Mailbox Abstractions for Static Analysis of Actor Programs*

Quentin Stiévenart[1], Jens Nicolay[2], Wolfgang De Meuter[3], and Coen De Roover[4]

1 **Software Languages Lab, Vrije Universiteit Brussel, Belgium**
`qstieven@vub.ac.be`

2 **Software Languages Lab, Vrije Universiteit Brussel, Belgium**
`jnicolay@vub.ac.be`

3 **Software Languages Lab, Vrije Universiteit Brussel, Belgium**
`wdmeuter@vub.ac.be`

4 **Software Languages Lab, Vrije Universiteit Brussel, Belgium**
`cderoove@vub.ac.be`

Abstract

Properties such as the absence of errors or bounds on mailbox sizes are hard to deduce statically for actor-based programs. This is because actor-based programs exhibit several sources of unboundedness, in addition to the non-determinism that is inherent to the concurrent execution of actors. We developed a static technique based on abstract interpretation to soundly reason in a finite amount of time about the possible executions of an actor-based program. We use our technique to statically verify the absence of errors in actor-based programs, and to compute upper bounds on the actors' mailboxes. Sound abstraction of these mailboxes is crucial to the precision of any such technique. We provide several mailbox abstractions and categorize them according to the extent to which they preserve message ordering and multiplicity of messages in a mailbox. We formally prove the soundness of each mailbox abstraction, and empirically evaluate their precision and performance trade-offs on a corpus of benchmark programs. The results show that our technique can statically verify the absence of errors for more benchmark programs than the state-of-the-art analysis.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages – Program Analysis

Keywords and phrases static analysis, abstraction, abstract interpretation, actors, mailbox

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.25

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.3.2.11`

1 Introduction

Although most actor models disallow actors from sharing state, actor-based programs are still difficult to reason about. For instance, reasoning about a message-level data race still requires computing the execution interleavings of all involved actors. Static analyses to

* Quentin Stiévenart is funded by the GRAVE project of the "Fonds voor Wetenschappelijk Onderzoek" (FWO Flanders). Jens Nicolay is funded by the the SeCloud project sponsored by Innoviris, the Brussels Institute for Research and Innovation.

reason about actor-based programs are therefore required. To terminate in finite time and space, static program analyses need to account for several sources of unboundedness [26]. This is already challenging for higher-order programs, where the data domain is unbounded and control-flow is intertwined with the flow of data [31]. Adding actors to higher-order programs complicates matters further. Most actor models do not limit the number of actors created at run-time nor the number of messages exchanged, and correct but non-terminating actor programs are common. Due to the model's inherent concurrency, there are myriads of different executions possible for a given program with a given input.

To enable defect detection and other tool support, we present a static analysis that computes a sound over-approximation of the runtime behavior of a given actor-based higher-order program. If this over-approximation does not exhibit the sought-after defect, neither does the program for any possible input and any possible actor execution interleaving (i.e., the over-approximation is sound). A defect found in the over-approximation might, however, not have any counterpart in the runtime behavior of the program (i.e., the defect is a false positive). Such false positives often stem from the use of imprecise abstractions.

Static analyses for actor-based higher-order programs are few and far between. We argue that existing analyses use mailbox abstractions that undermine their precision. Before introducing our approach (Section 1.3), we discuss two important problems of existing work that hamper their use as the foundation for proper tool support.

1.1 Problem #1: Missing interleavings for ordered-message mailbox models

Most actor models schedule actors non-deterministically for execution at any given moment. This renders reasoning about an actor program by enumerating all possible execution interleavings computationally expensive.

Actor models are said to satisfy the *isolated turn principle* [13] or to feature *macro-step semantics* [2] if actors are precluded from sharing state and feature message reception as the only blocking operation. If this is the case, it is possible to treat message processing in isolation for every message and every actor. A macro step is a sequence of small operational steps, involving a single actor, from the reception of a message until the completion of the work associated with that message. Agha et al. [2] prove that, for actor models with unordered mailboxes, any small-step interleaving has a semantically equivalent macro-step interleaving. As a result, static analyses only need to account for the interleavings of macro steps rather than the interleavings of all small steps.

Macro-step semantics has been used in prior work to reduce the number of interleavings to verify actor programs [36, 28]. The situation, however, is different for actor models in which mailboxes do preserve the ordering of their messages. Examples of such actor models include the original actor model [1], and implementations such as Erlang [3] and Akka [24].

To illustrate how the order of message sends is impacted by macro-step semantics, consider Listing 1. In this example, two actors are defined by specifying their initial behavior (lines 2 and 7). The first actor (line 2) with behavior `beh1` takes no parameters, and handles three different messages (lines 3–5). After processing a message, an actor `become`s a new behavior. The second actor (line 7) with behavior `beh2` takes one parameter, `target`, and upon receiving message `start` (line 8) sends two messages to this target (lines 9 and 10). The main process then creates actor `t` (line 12) with behavior `beh1`, and actor `a` with behavior `beh2` (line 13), specifying actor `t` as its target. The process then sends message `start` to actor `a` (line 14), followed by message `m3` to actor `t`.

Listing 1 Example program motivating the need for static analyses to revisit macro-stepping for actor models with ordered-message mailboxes.

```
(define beh1
  (actor ()
    (m1 () (become beh1))
    (m2 () (become beh1))
    (m3 () (become beh1))))
(define beh2
  (actor (target)
    (start ()
      (send target m1)
      (send target m2)
      (become beh2 target))))
(define t (create beh1))
(define a (create beh2 t))
(send a start)
(send t m3)
```

For a static analysis to be sound for an actor model in which mailboxes preserve the ordering of their messages, it should account for actor `t` to receive messages in its mailbox in any of the following orders:

- `m1`, `m2`, `m3`: actor `a` sends its messages, after which the main process is scheduled for execution,
- `m3`, `m1`, `m2`: the main process sends its message, after which it is followed by actor `a`,
- `m1`, `m3`, `m2`: actor `a` sends a first message, the main process is scheduled, after which actor `a` sends its second message.

The same analysis sped up through macro-stepping will, however, no longer include the third interleaving in its over-approximation of the program's runtime behavior. This is because the analysis will not interleave the main process with actor `a`'s processing of the `start` message. According to the analysis, actor `a` will always send both `m1` and `m2` without interruptions. Analyses sped up through macro-stepping are therefore unsound for actor models in which mailboxes preserve message ordering.

To render the analysis sound again, we therefore propose to speed it up through a finer-grained variant of macro-stepping that we call *ordered* macro-stepping. During an ordered macro step, the analysis allows each actor to receive a message and to send a single message. The ordered macro step ends right before a second message is sent, as message sends can introduce other interleavings to be considered by the analysis. Two ordered macro steps (instead of one regular macro step) are therefore required to analyze actor `a`'s processing of the `start` message. The first one ends before actor `a` sends the second message, allowing the main actor to send its message before `a` sends message `m2`. The difference with regular macro-stepping is small, but ensures that analyses account for interleavings at message *sends* as well.

This example illustrates that regular macro-stepping, while useful to speed up static analysis, needs to be adapted for actor models with ordered-message mailboxes. Otherwise, important message interleavings might be discarded rendering the analysis unsound. For unordered-message mailboxes, regular macro-stepping suffices because messages can be reordered arbitrarily in the mailbox.

Listing 2 Example actor-based stack implementation adapted from Agha [1].

```
 1 (define stack-node
 2   (actor (content link)
 3     (push (v)
 4       (become stack-node v
 5         (lambda () (become stack-node content link))))
 6     (pop (customer)
 7       (if link
 8         (begin
 9           (send customer message content)
10           (link))
11         (begin
12           (error "stack␣underflow")
13           (terminate))))))
14 (define display (create display-actor))
15 (define act (create stack-node #f #f))
16 (send act push (read-int))
17 (send act pop display)
```

1.2 Problem #2: Loss of message ordering and multiplicity

To ensure termination in a finite amount of time and space, static analyses need to abstract every potentially unbounded program component. For actor-based programs this includes the actors' mailboxes. Static analyses can avoid abstracting mailboxes if the program's actor model explicitly constrains mailbox size or if mailbox bounds can be computed for the actor program ahead-of-time. However, only 2 out of the 11 actor models surveyed in De Koster et al. [14] allow explicit bounds on mailboxes, and computing mailbox bounds for any actor-based program is undecidable in general. Mailbox abstractions have to be chosen carefully, as illustrated by the following example.

Consider Listing 2, adapted from Agha [1]. This program uses an actor with behavior `stack-node` to represent a stack. When receiving the `push` message with a value `v` to be stored on the stack (line 3), the actor creates a closure capable of restoring its current state, i.e., the values of `content` and `link`. The actor then sets `content` to the pushed value and `link` to the closure. When receiving a `pop` message (line 6), the value of `content` is sent to the provided target actor `customer`, and the `link` closure is called to restore the previous state. Should the stack be empty upon a pop (i.e., `link` is `#f`), a stack underflow error is raised (line 12). The main process pushes a value obtained from the user on a stack `act` (line 16), pops one value from this stack (line 17), which will send it (line 9) to a `display` actor (omitted from the example, passed along on line 17) that will print the value received.

Although the program in Listing 2 contains an error statement on line 12, this error is not reachable in any execution of the program under any input nor under any interleaving. Some related work, such as D'Osualdo et al. [17], abstracts mailboxes as powersets. Lines 16–17 then result in a mailbox that is abstracted as the set {`push`, `pop`}. To preserve soundness, analyses need to extract messages from this mailbox non-deterministically. This is because there is no information about the *multiplicity* of the messages in the mailbox. Analyses therefore compute not one, but two mailboxes as the result of retrieving `push` from this mailbox: {`pop`} and {`push`, `pop`}. Retrieving the next message from the mailbox {`pop`} again yields two mailboxes: $\varnothing$ and {`pop`}. Through the former case, the analysis accounts for `pop` being present but once and deems the stack underflow error unreachable as a result. Through the latter case, the analysis accounts for pop being present more than once. It now deems the stack underflow error reachable as the stack may be empty when a subsequent pop is processed. This false positive results from a loss of precision due to the use of a powerset abstraction for the actor's mailbox.

Other related work, such as Agha et al. [2] and Garoche et al. [22], relies on a multiset definition of mailboxes. Multisets are sets that preserve multiplicity but, like powersets, are unordered. However, a mailbox abstraction that preserves multiplicity does not suffice either to analyze this program precisely. At the point where the stack actor has received the push message followed by the pop message, the analysis has computed its mailbox abstraction to the multiset $[\texttt{push} \mapsto 1, \texttt{pop} \mapsto 1]$. This multiset encodes the information that a both a push message and a pop messages are present once in the mailbox. Again, the analysis needs to extract the next message to process non-deterministically, giving rise to two possible successor mailboxes $[\texttt{pop} \mapsto 1]$ and $[\texttt{push} \mapsto 1]$. The former multiset represents the mailbox of the stack actor after it has processed message push. In contrast to the set abstraction, retrieving the next message from this mailbox gives rise to a single mailbox $[]$, because pop is present only once, and no stack underflow error can be reached through (spurious) subsequent pop messages. However, because ordering information is not preserved, pop might be processed before its corresponding push, the analysis still deems the stack underflow error reachable under a multiset abstraction for the actor's mailbox.

This example motivates the importance of mailbox abstractions that satisfy ordering *and* multiplicity: without one or the other, the analysis cannot automatically prove the program in Listing 2 free of errors.

1.3 Our approach

We argue that precise analysis of actor-based programs requires a proper mailbox abstraction. For actor models with ordered-message mailboxes (e.g., [1, 25, 24, 3]), this abstraction needs to preserve ordering and multiplicity of its messages (Section 1.2). In addition, those actor models require the analysis to interleave message sending using ordered macro-stepping for it to be sound (Section 1.1). For the others (e.g., [2, 22]), ordered macro-stepping is still sound but regular macro-stepping suffices. We therefore do not present one analysis, but a framework capable of analyzing programs from different actor models that features ordered macro-stepping and takes a mailbox abstraction as parameter.

