Soter: An Automatic Safety Verifier for Erlang

Emanuele D'Osualdo, Jonathan Kochems and Luke Ong

Department of Computer Science
University of Oxford

21 October 2012
AGERE!, Tucson, AZ

Example: Erathostene's sieve


```
1  counter(N) →
2      receive {poke, From} →
3          From!{ans, N}, counter(N+1)
4      end.
5
6  sieve(In, Out) →
7      In!{poke, self()},
8      receive {ans,X} →
9          Out!{prime,X},
10         F = spawn(fun()→
11                 filter(divisible_by(X), In)
12             end),
13         sieve(F,Out)
14     end.
```

Rob Pike.
Concurrency and message passing in Newsqueak.
Google Tech Talks, 2007.

Example: Erathostene's sieve


```
15  filter(Test, In) →
16      receive {poke, From} →
17          filter(Test, In, From)
18      end.
19
20  filter(Test, In, Out) →
21      In!{poke, self()},
22      receive {ans,Y} →
23          case Test(Y) of
24              false →Out!{ans,Y}, filter(Test, In);
25              true  → filter(Test, In, Out)
26          end
27      end.
```

Example: Erathostene's sieve


```
28  main() →
29      Me = self(),
30      Gen = spawn(fun()→ counter(2) end),
31      spawn(fun()→ sieve(Gen,Me) end),
32      dump().
33
34  dump() →
35      receive
36          {prime, X} →io:write(X),
37                      dump()
38      end.
```

Erlang

Erlang:

- Pure functional sequential fragment
- Call-by-value
- Dynamically typed
- Higher Order
- Process creation: `spawn` / `self`
- Message passing: `receive` / send `P ! Msg`
- Send is asynchronous, receive is blocking

Jim Larson.
Erlang for concurrent programming.
Communications of the ACM, 2009.

Specifications

Assertions:

- Local:

  `?assert_uncoverable(NUM)`

  `?soter_error(ERRMSG)`

Specifications

Assertions:

- Local:

  `?assert_uncoverable(NUM)`

  `?soter_error(ERRMSG)`

- Global:
  - Labels

    `?label(ID)`

    `?label_mail(ID)`

Specifications

Assertions:

- Local:

  `?assert_uncoverable(NUM)`

  `?soter_error(ERRMSG)`

- Global:
  - Labels

    `?label(ID)`

    `?label_mail(ID)`

  - Properties

    `-uncoverable(PROP).`

    $\texttt{PROP} ::= p_1, p_2, \ldots, p_n$

    $p ::= \texttt{ID} \geq k$

Specifications

Assertions:

- Local:

  `?assert_uncoverable(NUM)`

  `?soter_error(ERRMSG)`

- Global:
  - Labels

    `?label(ID)`

    `?label_mail(ID)`

  - Properties

    `-uncoverable(PROP).`

    $\texttt{PROP} ::= p_1, p_2, \ldots, p_n$

    $p ::= \texttt{ID} \geq k \mid \texttt{ID} > k \mid \texttt{ID}$

Specifications

Non-deterministic choice:

