Mailbox Abstractions for Static Analysis of Actor Programs (Artifact)

Quentin Stiévenart Jens Nicolay Wolfgang De Meuter Coen De Roover
{qstieven,jnicolay,wdmeuter,cderoove}@vub.ac.be
Software Languages Lab, Vrije Universiteit Brussel, Belgium

The artifact that accompanies the ECOOP paper titled *Mailbox Abstractions for Static Analysis of Actor Programs* is a specific version of SCALA-AM, a static analysis framework implemented in Scala. We describe in this document how to install and to use this artifact (Section 1), and then we explain how to reproduce the experiments conducted in the paper (Section 2).

To facilitate running the commands, all commands listed in this document can be copy-pasted from the following text file: `https://soft.vub.ac.be/~qstieven/ecoop2017/artifact.txt` (also available in the home directory of the VMs).

1 Description of the artifact

1.1 Installation

We describe here the procedure to install our artifact. However, for the sake of simplicity, we also provide `.vdi` images that can be used with VirtualBox, and which contain our artifact as well as all its dependencies[1]. The remainder of this section describes how to install the artifact locally, while Section 1.2 describes how to use the VM image.

Because the artifact is written in Scala, the only dependency required for the installation is `sbt`[2], which will take care of downloading extra dependencies. Once `sbt` is installed, you can download our modified version of SCALA-AM (or access the one provided in the archive) and compile it with `sbt`.

```
# Download it
$ git clone -b ecoop2017actors https://github.com/acieroid/scala-am
# ... or, extract it from the archive
$ unzip artifact.zip; cd artifact/
# Then, compile it
$ cd scala-am
$ sbt
> compile
```

From thereon, you can launch the commands that we provide in the same terminal (commands have to be given in the sbt prompt, indicated by `>`).

1.2 Usage of the VM Image

The password of the virtual machine is `ecoop2017` (the keyboard is configured as qwerty). We recommend to allocate at least 2048MB of RAM for the VM. To use our artifact, go to the `scala-am` directory in a terminal:

[1]A *light* image (terminal only) is available at `https://soft.vub.ac.be/~qstieven/ecoop2017/vm-light.vdi.xz` (972 MB), while a *heavyweight* image (full Ubuntu desktop) is available at `https://soft.vub.ac.be/~qstieven/ecoop2017/vm-full.vdi.xz` (1.8 GB). If you downloaded this document from the DROPS server, the *heavyweight* image is provided in the same archive as this document.

[2]See `sbt`'s manual for instructions on how to install it: `http://www.scala-sbt.org/release/docs/Setup.html`

```
$ cd scala-am
$ sbt
```

From thereon, you can launch the commands that we provide in the same terminal (commands have to be given in the sbt prompt, indicated by `>`).

1.3 Usage

To run the analysis on a benchmark, the following command can be used, where `<file>` denotes the file to analyze, `<mbox>` denotes the mailbox abstraction to use, and `<mbox-bound>` denotes the bound of the mailbox (for bounded mailboxes).

```
> run -m AAMGlobalStore --lang AScheme -f <file>
      -t 60s -l ConstantPropagation --mbox <mbox> --mbox-bound <mbox-bound>
```

The possible values for `<mbox>` are:

- `Powerset` for the powerset abstraction,
- `BoundedList` for the bounded list abstraction,
- `BoundedMultiset` for the bounded multiset abstraction,
- `Graph` for the graph abstraction.

For example, to analyze the `cell` benchmark with a bounded multiset abstraction with bound 3, one would run:

```
> run -m AAMGlobalStore --lang AScheme -f actors/cell.scm
      -t 60s -l ConstantPropagation --mbox BoundedMultiset --mbox-bound 3
```

Information about other parameters can be found by passing `-h` as a parameter.

Note: `sbt` commands (prefixed with a `>` here) have to be typed on a single line, although we represent them on two lines for readability.

2 Evaluation

This section follows the same structure as Section 6 of the paper, and describes for each subsection, how to reproduce the results given in the paper. Note that running times may differ depending on multiple factors, such as the version of Java and the machine used to run the benchmarks.