Our framework approaches the problem of statically analyzing actor-based programs through abstract interpretation [10]. We start by defining a simple actor language, λ_α, which is an extension of the λ-calculus (Section 2). We express the concrete semantics for λ_α as an abstract machine (Section 3). The result of executing an input program under these semantics is a flow graph that represents the program's runtime behavior and enables verifying behavioral properties. In this work we focus on verifying the absence of runtime errors and mailbox bounds. Because the computed flow graph can be infinite under concrete semantics, we apply a systematic abstraction, resulting in an abstract semantics for λ_α (Section 4). We leave the mailbox abstraction as a parameter of the abstract semantics, and present multiple instantiations of mailbox abstractions together with their properties, categorized into four categories (Section 5). We evaluate each of these mailbox abstractions on a set of benchmark programs with respect to performance and precision (Section 6), and compare our results with those obtained by Soter, a state-of-the-art tool for analyzing Erlang programs [17]. We conclude with a discussion of related work and the limitations of our approach (Section 7).

$$
\begin{array}{rcl}
e \in \mathit{Exp} & ::= & \mathit{ae} \mid (\mathit{ae}\ \mathit{ae}^*) \\
& \mid & (\texttt{letrec}\ ((x\ e)^*)\ e) \\
& \mid & (\texttt{error}) \\
& \mid & (\texttt{create}\ \mathit{ae}\ \mathit{ae}^*) \\
& \mid & (\texttt{send}\ \mathit{ae}\ t\ \mathit{ae}^*) \\
& \mid & (\texttt{become}\ \mathit{ae}\ \mathit{ae}^*) \\
& \mid & (\texttt{terminate})
\end{array}
\qquad
\begin{array}{rcl}
\mathit{ae} \in \mathit{AExp} & ::= & x \mid \mathit{lam} \mid \mathit{act} \\
\mathit{lam} \in \mathit{Lam} & ::= & (\lambda\ (x^*)\ e) \\
\mathit{act} \in \mathit{Act} & ::= & (\texttt{actor}\ (x^*) \\
& & \quad (t\ (y^*)\ e)^*) \\
x, y \in \mathit{Var} & & \text{a finite set of variable names} \\
t \in \mathit{Tag} & & \text{a finite set of tags}
\end{array}
$$

Figure 1 Grammar of the minimalistic higher-order λ_α language supporting concurrent actors.

Our work makes the following contributions:

- We present the concrete and an abstracted formal semantics of an actor-based higher-order programming language. The abstracted semantics computes a sound over-approximation of a given program's runtime behavior. To reduce non-determinism and hence speed up computation, the abstracted semantics is the first to incorporate a finer-grained variant of macro-stepping, called *ordered* macro-stepping. We show that regular macro-stepping is not sound when analyzing actor programs from ordered-message mailbox models.
- We leave the abstraction for the actors' mailboxes as a parameter to the abstracted semantics. We categorize possible mailbox abstractions according to the extent to which they preserve message ordering, and to the extent to which they preserve message multiplicity. We formally prove the soundness of each mailbox abstraction, and empirically evaluate their impact on the precision and running time of the analysis on a corpus of benchmark programs.
- We demonstrate how to use the sound over-approximation computed by our analysis to formally verify mailbox bounds and the absence of runtime errors. An evaluation shows that our technique is more precise than a state-of-the-art tool. The higher precision of our mailbox abstractions enables verifying these properties on 12 benchmark programs, of which 6 cannot be verified by the tool we compare with.

2 A Simple Actor Language: λ_α

Figure 1 defines the syntax of a minimalistic higher-order programming language based on the λ-calculus in A-Normal Form [19]. It supports actors through the following constructs:

- `actor` defines an actor behavior, associating each type of the messages the behavior can receive with a corresponding message processing body,
- `create` spawns a new actor from a given behavior and returns its process identifier,
- `send` sends a message to a specific actor identified by its process identifier,
- `become` changes the behavior of the current actor, and
- `terminate` ends the execution of the current actor.

Note that messages exchanged between actors consist of a *tag* t (a simple name) and an arbitrary number of arguments. Because tags are syntactic elements, like variable names, they are finite within a program. To facilitate benchmarking, the implementation used in our evaluation (Section 6.1) extends this language with additional features such as support for `if`-expressions. We refer to Listing 1 and 2 from the introduction for example programs in this language.

We assume the following about the concrete semantics of λ_α programs.

1. Mailboxes work in a FIFO fashion: received messages are sent to the back of the actor's mailbox, and the actor can only process the message at the front of its mailbox. Although the most widely used actor models differ here, the majority of models uses FIFO mailboxes: 6 of the 11 models reviewed in De Koster et al. [14] have FIFO mailboxes.
2. Messages are received in the same order as sent, and no message is lost during transmission. Modeling a real-world situation with messages being possibly lost or reordered would increase the complexity of the model without adding to the discussion.
3. No side effects can occur within the body of an actor. This is enforced by the language, as it does not include assignment constructs (e.g., `set!`). Many actor languages are free of side effects by definition, or contain only limited side effects. Such side effects lead to possible data races and are better avoided [7].

3 Concrete Semantics of λ_α as an Abstract Machine

3.1 State Space

We define the concrete semantics of λ_α as an abstract machine in Figure 2. This enables its abstraction using a systematic approach [38]. Each state's mapping of process identifiers to evaluation contexts is testament to its concurrency support. A process' evaluation context *ctx* can be waiting for a message (**wait**), can be stuck due to a programmer error (**error**), or can be processing a message (**ev** when an expression is evaluated in a given environment, and **ko** when a value is reached). It is always linked to a current actor behavior a, with the special case that the initial process is linked to the **main** behavior. Other actors have an instantiated behavior (**acti**), consisting of an actor expression and an extended environment. Its final component is a mailbox represented as a sequence of messages, where each message is composed of a tag (see Figure 1) and a list of values. The only values in λ_α are regular closures (**clo**) which combine a lambda expression with a definition environment, actor closures (**actd**) which combine an actor definition with a definition environment, and process identifiers (**pid**).

We use a *value store* σ to store values produced by the program. The machine's continuations κ are threaded through a separate *continuation store* Ξ. Separating the addresses at which values and continuations are allocated will render the abstract semantics more precise. Both stores are shared by all processes. This not to model shared-memory concurrency, but to enable an important optimization called *global store widening* [38], discussed in Section 6.1. Process identifiers, value addresses and continuation addresses are parameters of the semantics. We give instantiations of these parameters in Section 3.3.

3.2 Atomic Expressions

Atomic expressions *AExp* are expressions that the machine reduces to a value in a single step without having to allocate addresses or having to modify the store. They are evaluated through $\mathcal{A} : \mathit{AExp} \times \mathit{Env} \times \mathit{Store} \to \mathit{Val}$. Its definition is as usual, with the addition that actor definitions are wrapped with their definition environment, similarly to closures.

$$\mathcal{A}(x, \rho, \sigma) = \sigma(\rho(x)) \qquad \mathcal{A}(\mathit{lam}, \rho, \_) = \mathbf{clo}(\mathit{lam}, \rho) \qquad \mathcal{A}(\mathit{act}, \rho, \_) = \mathbf{actd}(\mathit{act}, \rho)$$

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathit{Procs} \times \mathit{Store} \times \mathit{KStore} & a \in \mathit{Actor} &::= \mathbf{acti}(\mathit{act}, \rho) \\
\pi \in \mathit{Procs} &= \mathit{Pid} \rightharpoonup \mathit{Context} & &\mid\ \mathbf{main} \\
\mathit{ctx} \in \mathit{Context} &= (\mathit{Control} \times \mathit{Kont} & \phi \in \mathit{Frame} &::= \mathbf{letk}(a, e, \rho) \\
&\quad \times \mathit{Actor} \times \mathit{Mbox}) & \kappa \in \mathit{Kont} &= \mathit{Frame} \times \mathit{KAddr} + \{\epsilon\} \\
c \in \mathit{Control} &::= \mathbf{ev}(e, \rho) \mid \mathbf{ko}(v) & \rho \in \mathit{Env} &= \mathit{Var} \rightharpoonup \mathit{Addr} \\
&\mid\ \mathbf{wait} \mid \mathbf{error} & \sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Val} \\
v \in \mathit{Val} &::= \mathbf{clo}(\mathit{lam}, \rho) & \Xi \in \mathit{KStore} &= \mathit{KAddr} \rightharpoonup \mathit{Kont} \\
&\mid\ \mathbf{actd}(\mathit{act}, \rho) & \mathit{mb} \in \mathit{Mbox} &= \mathit{Message}^* \\
&\mid\ \mathbf{pid}(p) & \mathit{addr} \in \mathit{Addr}&,\ \mathit{kaddr} \in \mathit{KAddr} \\
m \in \mathit{Message} &= \mathit{Tag} \times \mathit{Val}^* & p &\in \mathit{Pid}
\end{aligned}
$$

Figure 2 State space of the concrete abstract machine for λ_α.

3.3 Addresses, Process Identifiers and Allocation

Value addresses, continuation addresses and process identifiers are parameters of the semantics. They are produced by the allocation functions $\mathit{alloc} : \mathit{Var} \times \Sigma \to \mathit{Addr}$, $\mathit{kalloc} : \mathit{Exp} \times \Sigma \to \mathit{KAddr}$ and $\mathit{palloc} : \mathit{Exp} \times \Sigma \to \mathit{Pid}$ respectively. For λ_α's concrete abstract machine, an example instantiation is as follows.

$$
\begin{aligned}
\mathit{Addr} &= \mathit{Var} \times \mathbb{N} & \mathit{alloc}(x, \langle \_, \sigma, \_ \rangle) &= (x, |\operatorname{Dom}(\sigma)| + 1) \\
\mathit{KAddr} &= \mathbb{N} & \mathit{kalloc}(e, \langle \_, \_, \Xi \rangle) &= |\operatorname{Dom}(\Xi)| + 1 \\
\mathit{Pid} &= \mathbb{N} & \mathit{palloc}(e, \langle \pi, \_, \_ \rangle) &= |\operatorname{Dom}(\pi)| + 1
\end{aligned}
$$

3.4 Concrete Mailboxes

The following parameters to the abstract machine complete Figure 2's definition of mailboxes.

- $\mathit{empty} \in \mathit{Mbox}$ is a special element representing the empty mailbox.
- $\mathit{enq} : (\mathit{Message} \times \mathit{Mbox}) \to \mathit{Mbox}$ enqueues a message at the back of a mailbox.
- $\mathit{deq} : \mathit{Mbox} \to \mathcal{P}(\mathit{Message} \times \mathit{Mbox})$ dequeues a message from the front of the mailbox, resulting in the message and the new mailbox. Using a powerset as range will facilitate incorporating non-determinism in the abstract semantics. The result of dequeuing from the empty mailbox is the empty set.
- $\mathit{size} : \mathit{Mbox} \to \mathbb{N}$ computes the size of a mailbox.

The concrete representation of a mailbox is a sequence of messages, with the following definitions (where :: both denotes prepending a sequence with an element, and appending an element at the end of a sequence).

$$
\begin{aligned}
\mathit{empty} &= \epsilon & \mathit{deq}(\epsilon) &= \{\} & \mathit{size}(\epsilon) &= 0 \\
\mathit{enq}(m, \mathit{mb}) &= \mathit{mb} :: m & \mathit{deq}(m :: \mathit{mb}) &= \{(m, \mathit{mb})\} & \mathit{size}(m :: \mathit{mb}) &= \mathit{size}(\mathit{mb}) + 1
\end{aligned}
$$

3.5 Transition Relation

The small-step transition relation $(\mapsto) : \mathit{Pid} \times \mathit{Effect} \times \Sigma \times \Sigma$ defines the small-step semantics of the λ_α language, in Figure 3. We write $\varsigma \overset{p}{\underset{E}{\longmapsto}} \varsigma'$ as a shorthand for $(p, E, \varsigma, \varsigma') \in (\mapsto)$, meaning that from state ς, a small step on actor p can be performed to reach state ς', and this generates effect E. This transition relation is therefore annotated with the process identifier

p of the actor that performs a transition, and with an *effect* E used by the macro-step transition relation.

The possible effects correspond to the actions that actors can perform: creating a new actor (*Create*), sending a message (*Send*), receiving a message (*Receive*), changing the actor's behavior (*Become*), or terminating the actor (*Terminate*). The *NoEffect* effect denotes the absence of effect on a transition. We shorten $\varsigma \xmapsto[NoEffect]{p} \varsigma'$ to $\varsigma \overset{p}{\mapsto} \varsigma'$.

$$E \in \mathit{Effect} ::= \mathit{Create} \;\mid\; \mathit{Send} \;\mid\; \mathit{Receive} \;\mid\; \mathit{Become} \;\mid\; \mathit{Terminate} \;\mid\; \mathit{NoEffect}$$

Rules for transitions that do not affect other actors or the current actor's behavior or mailbox are called *sequential rules*. We only formalize the sequential rule for the `error` statement as an example. Other sequential rules follow the same structure. The non-sequential rules are related to how actors interact with each other and their mailbox.