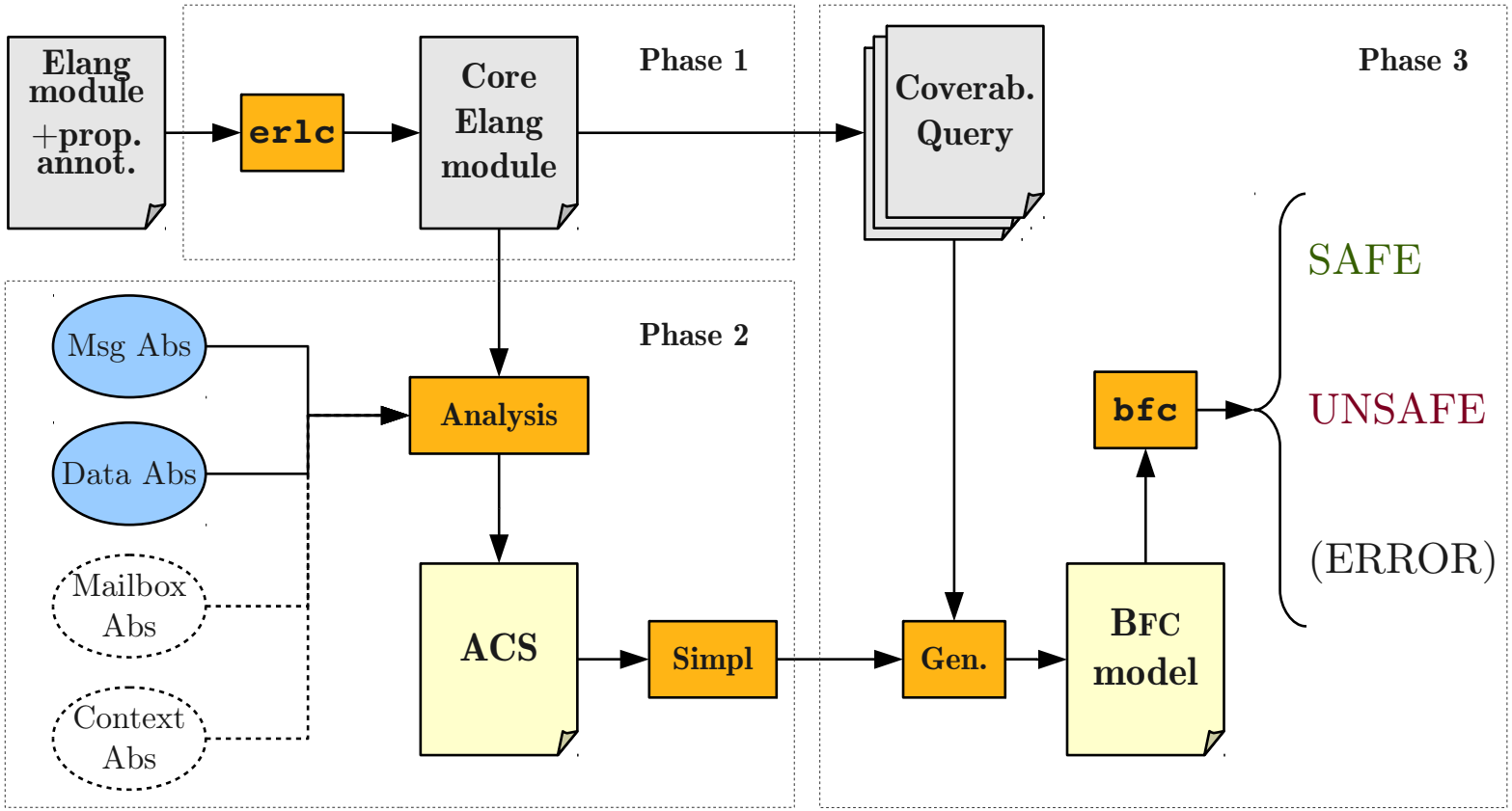
- `?any_nat()`
- `?any_bool()`
- `?SoterChoice(A,B,C)`
- ...

Demonstration

Demo:

http://mjolnir.cs.ox.ac.uk/soter

Soter


Elang module +prop. annot.
erlc
Core Elang module
Phase 1
Coverab. Query
Phase 3
Msg Abs
Data Abs
Mailbox Abs
Context Abs
Analysis
Phase 2
ACS
Simpl
Gen.
Bfc model
bfc
SAFE
UNSAFE
(ERROR)

Verification pathway

Our method:

- (Core) Erlang code as source
- A k-CFA-like analysis abstracts the control-flow
- The analysis produces an ACS which soundly approximates the program
- Model-check the ACS with VAS coverability checker (BFC)

The analysis is parametric and can be tuned for accuracy.

