::=$	$f := v$	Fields
$v \in$ **Value**	$::=$	$r \mid \text{null} \mid \epsilon$	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 2. Semantic entities of $\textsc{at}^f$.

4 A Semantic Framework to Debug Communicating Event-Loop Programs

This section describes the formalization approach of breakpoints and stepping operations for communicating event-loop programs. We first show how to define the operational semantics of a debugger given the operational semantics of a CEL programming language. We then use this technique to extend the semantics for AmbientTalk $\textsc{at}^f$ described in the previous section with reduction rules for CEL debugging. We have implemented the presented semantics in PLT-Redex [9] [1].

4.1 General Design of the Formalization

This section gives an overview of how to design the formal semantics of a debugger on top of the operational semantics of a programming language (called the *base language* in the remainder of this paper). The semantics of the debugger is modeled as a function, which, when applied to a debugging state (which includes the program being debugged), yields a new debugging state.

In general the state of the debugger can contain anything which is needed to denote the semantics of the modeled breakpoints. In order to model a wide set of CEL breakpoints as shown in table 1, it is sufficient to have the debugger state $\mathcal{D}$ consisting of five parts, $\mathcal{D}\langle B_p, B_c, s, H, K\rangle$. In what follows, we explain these five parts which are the core of the abstract model for a CEL debugger.

In order to keep track of which breakpoints the debugger has already checked and which breakpoints it should still check we choose to explicitly keep track of two (breakpoint) lists. The reason to choose for a list representation is to make the order in which the breakpoints are checked explicit. When modeling a non-deterministic debugger it could be better to have a set representation. The first list (B_p) keeps track of the pending breakpoints, i.e breakpoints which the debugger still needs to verify for a particular execution state. The second list (B_c) keeps track of which breakpoints have already been checked by the debugger.

[1]Our implementation is available http://users.ugent.be/~chscholl/Debug/AmbientTalk-D.zip

For each execution state of the program being debugged, the debugger traverses the list of pending breakpoints and performs an action. An action is a combination of updating the debugger state, halting the program execution, stepping the program execution, and moving the breakpoints from the list of pending breakpoints to the list of checked breakpoints.

To keep track of which action the debugger is performing, the debugger state contains an action tag s (representing step, halt, ...).

When modeling complex breakpoints the current state of the execution of the debugged program might not provide sufficient information to determine whether the breakpoint is applicable or not. In order to be able to model complex breakpoints the state of the debugger keeps track of the execution history H of the debugged program. Note that for a particular debugger the partial history of the execution history, for instance only tracking messages, may be sufficient to determine when a breakpoint applies.

The last part of the debugger state corresponds to the execution state of the program being debugged (K).

4.2 Debugger State for a CEL Debugger

The previous section introduced an abstract model for debugging based on the debugger state $\mathcal{D}$. This section shows how the five parts of the debugging state $\mathcal{D}$ are instantiated for a concrete CEL debugger. Figure 3 summarises the semantic extensions to the base programming language semantics AT^f for CEL debugging. In general, the debugger state $\mathcal{D}$ consists of a list of pending breakpoints B_p, a list of checked breakpoints B_c, an action tag s, a list of histories H, and a program state K.

We define CEL breakpoints as two-tuples consisting of a breakpoint tag b_t and an expression id ι_e. The breakpoint tag is used to denote which kind of breakpoint the user has defined. The expression id uniquely determines the AST node over which the breakpoint is defined. For example, a message receiver breakpoint over a message send expression with id ι_i is defined as $\mathcal{B}\langle mrb, \iota_i\rangle$. This also implies that the underlying programming language semantics needs to include id tags on the relevant AST nodes.

For the breakpoints defined in this paper it is sufficient to add id tags on send expressions and messages. To this end, we have extended the semantic entities presented in Figure 2 with two modifications (also shown in Figure 3). First, we added an identifier ι_m in the message entity m to identify the message. Second, we added an *id* in the send expression. We use the *id* of the send expression as metadata to identify which message is breakpointed. This identifier is needed because different breakpoints can be set on the same message.

For our simple debugger we only consider two action tags: *check* and *stop*. When the debugger is in the check state it verifies whether there is an applicable breakpoint. When a breakpoint applies, the debugger transitions itself to the *stop* state and halts execution.