2.1 Implementation

The implementation analyzes a subset of Scheme that has been extended to support actors constructs (`actor`, `create`, `send`, `become`, `terminate`). This extension is implemented in the file `src/main/scala/semantics/ascheme/ASchemeSemantics.scala` file. Should another researcher want to extend the language analyzed, this is the place where the extensions would be performed.

The machine abstraction that drives the semantics is implemented in the file `src/main/scala/machine/actors/ActorsAAMGlobalStore.scala`. This file contains the implementation of the core components of the analysis: state space, effects, macro-stepping semantics, and the fixpoint computation. This is the file that would be extended in order to modify how the static analysis behaves (e.g., to support a different type of macro-stepping), changing the language being analyzed.

Finally, the different mailbox abstractions presented in the paper are implemented in `src/main/scala/machine/actors/Mbox.scala`. Should other mailbox abstractions be implemented, this is where it would be done.

The rest of the artifact forms a generic static analysis framework, and has been described in a previous paper [1].

2.2 Benchmarks

The benchmarks used in the evaluation are the following:

Benchmark	File
`pp`	`actors/savina/pp.scm`
`count`	`actors/savina/count.scm`
`count-seq`	`actors/savina/count-seq.scm`
`fjt-seq`	`actors/savina/fjt-seq.scm`
`fjc-seq`	`actors/savina/fjc-seq.scm`
`factorial`	`actors/factorial.scm`
`stack`	`actors/stack.scm`
`cell`	`actors/cell.scm`
`parikh`	`actors/soter/parikh.scm`
`pipe-seq`	`actors/soter/pipe-seq.scm`
`unsafe-send`	`actors/soter/unsafe_send.scm`
`safe-send`	`actors/soter/safe_send.scm`
`state-factory`	`actors/soter/state_factory.scm`
`stutter`	`actors/soter/stutter.scm`

2.3 Running time and flow graph size

2.3.1 Scala-AM, automated

Table 2 of Section 6.3 of the paper lists running time in milliseconds, and the number of nodes in the produced graphs. The process to generate the numbers in this table is automated and can be run with the following sbt command:

```
> run-main ArtifactEvaluation timesize
```

(It will take around 30 minutes to run to completion)

This will run all the benchmarks 12 times. The results are given in the same format as Table 2. Timeouts are indicated by a negative number. The output will look as follows, where we manually aligned columns. Each cell includes the number of states generated, and the time taken in milliseconds. The bounds of bounded mailbox abstractions used are the one listed in Table 2.

```
benchmark       | powerset    | multiset  | list      | graph     |
pp              | 21, 134     | 8, 4      | 8, 4      | 8, 3      |
count           | 83, 306     | 22, 25    | 21, 26    | 22, 19    |
count-seq       | 45, 56      | 10, 1     | 8, 1      | 8, 1      |
fjt-seq         | 201, 1304   | 589, 3583 | 589, 3457 | 589, 2572 |
fjc-seq         | 15, 5       | 15, 3     | 15, 2     | 15, 2     |
factorial       | 1490, 31510 | 46, 596   | 52, 666   | 22, 48    |
stack           | 85, 225     | 42, 10    | 16, 2     | 16, 2     |
cell            | 70, 99      | 23, 3     | 15, 1     | 15, 1     |
parikh          | 31, 14      | 8, 0      | 8, 0      | 8, 0      |
pipe-seq        | 2952, 49384 | 24, 13    | 24, 13    | 24, 13    |
unsafe-send     | 4, 0        | 3, 0      | 3, 0      | 3, 0      |
safe-send       | 100, 89     | 32, 6     | 28, 3     | 30, 4     |
state-factory   | 76, 146     | 43, 65    | 160, 218  | 214, 246  |
stutter         | 28, 19      | 60, 28    | 34, 19    | 15, 3     |
```

2.3.2 Scala-AM, manual

The size of the graph and the running time is also given each time an analysis is run, e.g.:

```
> run -m AAMGlobalStore --lang AScheme -f actors/cell.scm
      -t 60s -l ConstantPropagation --mbox BoundedMultiset --mbox-bound 3
```

```
...
Visited 23 states in 0.23487492 seconds, 1 possible results: Set(cell@9.22)
...
```

Note: the value of n for the bounded multiset and bounded list abstractions has been chosen manually as explained in the paper (we increased the bound until maximal precision was reached). The values for n used in our experiments are listen in Table 2 of the paper.