- T-Err: evaluating an `error` statement yields an error state.
- T-Create: `create` spawns a new actor with the given behavior (**actd**), where constructor parameters are bound to the given arguments to create an actor instantiation (**acti**). The newly created actor starts in a **wait** status, and with an empty continuation and mailbox.
- T-Become: `become` changes an actor's behavior by updating its current behavior and binding its constructor parameters to the given arguments. The return value of `become` is the new behavior.
- T-Receive: when an actor is **wait**ing, it can dequeue a message from the front of its mailbox and process it, by evaluating the corresponding message processing body of its current behavior in an extended environment.
- T-Wait: an actor with an empty continuation has computed a value and has therefore completed the processing of a message. It then goes back to **wait**ing for new messages.
- T-Send and T-Send-Self: when an actor sends a message, two different rules may apply: one for an actor sending a message to a different actor, the other for an actor sending a message to itself. To send a message from a *sender* actor to a different *receiver* actor (T-Send), the receiver actor and the message arguments have to be evaluated. The message is then enqueued on the receiver actor's mailbox. Self-sends are handled by a different rule (T-Send-Self) to avoid incorrect updates to the process map.
- T-Terminate `terminate` removes the actor from the process map. Here, $\pi - p$ denotes the removal of the element of which the key is p.

3.6 Macro-Stepping Semantics

As motivated in Section 1.1, we speed up our analysis through a variant of regular macro-stepping [2] that we call *ordered macro-stepping*. We now formalize a general macro-stepping semantics from which either can be instantiated.

The transition relation ($\xmapsto[E]{p}$) performs a small step in the evaluation of a program. A *macro step* is a sequence of small steps of which the first can produce any effect, and the remaining steps are constrained to a restricted set of effects. The particular restriction determines whether the macro step is ordered. We first define a restricted multi-stepping transition relation ($\longmapsto^{*\downarrow}$) $\subseteq (\mathit{Pid} \times \mathcal{P}(\mathit{Effect}) \times \mathcal{P}(\mathit{Effect}) \times \Sigma \times \Sigma)$. It performs multiple small steps of the transition relation on a single actor until it reaches a transition producing an effect that is disallowed. This multi-stepping transition relation is defined in Figure 4, where set X denotes effects that are never allowed and function $f : \mathit{Effect} \to \mathcal{P}(\mathit{Effect})$ defines which effects are no longer allowed once a given effect has been produced.

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(error)}, \rho), \kappa, a, mb\rangle}{\langle \pi, \sigma, \Xi\rangle \overset{p}{\mapsto} \langle \pi[p \mapsto \langle \mathbf{error}, \kappa, a, mb\rangle], \sigma, \Xi\rangle}\ \textsc{T-Error}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(create}\ \ æ_a\ æ_1 \dots æ_n\texttt{)}, \rho), \kappa, a, mb\rangle \\ p' = palloc(æ_a, \langle \pi, \sigma, \Xi\rangle) \qquad \mathbf{actd}(act, \rho_a) = \mathcal{A}(æ_a, \rho, \sigma) \\ \texttt{(actor (}x_1 \dots x_n\texttt{)} \dots\texttt{)} = act \qquad addr_i = alloc(x_i, \langle \pi, \sigma, \Xi\rangle) \\ v_i = \mathcal{A}(æ_i, \rho, \sigma) \qquad \rho'_a = \rho_a[x_i \mapsto addr_i] \qquad a' = \mathbf{acti}(act, \rho'_a)\end{array}}{\begin{array}{l}\langle \pi, \sigma, \Xi\rangle \xmapsto[Create]{p} \langle \pi[p \mapsto \langle \mathbf{ko}(\mathbf{pid}(p')), \kappa, a, mb\rangle, \\ \qquad\qquad\qquad p' \mapsto \langle \mathbf{wait}, \epsilon, a', empty\rangle], \\ \qquad\qquad\quad \sigma[addr_i \mapsto v_i], \Xi\rangle\end{array}}\ \textsc{T-Create}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(become}\ \ æ_a\ æ_1 \dots æ_n\texttt{)}, \rho), \kappa, a, mb\rangle \\ \mathbf{actd}(act, \rho_a) = \mathcal{A}(æ_a, \rho, \sigma) \\ \texttt{(actor (}x_1 \dots x_n\texttt{)} \dots\texttt{)} = act \qquad addr_i = alloc(x_i, \langle \pi, \sigma, \Xi\rangle) \\ v_i = \mathcal{A}(æ_i, \rho, \sigma) \qquad \rho'_a = \rho_a[x_i \mapsto addr_i] \qquad a' = \mathbf{acti}(act, \rho'_a)\end{array}}{\langle \pi, \sigma, \Xi\rangle \xmapsto[Become]{p} \langle \pi[p \mapsto \langle \mathbf{ko}(\mathbf{actd}(act, \rho_a)), \kappa, a', mb\rangle], \sigma[addr_i \mapsto v_i], \Xi\rangle}\ \textsc{T-Become}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{wait}, \epsilon, a, mb\rangle \qquad ((t, v_1 \dots v_n), mb') \in deq(mb) \\ \mathbf{acti}(\texttt{(actor (}x_1 \dots x_n\texttt{)} \dots \texttt{(}t\ \texttt{(}y_1 \dots y_n\texttt{)}\ e\texttt{)} \dots\texttt{)}, \rho_b) = a \\ addr_i = alloc(y_i, \langle \pi, \sigma, \Xi\rangle) \qquad \rho'_b = \rho_b[y_i \mapsto addr_i]\end{array}}{\begin{array}{l}\langle \pi, \sigma, \Xi\rangle \xmapsto[Receive]{p} \langle \pi[p \mapsto \langle \mathbf{ev}(e, \rho'_b), \epsilon, a, mb'\rangle], \\ \qquad\qquad\qquad \sigma[addr_i \mapsto v_i], \Xi\rangle\end{array}}\ \textsc{T-Receive}$$

$$\frac{\pi(p) = \langle \mathbf{ko}(v), \epsilon, a, mb\rangle}{\langle \pi, \sigma, \Xi\rangle \overset{p}{\mapsto} \langle \pi[p \mapsto \langle \mathbf{wait}, \epsilon, a, mb\rangle], \sigma, \Xi\rangle}\ \textsc{T-Wait}$$

$$\frac{\begin{array}{c}\pi(p_s) = \langle \mathbf{ev}(\texttt{(send}\ \ æ_0\ t\ æ_1 \dots ae_n\texttt{)}, \rho), \kappa_s, a_s, mb_s\rangle \qquad \mathbf{pid}(p_r) = \mathcal{A}(æ_0, \rho, \sigma) \\ \pi(p_r) = \langle c, \kappa_r, a_r, mb_r\rangle \qquad p_r \neq p_s \qquad v_i = \mathcal{A}(æ_i, \rho, \sigma) \qquad m = (t, v_1 \dots v_n)\end{array}}{\begin{array}{l}\langle \pi, \sigma, \Xi\rangle \xmapsto[Send]{p_s} \langle \pi[p_s \mapsto \langle \mathbf{ko}(\mathbf{pid}(p_r)), \kappa_s, a_s, mb_s\rangle, \\ \qquad\qquad\qquad p_r \mapsto \langle c, \kappa_r, a_r, enq(m, mb_r)\rangle], \\ \qquad\qquad\quad \sigma, \Xi\rangle\end{array}}\ \textsc{T-Send}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(send}\ \ æ_0\ t\ æ_1 \dots ae_n\texttt{)}, \rho), \kappa, a, mb\rangle \\ \mathbf{pid}(p) = \mathcal{A}(æ_0, \rho, \sigma) \qquad v_i = \mathcal{A}(æ_i, \rho, \sigma) \qquad m = (t, v_1 \dots v_n)\end{array}}{\langle \pi, \sigma, \Xi\rangle \xmapsto[Send]{p} \langle \pi[p \mapsto \langle \mathbf{ko}(\mathbf{pid}(p)), \kappa, a, enq(m, mb)\rangle], \sigma, \Xi\rangle}\ \textsc{T-Send-Self}$$

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(terminate)}, \rho), \_, \_, \_\rangle}{\langle \pi, \sigma, \Xi\rangle \xmapsto[Terminate]{p} \langle \pi - p, \sigma, \Xi\rangle}\ \textsc{T-Terminate}$$

Figure 3 Concrete transition relation for λ_α programs.

$$\frac{\varsigma_1 \xmapsto[E]{p} \varsigma_2 \qquad \varsigma_2 \xmapsto[Es]{p}{}^{*\downarrow}_{X \cup f(E)} \varsigma_N \qquad E \notin X}{\varsigma_1 \xmapsto[Es \cup \{E\}]{p}{}^{*\downarrow}_{X} \varsigma_N}\ \textsc{M-Main}$$

$$\frac{\varsigma_1 \xmapsto[E_1]{p} \varsigma_2 \qquad \nexists \varsigma_3, \varsigma_2 \xmapsto[E_2]{p} \varsigma_3 \qquad E_1 \notin X}{\varsigma_1 \xmapsto[E_1]{p}{}^{*\downarrow}_{X} \varsigma_2}\ \textsc{M-Blocked}$$

$$\frac{\varsigma_1 \xmapsto[E_1]{p} \varsigma_2 \qquad \varsigma_2 \xmapsto[E_2]{p} \varsigma_3 \qquad E_1 \notin X \qquad E_2 \in X}{\varsigma_1 \xmapsto[\{E_1\}]{p}{}^{*\downarrow}_{X} \varsigma_2}\ \textsc{M-Stop}$$

$$\varsigma_1 \xmapsto[Es \cup \{E\}]{p}{}^{M} \varsigma_N \iff \varsigma_1 \xmapsto[E]{p} \varsigma_2 \wedge \varsigma_2 \xmapsto[Es]{p}{}^{*\downarrow}_{f(E)} \varsigma_N$$

Figure 4 Concrete macro-stepping transition relation.

- M-Main: a small step producing an allowed effect can be performed, followed by a restricted multi-step with the set of disallowed effects augmented by the result of f on the produced effect.
- M-Stop: only a single small step can be performed, because the next small step would produce an effect that is disallowed.
- M-Blocked: only a single small step can be performed, because no further small steps can be performed from the resulting state on the same process (i.e., the process is blocked).

The macro-stepping transition relation $(\longmapsto^{M}) \subseteq (\mathit{Pid} \times \mathcal{P}(\mathit{Effect}) \times \Sigma \times \Sigma)$, also defined in Figure 4, first makes a single unrestricted small step followed by a restricted multi-step. Using $f(E) = \{\mathit{Receive}\}$ gives rise to the unordered macro-stepping semantics of Agha et al. [2]. Its restriction disallows receiving messages after the first small step of a macro step. Our ordered macro-stepping semantics follows from $f(\mathit{Send}) = \{\mathit{Receive}, \mathit{Send}\}$, and $f(E) = \{\mathit{Receive}\}$ otherwise. This restriction disallows actors from sending more than one message. Sending more results in another macro-step. As in the unordered macro-stepping semantics, message can only be received during the first small step of an ordered macro step.

3.7 Collecting Macro-Stepping Semantics

The collecting semantics of a λ_α program e under macro-stepping can be computed as the fixpoint of the function $\mathcal{F}_e : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. The collecting semantics $\mathrm{lfp}(\mathcal{F}_e)$ is a set containing every reachable state in the evaluation of program e under any possible interleaving. The granularity of interleavings is defined by the macro-stepping semantics, in particular by the restricting function f. As explained in Section 1.1, the use of macro-stepping semantics instead of interleaving semantics has the benefit of reducing the number of interleavings to consider when analyzing a program. Evaluation starts at an initial state given by the injection function $\mathcal{I} : \mathit{Exp} \to \Sigma$.

$$\mathcal{F}_e(S) = \{\mathcal{I}(e)\} \cup \left\{ \varsigma' \mid \underbrace{\langle \pi, \_, \_ \rangle}_{\varsigma} \in S \wedge p \in \mathrm{Dom}(\pi) \wedge \varsigma \xmapsto[Es]{p}{}^{M} \varsigma' \right\}$$

$$\mathcal{I}(e) = \langle [\mathbf{main} \mapsto \langle \mathbf{ev}(e, []), \mathit{empty}, \mathbf{main} \rangle], [], [] \rangle$$

3.8 Program Properties

Useful properties of actor-based programs can be inferred from the collecting semantics. We demonstrate this for reachability of error states and for bounds on actor mailboxes. Examples of other properties include the possible values of a variable, the messages and message arguments that an actor can receive during its lifetime, or the behaviors that an actor actually assumes. Because reachability within the collecting semantics is not decidable, we resort to abstraction in order to automatically verify these properties (Section 4.7).