In general, the history H of the debugger is represented by the different actor configurations $\overline{K}$ of the program. In our formalization of the rules we observed that a partial history consisting of a set of messages $\overline{m}$ is sufficient.

$d \in$ **Debugger**	::=	$\mathcal{D}\langle B_p, B_c, s, H, K\rangle$	Debuggers
$b \in$ **Breakpoint**	::=	$\mathcal{B}\langle b_t, \iota_i\rangle$	Breakpoints
$s \in$ **Actions**	::=	check | stop	Action Tags
$h \in$ **History**	::=	$\overline{m}$	Histories
$b_t \in$ **Breakpoint tag**	::=	msb | mrb | fvb	Tags
		fsb | abb | aab	
	$\iota_i \in$ **BreakpointId**		
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 3. Semantic extensions for CEL debugging.

4.3 Formalizing CEL Debuggers

In this section we give an overview of the formalization of the CEL debugger. Figure 4 shows an overview of selected reduction rules for the CEL debugger. The general form of the reduction rules of the debugger consists of transitions between debugger states:

$$\mathcal{D}\langle B_p, B_c, s, H, K\rangle \rightarrow_d \mathcal{D}\langle B_p', B_c', s', H', K'\rangle$$

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 inspecting the list of pending breakpoints B_p one-by-one from left to right. When the actor configuration K is in a state which triggers the first breakpoint of the list, the debugger moves to the stopped state. When the breakpoint is not triggered by the actor configuration, it is moved from the pending list to the checked list.

The reduction rules of the CEL debugger can be categorised into four classes:

1. Reduction rules for modeling the connection of the debugger with the base language.
2. Reduction rules for modeling history independent breakpoints.
3. Reduction rules for modeling history dependent breakpoints.
4. Bookkeeping reduction rules for transitioning the state of the debugger when the head of the pending breakpoints list is not applicable.

In what follows, we describe each of the rules according to the category it belongs to.

4.3.1 Connection of the Debugger with the Base Language

The semantics of the debugger is defined in terms of the underlying base language. The debugger has the whole actor configuration as part of its state and calls upon the base language to take one evaluation step. There is only one rule involved in connecting the debugger with the base language.

CEL-STEP When the debugger has transitioned into a state where the pending breakpoints list B_p is empty the debugger transitions into a new debugging state where the actor configuration K is updated by reducing it one step with the underlying one-step evaluation relation of the base language. Second, the list of checked breakpoints is reinstalled as the list of pending breakpoints B_p and the list of checked breakpoints B_c is reset to the empty list.

4.3.2 History-Independent Breakpoints

We differentiate the set of breakpoints into history-independent and history-dependent breakpoints. The most simple breakpoints only require the actor configuration K in order to determine whether they should be triggered while more bookkeeping is required for the history dependent breakpoints. In the current set of CEL breakpoints we only identified one history-independent breakpoint (message sender breakpoint).

TRIGGER-MSB In order to trigger the message sender breakpoint there needs to be an actor in the actor configuration K which is about to send a message. The *id* of the message send operator needs to be the same as the identifier ι_i of the breakpoint corresponding to the message sender $\mathcal{B}\langle msb, \iota_i\rangle$. When this is the case, the breakpoint is removed from B_p and is added to checked breakpoints B_c. Note that a breakpoint at the sender side of the message sent reduces the debugger $\mathcal{D}$ to a *stop* state, just after the message is created and before adding it to the outbox queue.

4.3.3 History-Dependent Breakpoints

Next to history-independent breakpoints we also model history-dependent breakpoints. We focus on one such breakpoint i.e. message receiver breakpoint, but have observed that asynchronous before and asynchronous after can be modeled completely analogous. Moreover, because the semantics of the underlying language translates futures into regular asynchronous messages the future resolver and future resolution breakpoints can also be modeled with similar reduction rules.

The reason why these breakpoints need to depend on a history is because there are two distinct execution times in the evaluation of the interpreter which are important for determining when to halt execution. For example, for the message receiver breakpoint the debugger needs to know whether the breakpoint was active when the message was sent and later on it needs to remember to halt execution when the message is actually received. In order to model this behavior correctly our semantics relies on the message history.