Actor Communicating Systems

We define an abstract model:

- finite set of *control states* q
- finite set of *pid-classes* ι
- finite set of *messages* $\mathtt{m}$
- finite set of rules:

$$\textbf{Seq. red.} \qquad \iota : q \xrightarrow{\tau} q'$$

$$\textbf{Send} \qquad \iota : q \xrightarrow{\iota'!\mathtt{m}} q'$$

$$\textbf{Receive} \qquad \iota : q \xrightarrow{?\mathtt{m}} q'$$

$$\textbf{Spawn} \qquad \iota : q \xrightarrow{\nu\iota'.q'} q''$$

Challenges

Sources of infinity in the state space:

- recursive function definitions
- infinite domains of values
- message space is infinite
- higher order
- unbounded dynamic processes creation
- mailboxes (unbounded capacity, FIFFO)

Challenges

Sources of infinity in the state space: **Analysis**

- recursive function definitions **finite/regular abs**
- infinite domains of values **regular abs**
- message space is infinite **regular abs**
- higher order **regular abs**
- unbounded dynamic processes creation **finite set abs**
- mailboxes (unbounded capacity, FIFFO) **finite set abs**

Challenges

Sources of infinity in the state space:	**ACS**
recursive function definitions	**finite/regular abs**
infinite domains of values	**finite abs**
message space is infinite	**finite abs**
higher order	**regular abs**
unbounded dynamic processes creation	**counter abs**
mailboxes (unbounded capacity, FIFFO)	**counter abs**

Erlang mailboxes


```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

Q [z, b, a, a]

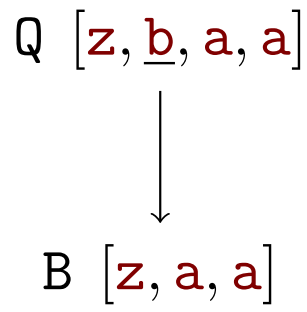
Erlang mailboxes


```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

Q [z, b, a, a]

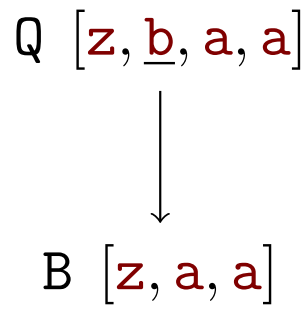
Erlang mailboxes


```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```



Erlang mailboxes


```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```



First In First Fireable Out (FIFFO) mailbox + finite control

=

Turing powerful.

Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

Q [z, b, a, a]

B [z, a, a]

ACS

ι : Q $\xrightarrow{?a}$ A

ι : Q $\xrightarrow{?b}$ B

ι : Q $\xrightarrow{?c}$ C

Q [z, b, a, a]

Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

Q [z, b, a, a]

B [z, a, a]

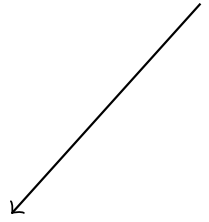
ACS

ι : Q $\xrightarrow{?a}$ A

ι : Q $\xrightarrow{?b}$ B

ι : Q $\xrightarrow{?c}$ C

Q [z, b, a, a]

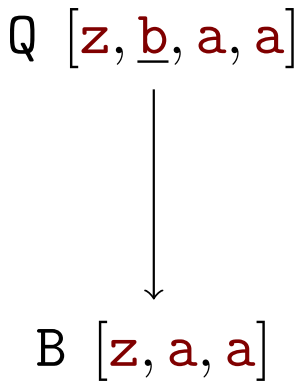


B [z, a, a]

Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

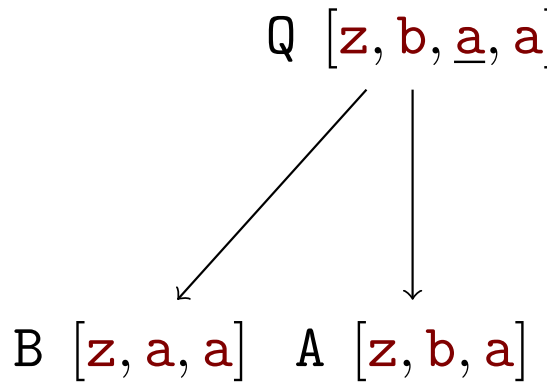


ACS

ι : Q $\xrightarrow{?a}$ A

ι : Q $\xrightarrow{?b}$ B

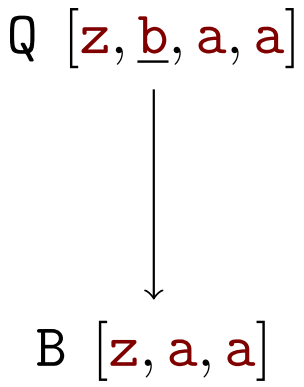
ι : Q $\xrightarrow{?c}$ C



Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

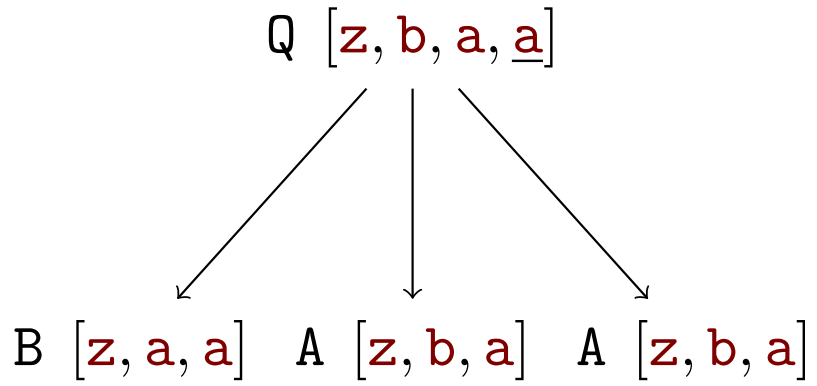


ACS

ι : Q $\xrightarrow{?a}$ A

ι : Q $\xrightarrow{?b}$ B

ι : Q $\xrightarrow{?c}$ C

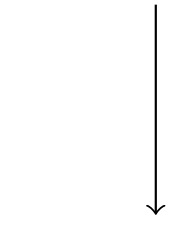


Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

Q [z, b, a, a]



B [z, a, a]