Reachability of error states. Predicate $\mathit{ErrorReachable}_e$ holds when an error is reachable in program e.

$$\mathit{ErrorReachable}_e \iff \exists \langle \pi, \_, \_ \rangle \in \mathrm{lfp}(\mathcal{F}_e), p \in \mathrm{Dom}(\pi) \mid \pi(p) = \langle \mathbf{error}, \_, \_, \_ \rangle$$

Mailbox bounds. Function $\mathit{MailboxBound}_e(p)$ computes the maximal number of messages an actor with process identifier p can have in its mailbox when executing program e.

$$\mathit{MailboxBound}_e(p) = \mathit{max}\left(\{\mathit{size}(mb) \mid \langle \pi, \_, \_ \rangle \in \mathrm{lfp}(\mathcal{F}_e) \wedge \pi(p) = \langle \_, \_, \_, mb \rangle\}\right)$$

4 Abstract Interpretation of λ_α

The semantics of λ_α can be abstracted systematically in a sound manner using the abstracting abstract machines approach of Van Horn and Might [38], through the abstraction function α given in the accompanying technical report[1].

4.1 Abstract State Space

The state space resulting from systematic abstraction is given in Figure 5. Abstract components that are the counterpart of a concrete component are denoted by a hat ($\hat{X}$). The abstraction of addresses and process identifiers is a parameter of the analysis. We also leave the abstraction of the mailbox a parameter of the analysis, of which we discuss possible instantiations in Section 5. Systematic abstraction has made process map, value store and continuation store to map elements of their domain to *sets* of contexts, values and continuations. This change in ranges stems from the abstract semantics having to compute a sound over-approximation with but a finite amount of addresses and process identifiers. Messages are now composed of a tag and a sequence of *sets* of abstract values. Section 4.4 motivates this change by reduced non-determinism.

4.2 Abstract Atomic Expressions

Abstract evaluation of atomic expressions might yield more than one abstract value, as the value store now maps addresses to *sets of abstract values* because a single abstract address can correspond to multiple concrete ones. We therefore obtain the following definition of $\hat{\mathcal{A}} : \mathit{AExp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \to \mathcal{P}(\mathit{Val})$

$$\hat{\mathcal{A}}(x, \rho, \sigma) = \sigma(\rho(x)) \qquad \mathcal{A}(lam, \rho, \_) = \{\mathbf{clo}(lam, \hat{\rho})\} \qquad \mathcal{A}(act, \rho, \_) = \{\mathbf{actd}(act, \hat{\rho})\}$$

[1] https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \widehat{Procs} \times \widehat{Store} \times \widehat{KStore} \\
\hat{\pi} \in \widehat{Procs} &= \widehat{Pid} \rightharpoonup \mathcal{P}(\widehat{Context}) \\
\widehat{ctx} \in \widehat{Context} &= (\widehat{Control} \times \widehat{Kont} \\
&\quad \times \widehat{Actor} \times \widehat{Mbox}) \\
\hat{c} \in \widehat{Control} &::= \mathbf{ev}(e, \hat{\rho}) \mid \mathbf{ko}(\hat{v}) \\
&\quad \mid \mathbf{wait} \mid \mathbf{error} \\
\hat{v} \in \widehat{Val} &::= \mathbf{clo}(lam, \hat{\rho}) \\
&\quad \mid \mathbf{actd}(act, \hat{\rho}) \\
&\quad \mid \mathbf{pid}(\hat{p}) \\
\hat{m} \in \widehat{Message} &= Tag \times \mathcal{P}(\widehat{Val})^*
\end{aligned}
\qquad
\begin{aligned}
\hat{a} \in \widehat{Actor} &::= \mathbf{acti}(act, \hat{\rho}) \mid \mathbf{main} \\
\hat{\phi} \in \widehat{Frame} &::= \mathbf{letk}(\widehat{addr}, e, \hat{\rho}) \\
\hat{\kappa} \in \widehat{Kont} &= \widehat{Frame} \times \widehat{KAddr} + \{\epsilon\} \\
\hat{\rho} \in \widehat{Env} &= Var \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Val}) \\
\hat{\Xi} \in \widehat{KStore} &= \widehat{KAddr} \rightharpoonup \mathcal{P}(\widehat{Kont}) \\
\widehat{mb} &\in \widehat{Mbox} \\
\widehat{addr} \in \widehat{Addr}&, \widehat{kaddr} \in \widehat{KAddr} \\
\hat{p} &\in \widehat{Pid}
\end{aligned}
$$

Figure 5 State space of the abstracted abstract machine for λ_α.

4.3 Abstract Addresses, Process Identifiers and Allocation

Functions $\widehat{alloc} : Var \times \hat{\Sigma} \to \widehat{Addr}$, $\widehat{kalloc} : Exp \times \hat{\Sigma} \to \widehat{KAddr}$, and $\widehat{palloc} : Exp \times \hat{\Sigma} \to \widehat{Pid}$ determine the allocation of value addresses, continuation addresses and process identifiers respectively. The instantiation of these parameters to the analysis influences precision, but not soundness, as the AAM technique has been proven sound under *any allocation strategy* [32, 23]. The following instantiation results in a flow-sensitive, context-insensitive 0-CFA analysis.

$$
\begin{aligned}
\widehat{Addr} &= Var & \widehat{alloc}(x, \hat{\varsigma}) &= x \\
\widehat{KAddr} &= Exp & \widehat{kalloc}(e, \hat{\varsigma}) &= e \\
\widehat{Pid} &= Exp & \widehat{palloc}(e, \hat{\varsigma}) &= e
\end{aligned}
$$

4.4 Abstract Transition Relation

The abstract transition rules, depicted in Figure 6, act on components of the abstract state space. We highlight the differences with the concrete rules, which arise due to sound over-approximation.

- The process map $\hat{\pi}$ now maps each process identifier to a *set* of processes. Hence the premise $\pi(p) = \ldots$ becomes $\hat{\pi}(\hat{p}) \ni \ldots$, at the cost of non-determinism when one abstract process identifier is mapped to more than one abstract process.
- For the same reason, and because the store now maps each abstract address to a *set* of values, process map updates and store updates become join operations: $\pi[p \mapsto \ldots]$ becomes $\hat{\pi} \sqcup [\hat{p} \mapsto \{\ldots\}]$. Introducing *abstract counting* [33, 34] enables to perform strong updates on the store and process map when an abstract address or an abstract process identifier is mapped to a single element.
- In rules AbsT-Create, AbsT-Become, AbsT-Send and AbsT-Send-Self, the concrete $v_i = \mathcal{A}(\ldots)$ become $\hat{V}_i = \hat{\mathcal{A}}(\ldots)$, where $\hat{V}_i \in \mathcal{P}(\widehat{Val})$, instead of $\hat{v}_i \in \hat{\mathcal{A}}(\ldots)$. This is because the result of the atomic evaluation will eventually be added to the store, which now maps to sets of values. Not having to fire rules for individual set elements, non-determinism is reduced.
- For the same reason, we directly store *sets* of values in messages in rules AbsT-Send and AbsT-Send-Self.

- In the rule AbsT-Terminate, it is not sound to remove the context of the terminating actor from the process map. This is because an abstract actor may correspond to more than one concrete actor, in which case only one of the concrete actors would terminate. Removing the abstract actor would in effect terminate all the concrete actors it corresponds to. This is problematic in terms of precision, but is remedied by our introduction of abstract counting [33] on the process map.
- The condition $p_r \neq p_s$ disappears from the rule AbsT-Send. Due to abstraction, a single abstract process identifier may correspond to more than one concrete process identifier. When a message is sent from a process with identifier $\hat{p}$, then either the target has a different process identifier and only AbsT-Send applies; or the target has the same process identifier. In the second case, the message may be sent to the same process or a different process, and both AbsT-Send and AbsT-Send-Self may apply. Requiring $\hat{p}_r \neq \hat{p}_s$ would incorrectly ignore the case in which an actor sends a message to a different one with the same abstract process identifier. With abstract counting, the condition can be restored when both $\hat{p}_r$ and $\hat{p}_s$ each correspond to a single process identifier.

4.5 Abstract Macro-Stepping Semantics

The formalization of macro-stepping for the abstract semantics remains the same: a single abstract small step is performed, followed by a number of effect-restricted abstract small steps. We obtain an abstract macro-stepping transition relation $(\widehat{\longmapsto}^{M}) \subseteq (\widehat{Pid} \times \mathcal{P}(\mathit{Effect}) \times \hat{\Sigma} \times \hat{\Sigma})$. Its soundness follows from the soundness of the abstract small-step transition relation, and is proven in Section 6.6.

4.6 Abstract Collecting Macro-Step Semantics

The abstract collecting semantics of a λ_α program e is the fixpoint of $\hat{\mathcal{F}}_e : \mathcal{P}(\hat{\Sigma}) \to \mathcal{P}(\hat{\Sigma})$.

$$\hat{\mathcal{F}}_e(\hat{S}) = \left\{\hat{\mathcal{I}}(e)\right\} \cup \left\{ \hat{\varsigma}' \mid \underbrace{\langle \hat{\pi}, \_, \_\rangle}_{\hat{\varsigma}} \in \hat{S} \wedge \hat{p} \in \mathrm{Dom}(\hat{\pi}) \wedge \hat{\varsigma} \,\widehat{\xrightarrow[Es]{\hat{p}}}^{M}\, \hat{\varsigma}' \right\}$$

$$\hat{\mathcal{I}}(e) = \langle [\mathbf{main} \mapsto \left\{\langle \mathbf{ev}(e, []), \widehat{\mathit{empty}}, \mathbf{main}\rangle\right\}], [], []\rangle$$

The abstract collecting semantics $\mathrm{lfp}(\hat{\mathcal{F}}_e)$ is therefore a set of abstract states that over-approximates the set of states reachable in all concrete execution of program e. If the abstractions used yield a finite state space, reachability within the abstract collecting semantics becomes decidable. This is the case if the number of addresses, process identifiers and mailboxes are bounded. The 0-CFA formulation of addresses and process identifiers described in Section 4.3 is bounded, as well as the bounded mailbox abstractions described in Section 5.

4.7 Abstract Program Properties

Our analysis computes a sound over-approximation of the program's behavior. More precisely, its abstract collecting semantics is a set of abstract program states that *at least* represent every reachable concrete program state. However, the computed set may also contain *spurious* abstract states that correspond to concrete program states that are *not* found in the concrete collecting semantics. This impacts the abstract program properties in several ways.

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(error)}, \hat{\rho}), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle}{\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \widehat{\xmapsto{\hat{p}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{error}, \hat{\kappa}, \hat{a}, \widehat{mb}\rangle\right\}], \hat{\sigma}, \hat{\Xi}\rangle} \ \textsc{AbsT-Error}$$

$$\frac{\begin{array}{c} \hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(create}\ \ \textit{æ}_a\ \textit{æ}_1 \ldots \textit{æ}_n\texttt{)}, \hat{\rho}), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle \\ \hat{p}' = \widehat{palloc}(\textit{æ}_a, \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle) \qquad \mathbf{actd}(act, \hat{\rho}_a) = \hat{\mathcal{A}}(\textit{æ}_a, \hat{\rho}, \hat{\sigma}) \\ \texttt{(actor (}x_1 \ldots x_n\texttt{)} \ldots\texttt{)} = act \qquad \widehat{addr}_i = \widehat{alloc}(x_i, \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle) \\ \hat{V}_i = \hat{\mathcal{A}}(\textit{æ}_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{\rho}'_a = \hat{\rho}_a[x_i \mapsto \widehat{addr}_i] \qquad \hat{a}' = \mathbf{acti}(act, \hat{\rho}'_a) \end{array}}{\begin{array}{l} \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \widehat{\xmapsto[\mathit{Create}]{\hat{p}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{ko}(\mathbf{pid}(\hat{p}')), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle\right\}] \\ \qquad\qquad \sqcup [\hat{p}' \mapsto \left\{\langle \mathbf{wait}, \epsilon, \hat{a}', \widehat{empty}\rangle\right\}], \\ \qquad\quad \hat{\sigma} \sqcup [\widehat{addr}_i \mapsto \hat{V}_i], \hat{\Xi}\rangle \end{array}} \ \textsc{AbsT-Create}$$