Note: running times may differ depending on the version of Java and the machine used. However, the overall picture of differences in timing between benchmarks and mailbox abstractions should remain similar.

2.4 Precision

Figure 8 of Section 6.4 of the paper was produced by listing the mailboxes and messages dequeued from each mailbox in each benchmark, both for a concrete run and for an abstract run with a given mailbox abstractions. We then manually compared these mailboxes and messages, and counted the number of spurious mailboxes and messages, as explained in the paper.

While we could not automate this task, as it is relatively complex, we describe how one proceeds to produce Figure 8 on the `count-seq` benchmark. First, one has to know which are the concrete values for mailboxes and messages that can appear in the execution of the program. In some cases, these are not computable (e.g., the `stutter` benchmarks produces unbounded mailboxes), and require a human to be able to know whether a mailbox can arise in a concrete run or not. On the `count-seq` benchmark, we can generate them with the following command (we use a bounded list with bound 3, which is sufficient to achieve maximal precision).

```
> run -m AAMGlobalStore --lang AScheme -f actors/savina/count-seq.scm
      -t 60s -l Concrete -c --mbox BoundedList --mbox-bound 3
...
Dequeued per behavior and mailbox:
counting-actor@21.27, (actor counting ...), 1, retrieve(producer-actor@22.28)
retrieve(producer-actor@22.28)
counting-actor@21.27, (actor counting ...), 2, increment, retrieve(producer-actor@22.28)
increment()
counting-actor@21.27, (actor counting ...), 1, increment
increment()
producer-actor@22.28, (actor producer ...), 1, increment
increment()
producer-actor@22.28, (actor producer ...), 1, result({1})
result({1})
Visited 8 states in 0.098189418 seconds, 1 possible results: Set(producer-actor@22.28)
...
```

We extracted only the relevant part of the output. After the *Dequeued per behavior and mailbox:* line, the output shows the mailboxes that each actor can have, and which messages can be dequeued for each of these mailboxes. We see for example that the actor `counting-actor` can have the concrete mailbox [`retrieve(...)`], and that from this mailbox, the message `retrieve(...)` can be dequeued.

Once we know what the possible concrete mailboxes are, we run the analysis with a given mailbox abstraction (here with the powerset abstraction). Note that the order of mailboxes in the output might differ (we reordered it to improve clarity).

```
> run -m AAMGlobalStore --lang AScheme -f actors/savina/count-seq.scm
      -t 60s -l ConstantPropagation --mbox Powerset
...
counting-actor@21.27, (actor counting ...), +, retrieve(producer-actor@22.28)
retrieve(producer-actor@22.28)
counting-actor@21.27, (actor counting ...), +, increment + retrieve(producer-actor@22.28)
```

```
increment(), retrieve(producer-actor@22.28)
counting-actor@21.27, (actor counting ...), +, increment
increment()
producer-actor@22.28, (actor producer ...), +, increment + result({0})
increment(), result({0})
producer-actor@22.28, (actor producer ...), +, increment
increment()
producer-actor@22.28, (actor producer ...), +, result({Int})
result({Int})
producer-actor@22.28, (actor producer ...), +, increment + result({Int})
increment(), result({Int})
producer-actor@22.28, (actor producer ...), +, result({0})
result({0})
Visited 45 states in 0.275468673 seconds, 1 possible results: Set(producer-actor@22.28)
...
```

We then have to compare the concrete and abstract results, and to determine:

- Which concrete mailbox correspond to which abstract mailbox. If an abstract mailbox corresponds to no concrete mailbox, this counts as a *spurious mailbox*.
- For each spurious mailbox, the messages dequeued are *spurious messages dequeued from spurious mailboxes*.
- Which message dequeued from a concrete mailbox correspond to which message dequeued from the corresponding abstract mailbox. If a message dequeued from an abstract mailbox does not correspond to the message dequeued from the corresponding concrete mailbox, this counts as a *spurious message dequeued from non-spurious mailboxes*.