The semantics of a message receiver breakpoint (MRB) consist of two reduction rules: SAVE-MRB and TRIGGER-MRB. The first rule is applicable when a matching asynchronous message is sent. The debugger then saves this message into the history and uses this message in order to determine whether it should halt execution when the message is received.

SAVE-MRB This rule represents the reduction of the debugger $\mathcal{D}$ at the sender side of a message, when a message receiver breakpoint is set. The breakpoint $\mathcal{B}\langle mrb, \iota_i\rangle$ will be added to the checked breakpoints B_c, the history H is updated with the sent message. Note that the action tag of the debugger is not changed and stays *check* in order to verify whether any other breakpoint is triggered.

TRIGGER-MRB The message receiver breakpoint is triggered when a message (with a message receiver breakpoint) is about to be processed by the receiving actor. In the semantics this means that the message needs to be the first message in the inbox of the receiving actor and that the currently executing expression of the actor has reduced to a value v, i.e. the actor is idle. Finally, the message to be processed should be contained in the history. When all these conditions are met the debugger $\mathcal{D}$ changes its action to *stop* and removes the message from the message history.

4.3.4 Bookkeeping of Pending Breakpoints

For each of the breakpoint triggering rules there should be an "anti rule" which instructs the debugger to move the breakpoint to the list of checked breakpoints. Instead of listing all these individual rules we compressed them into one rule NOT-APPLICABLE-BREAKPOINT which should be triggered when the breakpoint at the head of the list is not applicable.[2]

5 Discussion

Formalizing a debugger as we did in the previous section provided us with a better understanding of the design space for breakpoints and stepping operations. This section first briefly discusses how our approach is applicable to existing languages and its implications for practical language implementations. The section then concludes by contrasting the message breakpoints in previous systems with the design space opened up by using message histories.

Application to Existing CEL Languages. The presented formalization of the debugger is designed to be applicable to a wide range of CEL languages. While we started out from the semantics of AmbientTalk, it focuses on the key

[2]The PLT-Redex semantics list all the rules individually.

(CEL-STEP)

$$\frac{K \rightarrow_k K'}{\mathcal{D}\langle (), B_c, \text{check}, H, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \text{check}, H, K'\rangle}$$

(TRIGGER-MSB)

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, Q_{out}, \iota_n, e_\square[\iota_a.\iota_o \leftarrow_{\iota_i} m(\overline{v})]\rangle \in K}{\mathcal{D}\langle \mathcal{B}\langle msb, \iota_i\rangle \cdot B_p, B_c, \text{check}, H, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle msb, \iota_i\rangle, \text{stop}, H, K\rangle}$$

(SAVE-MRB)

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, Q_{out}, \iota_n, e_\square[\iota_a.\iota_o \leftarrow_{\iota_i} m(\overline{v})]\rangle \in K}{\mathcal{D}\langle \mathcal{B}\langle mrb, \iota_i\rangle \cdot B_p, B_c, \text{check}, H, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle mrb, \iota_i\rangle, \text{check}, H \cdot \mathcal{M}\langle \iota_m, \iota_a.\iota_o, m, \overline{v}\rangle, K\rangle}$$

(TRIGGER-MRB)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, Q_{out}, \iota_n, v\rangle \in K}{\mathcal{D}\langle B_p, B_c, \text{check}, m \uplus H, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{stop}, H, K\rangle}$$

(NOT-APPLICABLE-BREAKPOINT)

$$\frac{not - applicable - breakpoint}{\mathcal{D}\langle \mathcal{B}\langle b_t, \iota_i\rangle \cdot B_p, B_c, \text{check}, H, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle b_t, \iota_i\rangle, \text{check}, H, K\rangle}$$

Figure 4. Reduction rules for breakpoints.

concurrency properties shared among CEL languages and aspects such as the object model can be adapted. Specifically, the semantics should be compatible to languages such as E, Newspeak, and even JavaScript with Web Workers.[3] The main restriction for JavaScript is that messages are passed as JSON, which means they all have pass-by-copy semantics.

Furthermore, the debugger semantics requires only minimal adaptations in the interpreter and thus can be applied to other interpreters straightforwardly. More concretely, the debugger requires that the interpreter (1) annotates AST nodes with a unique id, (2) has a representation of all the actors in the configuration and (3) annotates messages with unique ids. Depending on the interpreter, this may require changes to parse and the code that handles sending and receiving messages in order to tag them correctly.