$$\frac{\begin{array}{c} \hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(become}\ \ \textit{æ}_a\ \textit{æ}_1 \ldots \textit{æ}_n\texttt{)}, \hat{\rho}), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle \\ \mathbf{actd}(act, \hat{\rho}_a) = \hat{\mathcal{A}}(\textit{æ}_a, \hat{\rho}, \hat{\sigma}) \\ \texttt{(actor (}x_1 \ldots x_n\texttt{)} \ldots\texttt{)} = act \qquad \widehat{addr}_i = \widehat{alloc}(x_i, \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle) \\ \hat{V}_i = \hat{\mathcal{A}}(\textit{æ}_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{\rho}'_a = \hat{\rho}_a[x_i \mapsto \widehat{addr}_i] \qquad \hat{a}' = \mathbf{acti}(act, \hat{\rho}'_a) \end{array}}{\begin{array}{l} \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \widehat{\xmapsto[\mathit{Become}]{\hat{p}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{ko}(\mathbf{actd}(act, \hat{\rho}_a)), \hat{\kappa}, \hat{a}', \widehat{mb}\rangle\right\}], \\ \qquad\qquad \hat{\sigma} \sqcup [\widehat{addr}_i \mapsto \hat{V}_i], \hat{\Xi}\rangle \end{array}} \ \textsc{AbsT-Become}$$

$$\frac{\begin{array}{c} \hat{\pi}(\hat{p}) \ni \langle \mathbf{wait}, \epsilon, \hat{a}, \widehat{mb}\rangle \qquad ((t, \hat{V}_1 \ldots \hat{V}_n), \widehat{mb}') \in \widehat{deq}(\widehat{mb}) \\ \mathbf{acti}(\texttt{(actor (}x_1 \ldots x_n\texttt{)} \ldots \texttt{(}t\ \texttt{(}y_1 \ldots y_n\texttt{)}\ e\texttt{)} \ldots\texttt{)}, \hat{\rho}_b) = \hat{a} \\ \widehat{addr}_i = \widehat{alloc}(y_i, \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle) \qquad \hat{\rho}'_b = \hat{\rho}_b[y_i \mapsto \widehat{addr}_i] \end{array}}{\begin{array}{l} \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \widehat{\xmapsto[\mathit{Receive}]{\hat{p}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{ev}(e, \hat{\rho}'_b), \epsilon, \hat{a}, \widehat{mb}'\rangle\right\}], \\ \qquad\qquad \hat{\sigma} \sqcup [\widehat{addr}_i \mapsto \hat{V}_i], \hat{\Xi}\rangle \end{array}} \ \textsc{AbsT-Receive}$$

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ko}(\hat{v}), \epsilon, \hat{a}, \widehat{mb}\rangle}{\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \widehat{\xmapsto{\hat{p}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{wait}, \epsilon, \hat{a}, \widehat{mb}\rangle\right\}], \hat{\sigma}, \hat{\Xi}\rangle} \ \textsc{AbsT-Wait}$$

$$\frac{\begin{array}{c} \hat{\pi}(\hat{p}_s) \ni \langle \mathbf{ev}(\texttt{(send}\ \ \textit{æ}_0\ t\ \textit{æ}_1 \ldots ae_n\texttt{)}, \hat{\rho}), \hat{\kappa}_s, \hat{a}_s, \widehat{mb}_s\rangle \qquad \mathbf{pid}(\hat{p}_r) \ni \hat{\mathcal{A}}(\textit{æ}_0, \hat{\rho}, \hat{\sigma}) \\ \hat{\pi}(\hat{p}_r) \ni \langle \hat{c}, \hat{\kappa}_r, \hat{a}_r, \widehat{mb}_r\rangle \qquad \hat{V}_i = \mathcal{A}(\textit{æ}_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{m} = (t, \hat{V}_1 \ldots \hat{V}_n) \end{array}}{\begin{array}{l} \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \widehat{\xmapsto[\mathit{Send}]{\hat{p}_s}} \langle \hat{\pi} \sqcup [\hat{p}_s \mapsto \left\{\langle \mathbf{ko}(\mathbf{pid}(\hat{p}_r)), \hat{\kappa}_s, \hat{a}_s, \widehat{mb}_s\rangle\right\}] \\ \qquad\qquad \sqcup [\hat{p}_r \mapsto \left\{\langle \hat{c}, \hat{\kappa}_r, \hat{a}_r, \widehat{enq}(\hat{m}, \widehat{mb}_r)\rangle\right\}], \\ \qquad\quad \hat{\sigma}, \hat{\Xi}\rangle \end{array}} \ \textsc{AbsT-Send}$$

$$\frac{\begin{array}{c} \hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(send}\ \ \textit{æ}_0\ t\ \textit{æ}_1 \ldots ae_n\texttt{)}, \hat{\rho}), \hat{\kappa}, \hat{a}, \widehat{mb}\rangle \\ \mathbf{pid}(\hat{p}) \ni \hat{\mathcal{A}}(\textit{æ}_0, \hat{\rho}, \hat{\sigma}) \qquad \hat{V}_i = \hat{\mathcal{A}}(\textit{æ}_i, \hat{\rho}, \hat{\sigma}) \qquad \hat{m} = (t, \hat{V}_1 \ldots \hat{V}_n) \end{array}}{\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \widehat{\xmapsto[\mathit{Send}]{\hat{p}}} \langle \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{ko}(\mathbf{pid}(\hat{p})), \hat{\kappa}, \hat{a}, \widehat{enq}(\hat{m}, \widehat{mb})\rangle\right\}], \hat{\sigma}, \hat{\Xi}\rangle} \ \textsc{AbsT-Send-Self}$$

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(terminate)}, \hat{\rho}), \_, \_, \_\rangle}{\langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle \widehat{\xmapsto[\mathit{Terminate}]{\hat{p}}} \langle \hat{\pi}, \hat{\sigma}, \hat{\Xi}\rangle} \ \textsc{AbsT-Terminate}$$

Figure 6 Abstract transition relation for λ_α programs.

Reachability of abstract error states. Predicate $\widehat{ErrorReachable}_e$ may report error states that are never reachable in program e, due to spurious program states. However, every reachable error state is reported. If the analysis reports nothing, the program e contains no reachable error states. It can therefore be used to *prove* the absence of errors in a program.

$$\widehat{ErrorReachable}_e \iff \exists \langle \hat{\pi}, \_, \_ \rangle \in \mathrm{lfp}(\hat{\mathcal{F}}_e), \hat{p} \in \mathrm{Dom}(\hat{\pi}) \mid \hat{\pi}(\hat{p}) \ni \langle \mathbf{error}, \_, \_, \_ \rangle$$

Abstract mailbox bounds. Function $\widehat{MailboxBound}_e(p)$ computes an *upper-bound* on the number of messages in the mailbox of actor p. Because it is an upper-bound, actor p may never reach this bound. However, the mailbox of process p will *never* exceed this bound, because the analysis is sound. Depending on the precision of the mailbox abstraction, $\widehat{size}$ might yield ∞, although the size of the mailbox might be bounded in all concrete executions.

$$\widehat{MailboxBound}_e(p) = max\left(\left\{\widehat{size}(\widehat{mb}) \mid \langle \hat{\pi}, \_, \_ \rangle \in \mathrm{lfp}(\hat{\mathcal{F}}_e) \wedge \hat{\pi}(\alpha(p)) \ni \langle \_, \_, \_, \widehat{mb} \rangle\right\}\right)$$

5 Mailbox Abstractions

The representation of the actors' mailboxes $\widehat{Mbox}$ is a parameter to the analysis. In this section we describe multiple sound instantiations of this parameter. Because mailboxes are merely containers of messages, they do not depend on the values of the messages themselves. Therefore, whether abstract or concrete messages are stored in the abstract mailboxes does not influence their properties nor soundness, and we describe mailbox abstractions in the context of concrete messages for the sake of clarity. Analogous to Section 3.4, it suffices to provide definitions for the following. We define α and $\sqsubseteq$ for each mailbox abstraction and provide complete soundness proofs in an accompanying technical report[2].

- $(\sqsubseteq) \subseteq \widehat{Mbox} \times \widehat{Mbox}$ is a partial order relation.
- $\alpha : Mbox \to \widehat{Mbox}$ is the abstraction function.
- $\widehat{enq} : \widehat{Message} \times \widehat{Mbox} \to \widehat{Mbox}$ enqueues a message at the back of a mailbox.
- $\widehat{deq} : \widehat{Mbox} \to \mathcal{P}(\widehat{Message} \times \widehat{Mbox})$ dequeues a message from the front of a mailbox. Depending on the abstraction, this operation may be non-deterministic. Each element of the resulting set is a tuple containing the message dequeued from the mailbox and the subsequent mailbox.
- $\widehat{size} : \widehat{Mbox} \to \mathbb{N} \cup \{\infty\}$ computes the size of a mailbox, and may over-approximate.
- $\widehat{empty} : \widehat{Mbox}$ represents the empty mailbox.

A mailbox abstraction is sound if all of the above definitions are sound over-approximations of their concrete counterparts. Formally, this means the following.

- $\widehat{enq}$ is a sound over-approximation of enq: $\forall m, mb : \alpha(enq(m, mb)) \sqsubseteq \widehat{enq}(m, \alpha(mb))$.
- $\widehat{deq}$ is a sound over-approximation of deq: $\forall m, mb, mb' : (m, mb') \in deq(mb) \implies \exists \widehat{mb}', (m, \widehat{mb}') \in \widehat{deq}(\alpha(mb)) \wedge \alpha(mb') \sqsubseteq \widehat{mb}'$.
- $\widehat{size}$ is a sound over-approximation of $size$: $\forall mb, size(mb) \leq \widehat{size}(\alpha(mb))$.
- $\widehat{empty}$ represents the empty mailbox $empty$: $\alpha(empty) = \widehat{empty}$.

[2] https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf.

Table 1 Categorization of the concrete *List* mailbox and five mailbox abstractions.

	Ordering	**No Ordering**
Multiplicity	List, Bounded List (§5.3)	Multiset (§5.4), Bounded Multiset (§5.5)
No Multiplicity	Graph (§5.6)	Powerset (§5.2)

5.1 Categorization of Mailbox Abstractions

We now describe one unbounded (*Multiset*) and four bounded (*Powerset*, *Bounded List*, *Bounded Multiset*, *Graph*) mailbox abstractions. When the domain of messages is finite, all bounded mailbox abstractions are also finite. The domain of messages is finite if the value domain itself is finite, which is the case when abstract process identifiers are also finite. Note that a finite number of abstract process identifiers does not limit the analysis to programs with bounded actors, as discussed in Section 7.5.

Table 1 depicts a two-dimensional categorization of the mailbox abstractions. A mailbox abstraction preserves message *ordering* information if it can encode which messages have arrived before others (partially or up to some bound), that is, $\alpha(m_1 :: m_2 :: mb) \neq \alpha(m_2 :: m_1 :: mb)$. A mailbox abstraction preserves message *multiplicity* if it can encode the number of times a message has been received (up to some bound), that is, there exists a bound such that $\alpha(m : mb) \neq \alpha(m : m : mb)$. For completeness, Table 1 also categorizes the concrete (unbounded) *List* mailbox introduced in Section 3.4.

5.2 Powerset Abstraction

The powerset abstraction, used by D'Osualdo et al. [17], abstracts a concrete mailbox to the set of messages it contains.

$$\begin{aligned} \widehat{mb} \in PS &= \mathcal{P}(Message) & deq_{PS}(\widehat{mb}) &= \left\{(m, \widehat{mb}), (m, \widehat{mb} - m) \mid m \in \widehat{mb}\right\} \\ empty_{PS} &= \varnothing & size_{PS}(\varnothing) &= 0 \\ enq_{PS}(m, \widehat{mb}) &= \widehat{mb} \cup \{m\} & size_{PS}(\widehat{mb}) &= \infty \end{aligned}$$

Though sound, this coarse abstraction only keeps track of which messages are present in the mailbox, and preserves neither ordering nor multiplicity of messages.

5.3 Bounded List Abstraction

Combining the powerset abstraction with a bounded concrete mailbox results in the *bounded list* abstraction. It is defined as follows for a bound of n, where α_{L_n} is the abstraction function that converts a list to a set if its length exceeds the bound.

$$\begin{aligned} \widehat{mb} \in L_n &= Mbox \mid PS & & & deq_{L_n}(\widehat{mb}) &= deq_{PS}(\widehat{mb}) & &\text{if } \widehat{mb} \in PS \\ empty_{L_n} &= \epsilon & & & &= deq(\widehat{mb}) & &\text{if } \widehat{mb} \in Mbox \\ enq_{L_n}(m, \widehat{mb}) &= enq_{PS}(m, \widehat{mb}) & &\text{if } \widehat{mb} \in PS & size_{L_n}(\widehat{mb}) &= size_{PS}(\widehat{mb}) & &\text{if } \widehat{mb} \in PS \\ &= \alpha_{L_n}(enq(m, \widehat{mb})) & &\text{if } \widehat{mb} \in Mbox & &= size(\widehat{mb}) & &\text{if } \widehat{mb} \in Mbox \end{aligned}$$

We write $L_{\geq n}$ to denote bounded list abstractions with a bound of at least n. This sound abstraction preserves full precision over the messages in a mailbox —ordering and multiplicity are both preserved— up to the point where the bound is reached. Once the number of messages in the mailbox exceeds the bound n, the bounded list abstraction behaves like the powerset abstraction, rendering it finite.