ACS

ι : Q $\xrightarrow{?a}$ A

ι : Q $\xrightarrow{?b}$ B

ι : Q $\xrightarrow{?c}$ C

Q

a	b	c	z
2	1	0	1

B

a	b	c	z
2	0	0	1

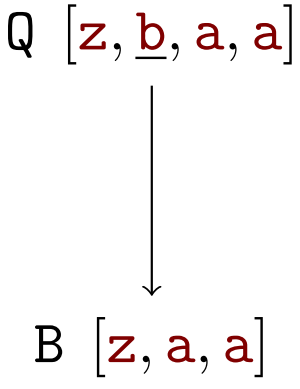
A

a	b	c	z
1	1	0	1

Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

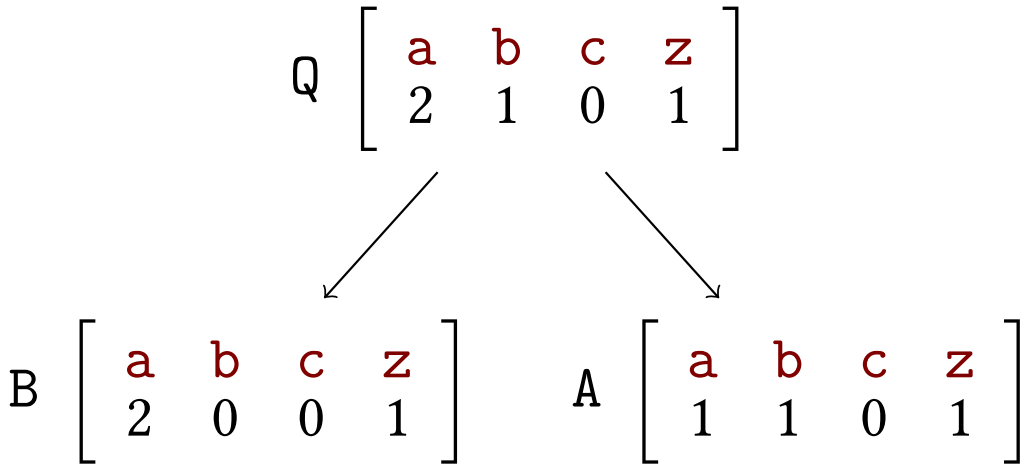


ACS

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{a}} \mathtt{A}$$

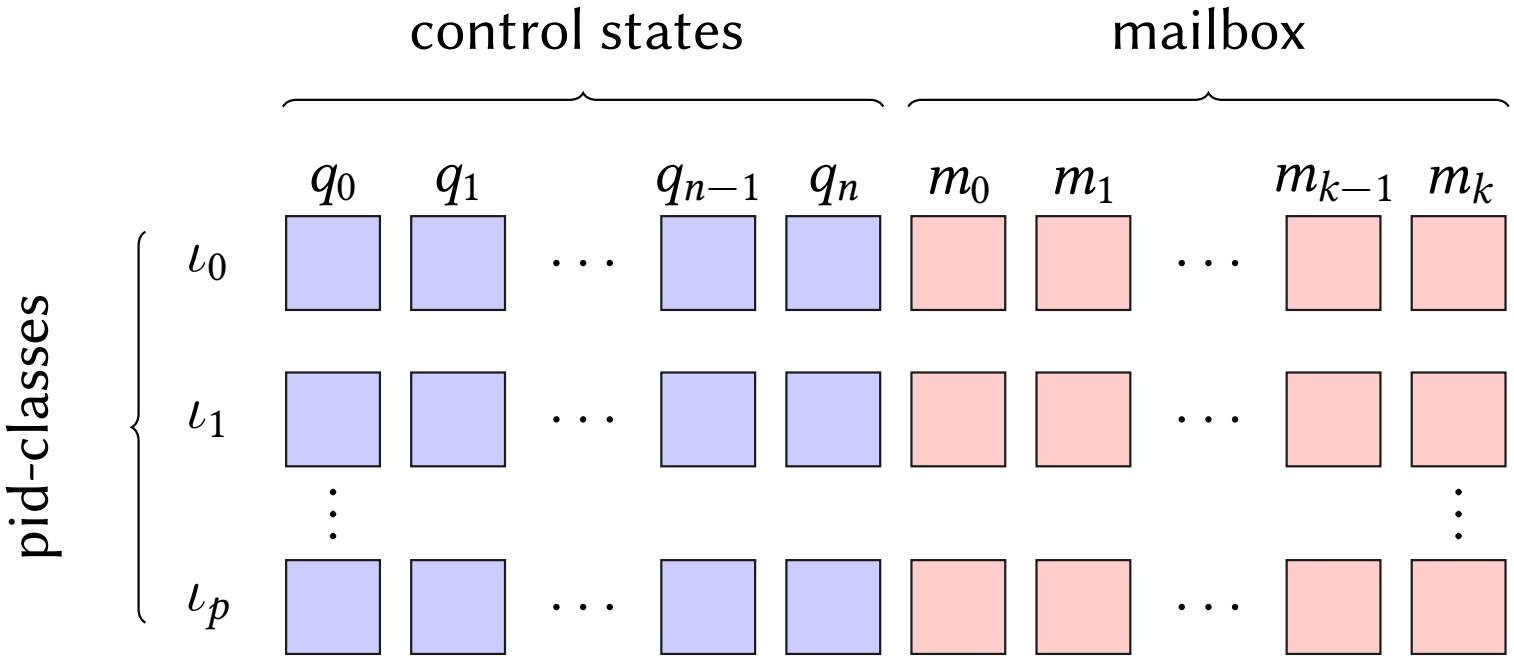
$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{b}} \mathtt{B}$$