Implications for Practical Systems. The application of the presented ideas to language systems for practical use is a higher hurdle. Keeping a message history can require major changes to an interpreter and might degrade performance significantly. For these reasons, the changes in SOMns for Kómpos, nor REME-D do not implement the discussed breakpoints based on a history directly. Instead, the chosen set of breakpoints is realized by passing flags on messages to indicate a set breakpoint. This is semantically equivalent to a predicate over the message history with the same properties but has better performance characteristics. On the other hand, predicates over the message history are much more flexible and would allow us to expose the predicate language to the debugger and thereby enable custom and arbitrarily complex breakpoints or stepping strategies.

Design Space for Breakpoints and Stepping based on History Predicates. As discussed in the introduction, an important goal of this work is to better understand the power of debugging abstractions and the completeness of a set of proposed features.

In our prior work on REME-D and Kómpos, we reasoned about the desired breakpoints and stepping operations from our programmer perspective. This can be illustrated with the point of interest in an actor system that we considered as relevant for breakpoints and stepping semantics. For example, fig. 1 depicts the interactions necessary to realize a message send that returns a future. It highlights the four points relevant for breakpoints and stepping as discussed in section 2.3.

While these points cover relevant elements for debugging message exchanges, they are comparably simple. Reconsidering them with the notion of a message history shows that they correspond to only minimal sequence in the history and do not include complex interactions between actors. Furthermore, they are only between two actors. Since our formalization also includes the actor configuration itself, breakpoints could also consider interactions between multiple actors that have complex interaction protocols. However, we did not yet explore the full potential of this approach, neither with concretely formalized breakpoints nor within our practical implementations. Nonetheless, with formalizing the breakpoints we used in REME-D and Kómpos based on the presented approach, we were able to determine that the

[3] *Web Workers API*, MDN web docs, access date: 2017-08-21, https://developer.mozilla.org/en-US/docs/Web/API/Web$_W$orkers$_A$PI

previously proposed breakpoint catalogues are strictly less powerful compared with an approach that uses predicates over message histories.

6 Related Work

To the best our knowledge, this is the first attempt to formalize a debugger for the communicating event-loop model. In this section, we summarize previous work conducted on formalizing debuggers for other programming paradigms.

The first formal specification for debuggers was proposed by da Silva [4]. This formalization based on a structural operational semantics, considers a debugger as a system which transits from one state to another using an evaluation history. The debugging operations are realized as predicates e.g. stepping predicates and path predicates to pause on expressions of an AST. This idea of a transition system based on the history has inspired our work.

In the context of distributed applications Ferrari et al. [5] proposed a debugging calculus for mobile ambients. Similar to Bernstein and Stark [1] and our approach, they model a debugger as an extension of the operational semantics of an underlying programming language. Ferrari et al. [6] proposes Causal Nets which allows the programmer to query a causal message graph generated by the execution of a set of distributed processes. While interesting, this is essential a post-mortem debugger. No explanation is provided of how to build a concrete debugger given an existing formal semantics of the base language.

Li et al. [12] introduced a formal semantics for debugging synchronous message-passing programs e.g. MPI, Occam and JCSP. The proposal shows 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. The semantics proposed uses the notion of transitions of states over trace configurations. In contrast to this work, our semantics model an asynchronous message-passing programs employing communicating event-loop concurrency.

7 Conclusion and Future Work

In this paper, we investigate debugging techniques for communicating event-loop (CEL) languages. In contrast to other studies, we use a formal approach to reason about breakpoints and stepping for debuggers for CEL languages. While this approach will not allow us to make any conclusions about the usefulness from a developer perspective or the acceptance of a tool, the formalization enables us to reason about the huge design space of possible debugging mechanisms, which so far was rarely explored.

Concretely, the formalization proposed in this paper models a debugger as a function, which when applied to a debugging state (which includes the program being debugged), yields a new debugging state. We have found that it is important to keep the history of program executions available in this debugging state in order to define the semantics of advanced (history-dependent) breakpoints. All the advanced breakpoints we investigated can be seen as predicates over the history of a program execution and metadata connecting the history to the program sources.