5.4 Multiset Abstraction

The list of messages can be abstracted to a multiset that keeps track of the multiplicity of each message, but has no ordering information.

$$
\begin{aligned}
\widehat{mb} \in \mathit{MS} &= M \to \mathbb{N} \\
\mathit{empty}_{\mathit{MS}} &= \lambda x.0 \\
\mathit{enq}_{\mathit{MS}}(m, \widehat{mb}) &= \widehat{mb}[m \mapsto \widehat{mb}(m) + 1]
\end{aligned}
\qquad
\begin{aligned}
\mathit{deq}_{\mathit{MS}}(\widehat{mb}) &= \{(m, \widehat{mb}[m \mapsto \widehat{mb}(m) - 1]) \\
&\qquad \mid m \in \mathrm{Dom}\,(mb) \;\wedge\; \widehat{mb}(m) \geq 1\} \\
\mathit{size}_{\mathit{MS}}(\widehat{mb}) &= \sum_{m \in \mathrm{Dom}(\widehat{mb})} \widehat{mb}(m)
\end{aligned}
$$

The multiset abstraction is sound but unbounded: there is no bound on the number of times each message may appear.

5.5 Bounded Multiset Abstraction

The multiset abstraction can be made finite by imposing a bound on the multiplicity of each message. Once this bound is exceeded for a message, the multiplicity of that message is abstracted and becomes ∞.

$$
\begin{aligned}
\widehat{mb} \in \mathit{MS}_n &= M \to (\mathbb{N}^{\leq n} \cup \{\infty\}) \\
\mathit{empty}_{\mathit{MS}_n} &= \lambda x.0 \\
\mathit{size}_{\mathit{MS}_n}(\widehat{mb}) &= \sum_{m \in \mathrm{Dom}(\widehat{mb})} \widehat{mb}(m)
\end{aligned}
\qquad
\begin{aligned}
\mathit{enq}_{\mathit{MS}_n}(m, \widehat{mb}) &= \widehat{mb}[m \mapsto \widehat{mb}(m) + 1] && \text{if } \widehat{mb}(m) < n \\
&= \widehat{mb}[m \mapsto \infty] && \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{deq}_{\mathit{MS}_n}(\widehat{mb}) = &\Big\{(m, \widehat{mb}[m \mapsto \widehat{mb}(m) - 1]) \mid m \in \mathrm{Dom}(\widehat{mb}) \;\wedge\; 1 \leq \widehat{mb}(m) \leq m\Big\} \\
&\cup \Big\{(m, \widehat{mb}), (m, \widehat{mb}[m \mapsto 0]) \mid m \in \mathrm{Dom}(\widehat{mb}) \;\wedge\; \widehat{mb}(m) = \infty\Big\}
\end{aligned}
$$

We write $\mathit{MS}_{\geq n}$ to denote multiset abstractions with a bound of at least n.

5.6 Graph Abstraction

We propose graphs as a new mailbox abstraction that preserves ordering. A mailbox is abstracted by a graph in which the nodes correspond to messages and the edges denote an ordering relation between messages: an edge between node a and b indicates that b appears after a in the mailbox. This abstraction also maintains information about the first and last message in the mailbox. Figure 7 depicts the following evolution of a mailbox using this abstraction.

- Enqueuing message 0 on the empty mailbox creates a node 0, and makes the *first* (f) and *last* (l) pointers point to this node (Figure 7a).
- Enqueuing message 1 creates a new node connected to the previous *first* node, updates the *first* pointer, but leaves the *last* pointer as is (Figure 7b).
- Enqueuing message 0 does not create a new node since the node 0 is already in the graph, but does add a new edge from `1` to 0, and updates the *first* pointer (Figure 7c).
- Dequeuing a message yields the message pointed by the *last* node. The resulting mailbox has the same graph, but the *last* node is updated to point to a successor of its current node (Figure 7d).

Informally, upon a dequeue operation, the node pointed by the l pointer is returned, and the mailbox is updated so that the *last* pointer points to a successor node of the returned node.

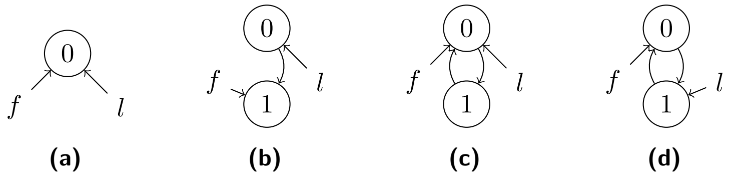

Figure 7 Visual representation of the graph abstraction.

Upon a queue operation, a new node is added with the corresponding message, the f pointer is updated to point to this new node, and an edge is added between this new node and the old node pointed by the f pointer. The size of the mailbox is known only when there is a single path from the l node to the l node, otherwise the size is approximated by ∞.

$$\begin{aligned}\widehat{mb} \in G = (&\mathcal{P}(\mathit{Message}) \\ &\times \mathcal{P}(\mathit{Message} \times \mathit{Message}) \\ &\times \mathit{Message} \times \mathit{Message}) \cup \{\bot\} \\ \mathit{empty}_G &= \langle \varnothing, \varnothing, \bot \rangle\end{aligned}$$

$$\begin{aligned}\mathit{size}_G(\bot) &= 0 \\ \mathit{size}_G(\langle V, E, f, l \rangle) &= 1 + \mathit{PathLength}(l, f, \langle V, E \rangle) \\ \mathit{enq}_G(m, \bot) &= \langle \{m\}, \{\}, m, m \rangle \\ \mathit{enq}_G(m, \langle V, E, f, l \rangle) &= \langle V \cup \{m\}, E \cup \{\langle f, m \rangle\}, m, l \rangle\end{aligned}$$

$$\begin{aligned}\mathit{deq}_G(\bot) &= \varnothing \\ \mathit{deq}_G(\langle V, E, f, l \rangle) &= \{(l, \bot)\} && \text{if } |\{(l, l') \in E \mid l' \in V\}| = 0 \\ \mathit{deq}_G(\langle V, E, f, l \rangle) &= \{(l, \langle V, E, f, l' \rangle) \mid (l, l') \in E, l' \in V\} && \text{otherwise}\end{aligned}$$

PathLength (defined in the accompanying technical report[3]) computes the length of the unique path between l and f. If no such unique path exists, it over-approximates with ∞. This sound abstraction preserves ordering information but does not preserve multiplicity. However, when there exists a single path from l to f, the size of the mailbox is equal to the length of that path. Function *PathLength* returns n if there is a single path between l and f, and this path has length n. Otherwise, it returns ∞. For example, this is the case in Figures 7a and 7b, but not in Figure 7c nor in Figure 7d. The graph abstraction is finite when the domain of messages is finite, and needs no bounding.

6 Evaluation

We used our implementation (Section 6.1) to evaluate the applicability of the different mailbox abstractions on a set of benchmark programs (Section 6.2). The experiments were executed with Scala 2.12.1 on a MacBook Pro with a 2.8 GHz i7 processor and 16 GB of memory. We compare mailbox abstractions in terms of running time of the analysis and size of the flow graph generated (Section 6.3), and precision (Section 6.4). Timing information represents the average of running each benchmark 10 times after 2 warmup runs. We also compare our implementation with Soter (Section 6.5), a state-of-the-art analyzer for Erlang, and conclude with some remarks on soundness (Section 6.6).

[3] `https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf`

6.1 Implementation

We implemented the technique presented in this paper in a modular static analysis tool [37], which is freely available[4]. The prototype is implemented in Scala and supports the actor model of λ_α on top of a subset of R5RS Scheme. It implements the mailbox abstractions presented in Section 5. We incorporated two additional optimizations: global store widening and abstract counting. Global store widening [38] is an abstraction that reduces the precision of the analysis in order to reach a fixed point faster. Abstract counting [34] replaces joins with updates in the process map when it is known that a process identifier maps to a single abstract actor.

6.2 Benchmarks

We translated benchmarks from multiple sources to λ_α, remaining as close as possible to their original implementation. We unrolled all loops that create a fixed number of actors, in order to benefit from the additional precision offered by abstract counting. Solutions to overcome this need for unrolling loops are given in Section 7.5. Moreover, in order to compare our approach with Soter, which analyzes Erlang programs, we also faithfully translated all the benchmarks in Erlang. The correspondance between the λ_α and Erlang versions of the benchmarks is as close as possible. We used the following benchmark programs for evaluation, which reflect specific patterns of mailboxes in actor programs and are in line with related work. They range from 12 LOC to 32 LOC.

- `pp`, `count`, `count-seq`, `fjt-seq`, `fjc-seq`: benchmark programs from the Savina benchmark suite [27], translated from Scala.
- `factorial`, `stack`: benchmark programs from Agha [1], translated from pseudo-code.
- `cell`: a typical example actor program.
- `parikh`, `pipe-seq`, `unsafe-send`, `safe-send`, `state-factory`, `stutter`: benchmark programs from Soter [16], translated from Erlang.

Note that all the benchmarks create a fixed number of actors (Table 2). When run with abstract semantics, this can correspond to the same number of abstract actors, or to fewer abstract actors, where one abstract actors models the behavior of a group of concrete actors (e.g., in `factorial`). We did not target benchmarks with an unbounded number of concrete actors, as this is an orthogonal problem to the points discussed in this paper. We discuss this case in Section 7.5.

6.3 Running Time and Flow Graph Size

We measured the impact of the different mailbox abstractions on the size of the flow graph generated by the analysis. Similarly to bounded model checking [6], the bounds for the multiset and list mailbox abstractions were determined by running each benchmark with increasing bounds ($n = 1, 2, \ldots$) for each of these bounded abstractions, selecting the lowest bound yielding maximal precision.

From the results of our experiments, summarized in Table 2, we conclude that the graph abstraction generally yields the smallest, or close to the smallest, number of states. Using the graph abstraction also resulted in the lowest running time in 7 out of 14 benchmarks. The powerset abstraction, on the other hand, yields comparatively poor results in general, timing out in 2 out of 14 benchmarks.

[4] `https://github.com/acieroid/scala-am`.

Table 2 Number of states (#s) and time taken (t, in milliseconds) to generate the flow graphs for each bounded mailbox abstraction. A time of ∞ means that the time limit of 60 seconds was exceeded; in this case #s is the number of states that have been explored when the time limit was reached. The size of the smallest flow graph on each row is underlined. The column P indicates the number of processes for each benchmark.

Benchmark	P	*PS* (powerset)		MS_n (multiset)			L_n (list)			*G* (graph)	
		#s	t	n	#s	t	n	#s	t	#s	t
`pp`	3	21	352	1	$\underline{8}$	24	1	$\underline{8}$	18	$\underline{8}$	15
`count`	3	83	829	1	22	90	1	$\underline{21}$	96	22	90
`count-seq`	3	45	207	1	10	15	2	$\underline{8}$	8	$\underline{8}$	8
`fjt-seq`	4	$\underline{201}$	4609	1	589	12191	1	589	9746	589	8832
`fjc-seq`	4	$\underline{15}$	38	1	$\underline{15}$	21	1	$\underline{15}$	22	$\underline{15}$	25
`factorial`	8	1486+	∞	1	46	1009	1	52	1644	$\underline{22}$	155
`stack`	3	85	636	1	42	46	4	$\underline{16}$	13	$\underline{16}$	13
`cell`	3	70	313	1	23	18	2	$\underline{15}$	11	$\underline{15}$	12
`parikh`	3	31	49	1	$\underline{8}$	7	2	$\underline{8}$	7	$\underline{8}$	8
`pipe-seq`	4	2662+	∞	1	$\underline{24}$	56	1	$\underline{24}$	47	$\underline{24}$	55
`unsafe-send`	2	4	4	1	$\underline{3}$	3	1	$\underline{3}$	3	$\underline{3}$	3
`safe-send`	2	100	273	1	32	29	4	$\underline{28}$	17	30	19
`state-factory`	3	76	553	1	$\underline{43}$	274	1	160	745	214	814
`stutter`	2	28	76	1	60	103	1	34	79	$\underline{15}$	23

The results also show that in 4 out of 14 benchmarks the bound for the list abstraction needs to be higher than the bound for the multiset abstraction to achieve maximal precision. In the `count-seq` benchmark, for example, a `counting` actor receives two kinds of messages: `increment` and `retrieve`. The bounded list abstraction therefore requires a bound of 2 to analyze this program precisely. On the other hand, the bound for the multiset abstraction is not on the size of the mailbox, but at the level of individual messages. The `increment` and `retrieve` messages appear only once, and therefore the multiset mailbox abstraction can analyze this program precisely with a bound of 1.