We summarize this information in the following table, for the `count-seq` benchmark (we meaningfully truncate message names for space reasons). Take line 2 as an example: the concrete mailbox is the sequence `[inc,ret]`, from which only message `inc` can be dequeued. The corresponding abstract mailbox is `{inc,ret}`, from which both `inc` and `ret` can be dequeued. The `ret` message is therefore a *spurious message dequeued from a non-spurious mailbox*. As an other example, take the last line: the abstract mailbox `{res(0)}` has no corresponding concrete mailbox: the only possible value of n for a message `res(n)` in a concrete run is 1. This is therefore a *spurious mailbox*. One message can be dequeued from that mailbox, and it is therefore a *spurious message dequeued from a spurious mailbox*. The **Spurious** column lists in order: the number of spurious mailboxes, the number of spurious messages dequeued from spurious mailboxes, and the number of spurious messages dequeued from non-spurious mailboxes.

Actor	**Concrete**	**Dequeued**	**Abstract**	**Dequeued**	**Spurious**
`counting-actor`	`[ret]`	`ret`	`{ret}`	`ret`	0/0/0
	`[inc,ret]`	`inc`	`{inc,ret}`	`inc`, `ret`	0/0/1
	`[inc]`	`inc`	`{inc}`	`inc`	0/0/0
`producer-actor`	`[inc]`	`inc`	`{inc}`	`inc`	0/0/0
	`[res(1)]`	`res(1)`	`{res(Int)}`	`res(Int)`	0/0/0
	–	–	`{inc,res(0)}`	`inc,res(0)`	1/2/0
	–	–	`{inc,res(Int)}`	`inc,res(Int)`	1/2/0
	–	–	`{res(0)}`	`res(0)`	1/1/0
Total					3/5/1

This process can be repeated for all benchmarks, and the individual results are given in the table below. The *Total* line is the sum of each column, and this is the information reported in Figure 8 of the paper.

Bounds: the mailbox bounds used to measure precision are the same bounds as used in Table 2 of the paper (i.e., the bounds that provide maximal precision for each abstraction).

Timing: we ran each benchmark until completion to assess the precision of the full output of the analysis, hence the timeout of 60s was not used to produce Figure 8.

Benchmark	*PS*			MS_n			L_n			*G*		
	A	B	C	A	B	C	A	B	C	A	B	C
`pp`	5	11	0	0	0	0	0	0	0	0	0	0
`count`	3	5	1	2	2	1	1	1	1	0	0	0
`count-seq`	3	5	1	1	1	1	0	0	0	0	0	0
`fjt-seq`	0	0	0	0	0	0	0	0	0	0	0	0
`fjc-seq`	0	0	0	0	0	0	0	0	0	0	0	0
`factorial`	7	14	0	5	7	0	5	6	0	5	5	0
`stack`	4	9	10	6	13	10	0	0	0	0	0	0
`cell`	0	0	1	0	0	1	0	0	0	0	0	0
`parikh`	4	7	1	0	0	1	0	0	0	0	0	0
`pipe-seq`	16	30	0	4	4	0	4	4	0	4	4	0
`unsafe-send`	0	0	0	0	0	0	0	0	0	0	0	0
`safe-send`	21	61	11	3	7	11	0	0	0	0	0	0
`state-factory`	2	3	0	0	0	0	3	4	0	6	6	0
`stutter`	2	2	2	4	7	2	4	4	2	0	0	0
Total	67	147	27	25	41	27	17	19	3	15	15	0

Table 1: Precision metrics for the different mailbox abstractions. Column *A* indicates the number of spurious mailboxes that correspond to no concrete mailbox in a concrete run of the benchmark. Column *B* indicates the number of spurious values dequeued from spurious mailboxes. Column *C* indicates the number of spurious values dequeued from non-spurious mailboxes.

2.5 Comparison with Soter

Table 3 of Section 6.5 compares the result of our analysis with Soter, another static analysis tool for actor programs. Because Soter analyzes Erlang programs, while our analysis analyzes programs written in an extended version of Scheme, we had to translate the benchmarks faithfully in Erlang. The translation of each benchmark is located at the same place as the Scheme benchmark, and has a `.erl` suffix. For example, the `count-seq` is located in `actors/savina/count-seq.erl`.