We have shown that the resulting formalism is powerful enough to define the breakpoints proposed in practical debuggers for asynchronous message passing systems. However, using predicates over the history surpasses the existing approaches to message-based breakpoints we are aware of in terms of expressiveness and the possible breakpoints that are supported.

In the future, we hope this formal approach will help us to determine whether all possibly useful features of a debugger are covered, for instance, with respect to specific kinds of concurrency issues. Furthermore, it remains an open question whether such a general history-based approach to breakpoints and stepping is realistic and does not have unacceptable high overheads for practical implementations.

Acknowledgments

We would like to thank the reviewers of this paper for their comments. Carmen Torres Lopez was funded by a grant of the Research Foundation Flanders (FWO), project number G004816N. Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31.

References

[1] Karen L. Bernstein and Eugene W. Stark. 1995. Operational Semantics of a Focusing Debugger. *Electronic Notes in Theoretical Computer Science* 1 (1995), 13 – 31. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

[2] Elisa Gonzalez Boix, Carlos Noguera, and Wolfgang De Meuter. 2014. Distributed debugging for mobile networks. *Journal of Systems and Software* 90 (2014), 76–90.

[3] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming*. LNCS, Vol. 6183. Springer, 405–428.

[4] Fabio Q. B. da Silva. 1992. *Correctness proofs of compilers and debuggers: an approach based on structural operational semantics.* Ph.D. Dissertation. University of Edinburgh, UK. British Library, EThOS.

[5] GianLuigi Ferrari and Emilio Tuosto. 2001. A Debugging Calculus for Mobile Ambients. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC '01).* ACM, New York, NY, USA, Article 1.

[6] Gian Luigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. 2008. Debugging Distributed Systems with Causal Nets. *ECEASST* 14 (2008).

[7] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16).* ACM, New York, NY, USA, 784–795.

[8] V. Jagannath, Z. Yin, and M. Budiu. 2011. Monitoring and Debugging DryadLINQ Applications with Daphne. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum.* 1266–1273.

[9] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam

Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization.. In *POPL*, John Field and Michael Hicks (Eds.). ACM, 285–296.

[10] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[11] Max Leske, Andrei Chic$_s$, and Oscar Nierstrasz. 2016. A Promising Approach for Debugging Remote Promises. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies (IWST'16)*. ACM, New York, NY, USA, Article 18, 9 pages.

[12] He Li, Jie Luo, and Wei Li. 2014. A formal semantics for debugging synchronous message passing-based concurrent programs. *Science China Information Sciences* 57, 12 (01 Dec 2014), 1–18.

[13] Stefan Marr and Hanspeter Mössenböck. 2015. Optimizing Communicating Event-Loop Languages with Truffle. (26 October 2015).

[14] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools. In *Proceedings of the 13th Symposium on Dynamic Languages (DLS'17)*. ACM, 12.

[15] Charles E McDowell and David P Helmbold. 1989. Debugging concurrent programs. *ACM Computing Surveys (CSUR)* 21, 4 (1989), 593–622.

[16] Mark S Miller, E Dean Tribble, and Jonathan Shapiro. 2005. Concurrency among strangers. In *International Symposium on Trustworthy Global Computing*. Springer, 195–229.

[17] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2016. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2016), 83–110.

[18] Semih Salihoglu, Jaeho Shin, Vikesh Khanna, Ba Quan Truong, and Jennifer Widom. 2015. Graft: A Debugging Tool For Apache Giraph. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1403–1408.

[19] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 796–807.

[20] Jan Schäfer and Arnd Poetzsch-Heffter. 2010. JCoBox: Generalizing Active Objects to Concurrent Components. In *ECOOP 2010 – Object-Oriented Programming (LNCS)*, Vol. 6183. Springer, Berlin, 275–299.

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

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

[23] Tom Van Cutsem, Stijn Mostinckx, Elisa Boix Gonzalez, Jessie Dedecker, and Wolfgang De Meuter. 2007. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)* (2007), 3–12.

[24] Roland Wismüller. 1997. Debugging Message Passing Programs Using Invisible Message Tags.. In *PVM/MPI (Lecture Notes in Computer Science)*, Marian Bubak, Jack Dongarra, and Jerzy Wasniewski (Eds.), Vol. 1332. Springer, 295–302.