6.4 Precision

To measure the precision of the different mailbox abstractions, we compared mailboxes and dequeued messages during static analysis with their corresponding concrete values. Resulting from over-approximation, a spurious abstract element lacks corresponding concrete elements in actual runs of the program. The more *spurious* elements, the less precise the results of an analysis. We counted the following spurious elements in the analysis results and summed the results for all benchmarks (Table 8).

1. Spurious mailboxes.
2. Spurious messages resulting from spurious mailboxes (*Spurious Messages #1*).
3. Spurious messages resulting from non-spurious mailboxes (*Spurious Messages #2*).

Any message dequeued from a spurious mailbox is a spurious message, directly linking the number of such spurious messages to the number of spurious mailboxes. This link is not that direct for spurious messages resulting from non-spurious mailboxes, and at least a different mailbox abstraction is required to decrease the number of spurious messages in this category. For example, a non-empty mailbox will always yield spurious messages if abstracted by a powerset, no matter the precision of the other abstractions used in the analysis.

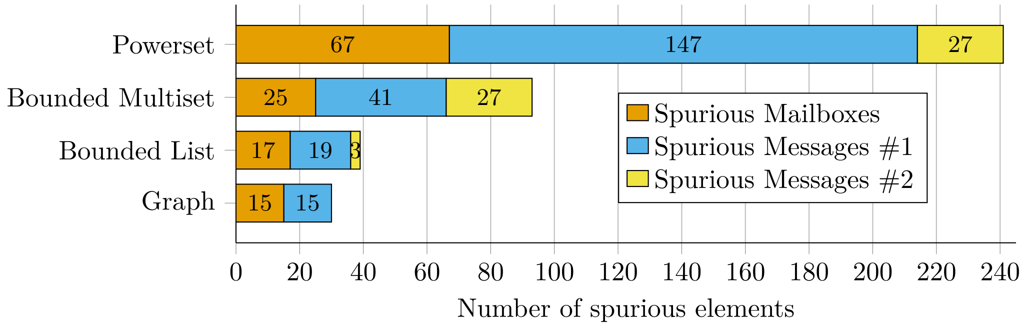

Figure 8 Precision metrics for the different mailbox abstractions (lower is better).

The results show that the coarse powerset abstraction is the most *imprecise* abstraction, resulting in many spurious elements. These spurious elements in turn result in spurious program states (Section 6.3), rendering the abstraction not scalable. This makes the use of the powerset abstraction unsuitable for proving program properties directly (Section 6.5).

The multi-set abstraction benefits from higher precision because it preserves multiplicity, therefore resulting in fewer spurious mailboxes. However, because it lacks order information, it does not improve over the powerset abstraction in the number of spurious messages resulting from non-spurious mailboxes.

The list and graph abstraction preserve both multiplicity and ordering, which renders them more precise. On benchmarks with an unbounded number of messages, both lose some precision. However, when the messages in an unbounded mailbox follow a specific pattern, the graph abstraction yields a better precision than the bounded list abstraction. This is because the list abstraction reduces to a powerset once the bound is reached, thereby losing ordering information. This is the case in the `stutter` benchmark, where the list abstraction results in 10 spurious elements, while the graph analyzes it with full precision (i.e., without spurious elements).

The only benchmark where the graph abstraction yields more spurious elements than the other abstractions is `state-factory`. This is because this benchmarks contains an actor receiving a specific message an unbounded number of times, as well as a single instance of another message. Due to the specific message being received an unbounded number of times, the graph abstraction does not maintain the multiplicity information over the message that is unique. Using the bounded multiset abstraction on the other hand preserves this multiplicity and yields no spurious elements. Using the powerset and bounded list abstractions does not preserve this multiplicity information, yielding spurious elements. However because these abstractions have a smaller domain size, they produce less spurious mailboxes in comparison with the graph abstraction (2 for the powerset abstraction, 3 for the bounded list abstraction, 6 for the graph abstraction).

6.5 Comparison with Soter

We compare our analysis of λ_α with Soter, a state-of-the-art analysis tool for Erlang programs [17]. We translated our benchmarks to Erlang in order to analyze them with Soter. The result of running Soter and our analysis on these benchmarks is given in Table 3. Some benchmarks have unbounded mailboxes, hence there is no bound to prove; other benchmarks make no use of the `error` construct, hence the absence of error is trivial. These benchmarks are therefore not included in Table 3.

Table 3 Comparison with Soter. Column *Type* is the verified property: absence of run-time errors (*Err.*) or bound on some mailbox (*Bnd.*). Column *Safe* is the expected analysis result. For both Soter and our technique, column *Res.* gives the result of the analysis, column t is the running time of the full analysis, and column *Abs.* lists the abstractions used. The time given for our technique is the range of the time taken by the abstractions listed in *Abs.*

			Soter			Us		
Benchmark	Type	Safe	Res.	Abs.	t (ms)	Res.	Abs.	t (ms)
`parikh`	Err.	✓	✓	D_0	38	✓	$MS_{\geq 1}, L_{\geq 1}, G$	7 – 8
`unsafe-send`	Err.	✗	✗	D_0	13	✗	$PS, MS_{\geq 0}, L_{\geq 0}, G$	3 – 4
`safe-send`	Err.	✓	✓	D_1	267	✓	$L_{\geq 4}, G$	17 – 19
`stutter`	Err.	✓	✗	D_2	53	✓	G	23 – 23
`stack`	Err.	✓	✗	D_2	2260	✓	$L_{\geq 4}, G$	13 – 13
`count-seq`	Err.	✓	✗	D_2	109	✓	$L_{\geq 2}, G$	8 – 8
`cell`	Err.	✓	✗	D_2	383	✓	$L_{\geq 2}, G$	11 – 12
`pipe-seq`	Bnd.	✓	✓	D_0	165	✓	$MS_{\geq 1}, L_{\geq 1}, G$	47 – 55
`state-factory`	Bnd.	✓	✓	D_0	622	✓	$MS_{\geq 1}, G$	274 – 814
`pp`	Bnd.	✓	✓	D_0	95	✓	$MS_{\geq 1}, L_{\geq 1}, G$	15 – 24
`count-seq`	Bnd.	✓	✓	D_0	71	✓	$MS_{\geq 1}, L_{\geq 2}, G$	8 – 10
`cell`	Bnd.	✓	✗	D_2	383	✓	$MS_{\geq 1}, L_{\geq 2}, G$	11 – 18
`fjc-seq`	Bnd.	✓	✗	D_2	81	✓	$MS_{\geq 1}, L_{\geq 1}, G$	21 – 25
`fjt-seq`	Bnd.	✓	✗*	n.a.	n.a.	✗	n.a.	n.a.

For both Soter and our technique, column *Abs.* lists the abstractions that enable verification of either the absence of errors or the bound on mailboxes. This is with a simple query on the generated flow graph alone in our case, and with some more complex post-processing for Soter. In the case of Soter, the only tunable parameter is the *data abstraction depth*, which varies between 0 and 2. We chose the lowest data abstraction depth that could be used to verify the properties, and else used an abstraction depth of 2. In the case of our technique, we list all mailbox abstractions that enabled proving each program property. In practice, choosing an abstraction to verify each program can be automated by running the analysis with each abstraction, increasing the bound for bounded abstractions, until one is able to prove the property. If no abstraction can be used to prove the property, one can conclude that either the property does not hold, or that the analysis yields a false positive. Overall, we see that our technique is able to verify mailbox bounds and the absence of run-time errors in a similar amount of time as Soter. With a proper mailbox abstractions the analysis takes less than one second for each benchmark.

An important distinction between our approach and Soter is that Soter generates a coarse flow graph as the model of a program, and then performs model checking on this graph to verify program properties. Our technique constructs a more precise flow graph of the program on which the verification can be performed directly, not requiring a separate model checking step to prove the absence of run-time errors or bounds on mailboxes. To highlight this difference, consider the `parikh` benchmark. It contains a `server` actor that expects `init` as a first message, but throws an error if it receives a second `init` message. With a powerset mailbox abstraction, which does not preserve multiplicity, the error is reachable in the graph generated by Soter. However, it can be proved unreachable by performing an extra model-checking step. On the other hand, our approach benefits from improved precision from the mailbox abstraction, resulting in a smaller and more precise flow graph that does *not* contain the error state. No further steps are therefore required.

Additionally, we are able to handle programs that Soter cannot handle. For example, `stutter` needs a mailbox abstraction that preserves ordering information among an unbounded number of messages following the pattern of Figure 7, and for which the graph abstraction is ideally suited. As another example, `stack` needs a mailbox abstraction that preserves ordering information on four consecutive messages. Note that on `fjt-seq`, our technique fail to prove the required bound. However, Soter produces unsound results: it proves a bound that is lower than the expected bound.

6.6 Soundness

The approach presented in this paper combines sound techniques: systematic abstraction of abstract machines [38], ordered macro-stepping semantics (a variant of macro-stepping semantics of Agha et al. [2]), and sound mailbox abstractions.. To prove the soundness of the analysis, we first note that the abstract semantics over-approximate the concrete semantics.

▶ **Theorem 1** (($\widehat{\longmapsto}$) is a sound over-approximation of ($\longmapsto$))**.** *If we have* $\varsigma_1 \overset{p}{\underset{E}{\longmapsto}} \varsigma_2$*, and* $\alpha(\varsigma_1) \sqsubseteq \hat{\varsigma}_1$*, then* $\exists \hat{\varsigma}_2$ *such that* $\hat{\varsigma}_1 \widehat{\overset{p}{\underset{E}{\longmapsto}}} \hat{\varsigma}_2$*,* $\alpha(\varsigma_2) \sqsubseteq \hat{\varsigma}_2$ *and* $\alpha(p) = \hat{p}$*.*

Proof. The proof follows a similar structure as in Van Horn and Might [38] and D'Osualdo [15], and is based on the soundness of mailbox abstractions (proven in the accompanying technical report[5]). Note that any address allocation strategy leads to a sound analysis [32, 23]. ◀

Our abstract version of macro-stepping semantics combines multiple small steps into a macro-step, in a sound manner (Theorem 3).

▶ **Theorem 2** (($\widehat{\longmapsto}^{*\downarrow}$) is a sound over-approximation of ($\longmapsto^{*\downarrow}$))**.** *If we have* $\varsigma_1 \overset{p}{\underset{E}{\longmapsto}}{}^{*\downarrow} \varsigma_N$ *and* $\alpha(\varsigma_1) \sqsubseteq \hat{\varsigma}_N$*, then* $\exists \hat{\varsigma}_N$ *such that* $\hat{\varsigma}_1 \widehat{\overset{p}{\underset{E}{\longmapsto}}}{}^{*\downarrow} \hat{\varsigma}_N$*,* $\alpha(\varsigma_N) \sqsubseteq \hat{\varsigma}_N$ *and* $\alpha(p) \sqsubseteq \hat{p}$*.*

Proof. The proof is by induction on the rules of $\longmapsto^{*\downarrow}$. For the cases M-Stop and M-Blocked, the proof directly follows from Theorem 1. The case M-Main consists of two parts: a first step of $\widehat{\mapsto}$, proven by Theorem 1, and a second step of $\widehat{\mapsto}^{*\downarrow}$ that follows by the induction hypothesis. ◀

▶ **Theorem 3** (($\widehat{\longmapsto}^{M}$) is a sound over-approximation of ($\longmapsto^{M}$))**.** *If we have* $\varsigma_1 \overset{p}{\underset{E}{\longmapsto}}{}^{M} \varsigma_N$ *and* $\alpha(\varsigma_1) \sqsubseteq \hat{\varsigma}_N$*, then* $\exists \hat{\varsigma}_N$ *such that* $\hat{\varsigma}_1 \widehat{\overset{p}{\underset{E}{\longmapsto}}}{}^{M} \hat{\varsigma}_N$*,* $\alpha(\varsigma_N) \sqsubseteq \hat{\varsigma}_N$ *and* $\alpha(p) \sqsubseteq \hat{p}$*.*

Proof. A macro-step is the composition of an unrestricted small-step followed by a restricted multi-step. Soundness therefore follows from Theorems 1 and 2. ◀

Our analysis therefore forms a sound *over-approximation* of the concrete semantics of a program.