2.6 Verification of absence of errors

2.6.1 Scala-AM, automated

We automated the process for the construction of the first part of Table 3, and one can run the following command to produce a table listing, for each abstraction (in order: powerset, bounded multiset, bounded list, graph) how many errors are detected as reachable. A count of 0 in a cell indicates that the verification of the property succeeded.

```
> run-main ArtifactEvaluation errors
```

(It will take around 10 seconds to run to completion)

The output looks as follows. This means for example that, for the `parikh` benchmark, the powerset abstraction detects one possible error (a false positive), while the other abstractions successfully detect no errors.

```
benchmark | powerset | multiset | list | graph |
parikh          | 1 | 0 | 0 | 0 |
unsafe-send     | 1 | 1 | 1 | 1 |
safe-send       | 3 | 2 | 0 | 0 |
stutter         | 1 | 1 | 1 | 0 |
stack           | 1 | 1 | 0 | 0 |
count-seq       | 1 | 1 | 0 | 0 |
cell            | 1 | 1 | 0 | 0 |
```

Relation with Table 3: Table 3 of the paper states whether each benchmark can be analyzed with full precision (i.e., detecting no errors when the benchmark is free of errors, and detecting exactly the same errors as the ones that are expected if it is not the case), and lists the abstraction for which full precision was achieved. The graph abstraction always has full precision in the case of these benchmarks (all benchmarks have no errors except `unsafe-send` which has one error).

Timing: The automated technique here only runs the benchmarks once. Timing information of Table 3 is retrieved from Table 2 (verification of errors also happens when computing the information of Table 2), where the time is taken by averaging 10 runs of the benchmarks after 2 warm-up runs.

Bounds: The bounds used for each abstraction and benchmark are the same bound as the ones given in Table 2.

2.6.2 Scala-AM, manual

To verify absence of errors, we run our analysis on each benchmark as follows (here for the `parikh` benchmark). The last part of the output lists the errors that may be reachable, or states whether no errors are reachable (in which case, the verification succeeded).

Here is the output with the powerset abstraction, where the analysis cannot prove the property:

```
> run -m AAMGlobalStore --lang AScheme -f actors/soter/parikh.scm
      -t 60s -l ConstantPropagation --mbox Powerset
[info] Running Main -m AAMGlobalStore --lang AScheme -f actors/soter/parikh.scm
                    -t 60s -l ConstantPropagation --mbox Powerset
...
Visited 31 states in 0.170430096 seconds, 1 possible results: Set(server-init-actor@20.21)
One error is reachable:
UserError({We should be already initialized!},7.38)
```

And here is the output with the graph abstract, where the analysis can prove that no errors are reachable:

```
> run -m AAMGlobalStore --lang AScheme -f actors/soter/parikh.scm
      -t 60s -l ConstantPropagation --mbox Graph
[info] Running Main -m AAMGlobalStore --lang AScheme -f actors/soter/parikh.scm
                    -t 60s -l ConstantPropagation --mbox Graph
...
Visited 8 states in 0.077454421 seconds, 1 possible results: Set(server-init-actor@20.21)
No error reachable!
```

This process is repeated for every benchmark and abstraction, with the bounds given in Table 3 of the paper. Again, these bounds have been selected manually by running the analysis with a bound of 1, and increasing the bound until the analysis can prove the property.

The timing information of Table 3 is the information used in Table 2 (see Section 2.3).

2.6.3 Soter

To run Soter on each benchmark for verifying absence of errors, one has to use the web interface of Soter[3]. On the left part of the user interface, one has to paste the Erlang version of the benchmark, select *Verify All* as the *Mode*, and click *Run!*. If the property can be verified, the right part of the user interface reports *The program is Safe!*. Otherwise, it reports *The program could not be proved safe!*. The *Stats* tab gives timing information.

2.7 Verification of mailbox bounds

We verify the following bounds on the benchmarks:

[3] `https://mjolnir.cs.ox.ac.uk/soter/`

Benchmark	Actor	Bound
`pipe-seq`	`pipe-node`	1
`state-factory`	`state-actor`	1
`pp`	`ping-actor`	1
	`pong-actor`	1
`count-seq`	`producer-actor`	1
	`counting-actor`	2
`cell`	`cell`	2
	`display-actor`	1
`fjc-seq`	`forkjoin-actor`	1
`fjt-seq`	`throughput-actor`	10

2.7.1 Scala-AM, automated

We automated the processus to generate the second part of Table 3: you can run it using the following command.

```
> run-main ArtifactEvaluation bounds
```

(It will take around 2 minutes to run to completion).