[5] `https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf`

7 Related Work

In this paper, we aim at providing a sound over-approximation of the behavior of actor programs. A number of existing tools supporting actors aim for a different goal: providing a very precise under-approximation. That is, tools based on model checking and concolic testing can detect errors in actor programs, based on a number of concrete executions of a program. They are said to be *sound for defect detection* [7] in that any detected error is an error that will arise under certain conditions. However, such tools can only prove the absence of errors by exploring the entire set of possible executions of a program, which might not be finite due to the numerous sources of unboundedness. Our technique, in contrast, is *sound for correctness*: if our technique cannot detect a defect, it proves that the given program is free of that defect under all possible inputs and interleavings. However, if a defect is detected, it might be a false positive resulting from a too coarse abstraction. Identifying whether a detected defect is a false positive or a true defect is up to the user of the analysis, and can be a burden if the number of false positives is high. Reducing the number of false positives of an analysis is important in order to reduce the burden on the user [9].

Similarly to D'Osualdo et al. [17], we apply the *abstracting abstract machine* (AAM) technique of Van Horn and Might [38] to actor programs. This technique enables a systematic, sound abstraction of concrete semantics given as an abstract machine. Instead of applying AAM to build a coarse model of the program and then performing model-checking on that model (as done by D'Osualdo et al. [17]), we use AAM as the only step in our static analysis. We show that with proper mailbox abstractions, this single step is sufficient to verify properties such as absence of errors and mailbox bounds, with a better precision than D'Osualdo et al. [17]. Our technique has two limitations: it does not deal with programs in which the number of actors is unbounded, and it reasons about every possible message interleaving. Both of these problems impact the scalability of the technique, but nonetheless should not overshadow the contributions of this paper. Indeed, our formalizations and observations of the properties of different mailbox abstractions are applicable to other static analysis techniques than AAM.

7.1 Actor Languages

In this paper, we focus on actors following the *classical* actor model introduced by Agha [1]. The foundations of this model have been formalized in detail by Agha et al. [2], with the difference that mailboxes are represented by multisets. We represent concrete mailboxes for each actor by queues, in order to be able to model other mailbox formalisms and implementations that assume that mailboxes are ordered [1, 25, 24, 3, 20]. Another difference with Agha et al. [2] is that we do not restrict values that can be communicated: our formalization supports messages that contain closures. For a recent survey of existing actor models and their specificities, we refer to De Koster et al. [14].

The concept of macro-stepping is introduced in Agha et al. [2], where a macro step is defined as multiple small steps made within a single actor between the reception of two messages. We introduce ordered macro-stepping, a finer-grained variant of macro-stepping that properly accounts for interleavings of message sends. This is because regular macro-stepping is not sound for analyzing programs from ordered-message mailbox actor models.

7.2 Abstract Interpretation of Actor Programs

Huch [26] represents some of the earliest work on static analysis of actors-based programs through abstract interpretation. The author identifies four sources of unboundedness that render analyzing actor programs challenging: data unboundedness, stack unboundedness, mailbox unboundedness, and unboundedness of the number of spawned processes. He solves the first two sources of unboundedness, and mitigates the last two by framing the analysis in the context of programs that "use only finite parts of the message queues and create only finitely many processes". Our analysis deals with unbounded number of messages, but we leave the problem of unbounded processes for future work.

A closely related work to ours is Soter [16, 17], to which we compare in Section 6. Static checks included in Erlang's analyzer `dialyzer` [7, 8] are *sound for defect detection*. Our approach is over-approximative and therefore *sound for correctness*.

Garoche et al. [22] present an abstract interpretation approach to verify properties of an actor calculus. The focus is on abstractions that enable reasoning about the number of actors bound to a process identifier, while this paper focuses on abstractions to reason about the mailbox content of an actor. Garoche et al. [21] extends the earlier approach to detect orphan messages in actor programs, using a *vector addition system*, similarly to D'Osualdo et al. [17]. The difference with our work is that we reason about the content of mailboxes while performing the control-flow analysis, while both Garoche et al. [21] and D'Osualdo et al. [17] only do so at a later stage. Moreover, Garoche et al. [21] uses the multiset representation for concrete mailboxes, while we take ordering information into account.

7.3 Type Systems

Multiple type systems have been formalized for actor programs. However, most of them only focus on detecting type errors in the sequential subset of the language [29, 30]. A notable exception is Dagnat and Pantel [11]. This type system focuses on detecting messages that will *never* be handled. However, it reasons about global properties of actors, while our analysis is able to reason about actors at different moments in their lifetime.

7.4 Model Checking and Specification Logics

Dam and Fredlund [12] introduce a specification logic and proof system for Core Erlang programs that can be used to perform model-checking on Erlang programs. This approach has been integrated in the Erlang Verification Tool [4], later extended to deal with OTP-specific constructs such as `gen_server` [5]. It supports verifying that an implementation satisfies a given specification, but is not fully automated like our approach.

Both dCUTE [36] and Basset [28] perform automated testing on actor programs and exploit reduction techniques to reduce the size of the explored state space. dCUTE uses concolic testing and incorporates dynamic partial order reduction (DPOR), while Basset uses model checking and allows to choose between DPOR or an actor-specific state comparison reduction technique. Both rely on concrete execution of the program, and only terminate if the program itself terminates. These techniques are sound for defect detection, while ours is sound for correctness and guaranteed to terminate in finite time. A common point is the use of macro-stepping to reduce the number of interleavings to explore. However, as we do assume ordering on the mailbox, we use the finer-grained ordered macro-stepping.

7.5 Limitations and Future Work

The main limitations of our work have an impact on the scalability of the analysis. They do not diminish the contributions of this paper. The different mailbox abstractions we propose, the evaluation of their impact on the properties of the analysis, and the adaptation of macro-stepping semantics to actor models with ordered mailboxes are our main contributions. These contributions are not limited to the analysis framework described in this paper.

The two main limitations, and how they could be addressed in the future, are the following.

- The use of abstract counting is crucial to obtain the precise results of Section 6. Without it, the analysis is unable to yield useful results. But even with abstract counting, results can become too imprecise if an abstract process identifier corresponds to more than one concrete actor. This is why we had to adapt some benchmarks in order to have different call sites for each created actor, so that each would get associated with a different process identifier. One solution to this problem is using a more precise context-sensitivity, so that multiple actors created at the same call site in different contexts are mapped to different process identifiers. But, the analysis and its precision have to be finite, so precision has to be lost at some point. To reason precisely about programs with an unbounded number of actors (e.g., where the number of actors spawned is dependent on user input), this precision loss will have to be remedied.
- While our analysis uses macro-stepping to reduce the amount of non-determinism, it still explores a program under all the possible message interleavings. Scaling to larger programs where that number of interleavings can become tremendous remains problematic. There is extensive literature on how to tackle this problem in the context of shared-memory concurrency [35, 18], and it has also been explored in the context of concolic testing of actor programs [36, 28]. We plan on adapting these techniques to our framework.

Note that in the language considered, messages are assumed to be received in the same order as sent. This limits the analysis to a local setting. Extending the analysis to a distributed setting where messages may be reordered under certain conditions[6] would require to relax this assumption.

We did not discuss the possible extension of this work to analyze programs that do not guarantee actor isolation. In order to analyze for example actor programs written in Scala, which may contain actors that share memory, it is necessary to adapt the analysis. However, the necessary changes are isolated thanks to the modular design of our approach: one has to introduce a new effect to represent reads and writes to shared memory, and to adapt the macro-stepping semantics so that a macro-step is interrupted upon side effects. This is done by redefining function f of Section 3.6.

8 Conclusion

We presented a framework for statically analyzing actor-based programs through abstract interpretation. Starting from the concrete semantics of an actor language, we apply systematic abstraction in order to obtain an abstract interpreter for that language. We introduce and incorporate a finer-grained variant of macro-stepping that we call ordered macro-stepping. This is because several actor models feature mailboxes that preserve ordering information about their messages, for which regular macro-stepping results in a static analysis that

[6] For example, Erlang ensures that messages sent from a given actors will be received in the same order, but nothing is guaranteed about the order of the messages sent from different actors.

may miss execution interleavings and therefore is unsound. We identify the abstraction used for the actors' mailboxes as a key component of any analysis for actor-based programs. Our analysis is therefore parameterized by the mailbox abstraction used, and we provide different instantiations of this parameter that differ in the extent to which the multiplicity and ordering of messages is preserved.

We evaluated the applicability of the different mailbox abstractions on a set of benchmark programs with regard to two program properties: absence of errors, and bounds on mailbox sizes. The use of suitable mailbox abstractions enabled our analysis to verify programs properties that related work could not. We found that the prevalent powerset mailbox abstraction, which preserves neither multiplicity nor ordering, is too imprecise to prove these properties. Using a graph-based mailbox abstraction, in contrast, resulted in sufficiently small flow graphs that enable proving them for all benchmark programs. Our results also show that our improvements in the precision of the computed flow graphs obviate the need for a separate model checking step.

We conclude that sound and precise abstraction of mailboxes is crucial to the precision of any static analysis for actor-based programs. Our work demonstrates that a well-chosen mailbox abstraction can improve the precision of the analysis significantly, thus enabling static verification of the absence of errors and the computation of mailbox bounds.

References

1 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

2 Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

3 Joe Armstrong. *Programming erlang : software for a concurrent world*. Pragmatic programmers. Pragmatic Bookshelf, 2007.

4 Thomas Arts, Mads Dam, Lars-Åke Fredlund, and Dilian Gurov. System description: Verification of distributed erlang programs. In *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, pages 38–41, 1998.

5 Thomas Arts and Thomas Noll. Verifying generic erlang client-server implementations. In *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, pages 37–52, 2000.

6 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

7 Maria Christakis and Konstantinos Sagonas. Static detection of race conditions in erlang. In *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, pages 119–133, 2010.

8 Maria Christakis and Konstantinos Sagonas. Static detection of deadlocks in erlang. Technical report, 2011.

9 Patrick Cousot. The verification grand challenge and abstract interpretation. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 189–201, 2005.

10 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*, pages 238–252. ACM, 1977.

11 Fabien Dagnat and Marc Pantel. Static analysis of communications for erlang. In *Proceedings of 8th International Erlang/OTP User Conference*, 2002.

12 Mads Dam and Lars-Åke Fredlund. On the verification of open distributed systems. In *Proceedings of the 1998 ACM symposium on Applied Computing, SAC'98, Atlanta, GA, USA, February 27 - March 1, 1998*, pages 532–540, 1998.

13 Joeri De Koster, Stefan Marr, Tom Van Cutsem, and Theo D'Hondt. Domains: Sharing state in the communicating event-loop actor model. *Computer Languages, Systems & Structures*, 45:132–160, 2016.

14 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, Amsterdam, The Netherlands, October 30, 2016*, pages 31–40, 2016.

15 Emanuele D'Osualdo. *Verification of Message Passing Concurrent Systems*. PhD thesis, University of Oxford, 2015.

16 Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Soter: an automatic safety verifier for erlang. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, pages 137–140, 2012.

17 Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Automatic verification of erlang-style concurrency. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 454–476, 2013.

18 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121, 2005.

19 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.

20 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. *arXiv preprint arXiv:1611.06276*, 2016.

21 Pierre-Loïc Garoche. *Static Analysis of an Actor-based Process Calculus by Abstract Interpretation*. PhD thesis, National Polytechnic Institute of Toulouse, France, 2008.

22 Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux. Static safety for an actor dedicated process calculus by abstract interpretation. In *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, pages 78–92, 2006.

23 Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 407–420, 2016.

24 Munish K. Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

25 Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, pages 1–6, 2012.

26 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), Paris, France, September 27-29, 1999.*, pages 261–272, 1999.

27 Shams Mahmood Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, pages 67–80, 2014.

28 Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 468–479, 2009.

29 Anders Lindgren. A prototype of a soft type system for erlang. Master's thesis, Uppsala University, 1996.

30 Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, pages 136–149, 1997.

31 Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10, 2012.

32 Matthew Might and Panagiotis Manolios. A posteriorisoundness for non-deterministic abstract interpretations. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, pages 260–274, 2009.

33 Matthew Might and Olin Shivers. Improving flow analyses via gammacfa: abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 13–25, 2006.

34 Matthew Might and David Van Horn. A family of abstract interpretations for static analysis of concurrent higher-order programs. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 180–197, 2011.

35 Doron A. Peled. Ten years of partial order reduction. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 17–28, 1998.

36 Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, pages 339–356, 2006.

37 Quentin Stiévenart, Maarten Vandercammen, Wolfgang De Meuter, and Coen De Roover. Scala-AM: A modular static analysis framework. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 85–90, 2016.

38 David Van Horn and Matthew Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Commun. ACM*, 54(9):101–109, 2011.