This will output a table, listing for each benchmark and each mailbox abstraction (in order: powerset, bounded multiset, bounded list, graph) whether the mailbox bounds could be verified (`v`) or not (`x`). Again, the bounds for the bounded abstractions (multiset and list) have been selected manually.

2.7.2 Scala-AM, manual

The output of the analysis lists the bound on the mailbox of each actor, and uses `+` to indicate that it could not deduce a bound for a mailbox. For example, if we run the analysis on `state-factory` with a suitable abstraction, we see that both the actors `ping-actor` and `pong-actor` have a mailbox size bounded by 1.

```
> run -m AAMGlobalStore --lang AScheme -f actors/savina/pp.scm
      -t 60s -l ConstantPropagation --mbox BoundedMultiset --mbox-bound 1
[info] Running Main -m AAMGlobalStore --lang AScheme -f actors/savina/pp.scm
                    -t 60s -l ConstantPropagation --mbox BoundedMultiset --mbox-bound 1
...
Bounds:
ping-actor@26.24: 1
pong-actor@24.24: 1
...
```

On the other hand, with an unsuitable abstraction (here, the powerset abstraction), the analysis is inconclusive, as indicated by the `+` signs.

```
> run -m AAMGlobalStore --lang AScheme -f actors/savina/pp.scm
      -t 60s -l ConstantPropagation --mbox Powerset
[info] Running Main -m AAMGlobalStore --lang AScheme -f actors/savina/pp.scm
                    -t 60s -l ConstantPropagation --mbox Powerset --mbox-bound 1
...
Bounds:
ping-actor@26.24: +
pong-actor@24.24: +
...
```

2.7.3 Soter

To run Soter on each benchmark for verifying mailbox bounds, the process is similar to the verification of absence of errors. Each benchmark code is annotated with the properties to verify, and Soter will output

The program is Safe! when the property is verified, otherwise it outputs *The program could not be proved safe!*.

Note: The `fjt-seq` benchmark is a special case. Our analysis is unable to prove the bound on that benchmark, while Soter is "able" to prove it, but also to prove *lower* bounds, meaning that it produces unsound results. This is why this benchmark is earmarked in the paper. Our technique cannot prove this benchmark, but one cannot rely on Soter results for this benchmark, because they are unsound.

2.8 Soundness

The soundness proofs are given in an accompanying technical report[4]. We also mechanized some of these proofs[5] in Coq. The Coq proof script can be run using `coqc` (Coq version 8.6 is required). The theorems proved will be printed on the screen. In the table below, we show the correspondance between the theorems proved in the Coq script and in the technical report (not all lemmas and theorems are mechanized, as explained in the technical report).

To run the proof script, you need Coq installed, and you should run the following commands. Coq is already installed in the provided VM (skip the first command if you use the VM).

```
# Download proof script
$ git clone https://github.com/acieroid/mailbox-abstraction-proofs
# ... or use the version given in the archive
$ unzip artifact.zip; cd artifact/
# Then, run the proofs
$ cd mailbox-abstraction-proofs
$ coqc proofs
```

Theorem/Lemma	**Name**
Lemma 1	`SetMbox_empty_sound`
Lemma 2	`SetMbox_enq_sound`
Lemma 3	`SetMbox_deq_sound`
Lemma 4	`SetMbox_size_sound`
Theorem 1	`SetMbox_sound`
Lemma 5	`BListMbox_empty_sound`
Lemma 6	`BListMbox_enq_sound`
Lemma 7	`BListMbox_deq_sound_overapprox`
Lemma 8	`BListMbox_size_sound`
Theorem 2	`BListMbox_sound`
Lemma 9	`MultiSetMbox_empty_sound`
Lemma 10	`MultiSetMbox_enq_sound`
Lemma 11	`MultiSetMbox_deq_sound`
Lemma 13	`BMultiSetMbox_empty_sound`
Lemma 14	`BMultiSetMbox_enq_sound`
Lemma 15	`BMultiSetMbox_deq_sound`
Lemma 17	`Graph_empty_sound`
Lemma 18	`Graph_enq_sound`

References

[1] 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.

[4] `https://soft.vub.ac.be/~qstieven/ecoop2017/techreport.pdf`
[5] `https://github.com/acieroid/mailbox-abstraction-proofs`