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{c}} \mathtt{C}$$

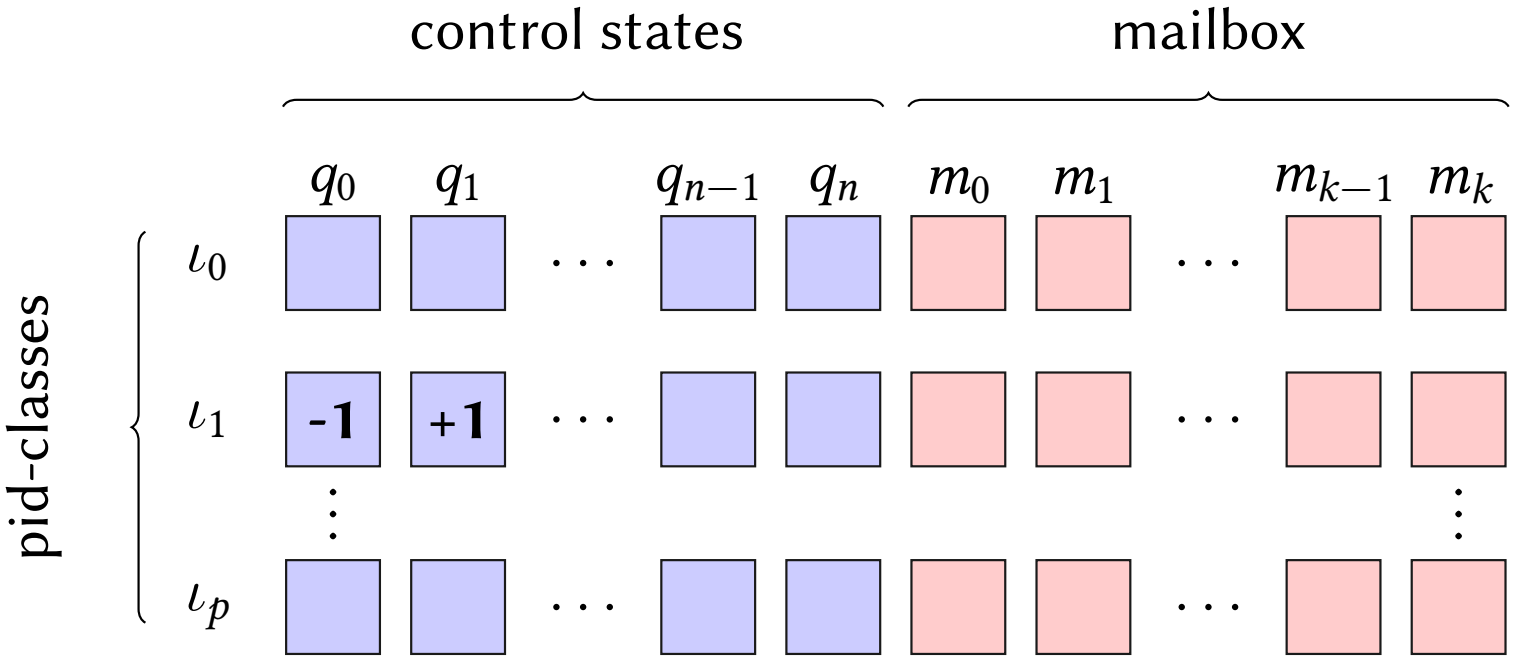


Parikh abstraction on ACS mailboxes.

Underlying VAS

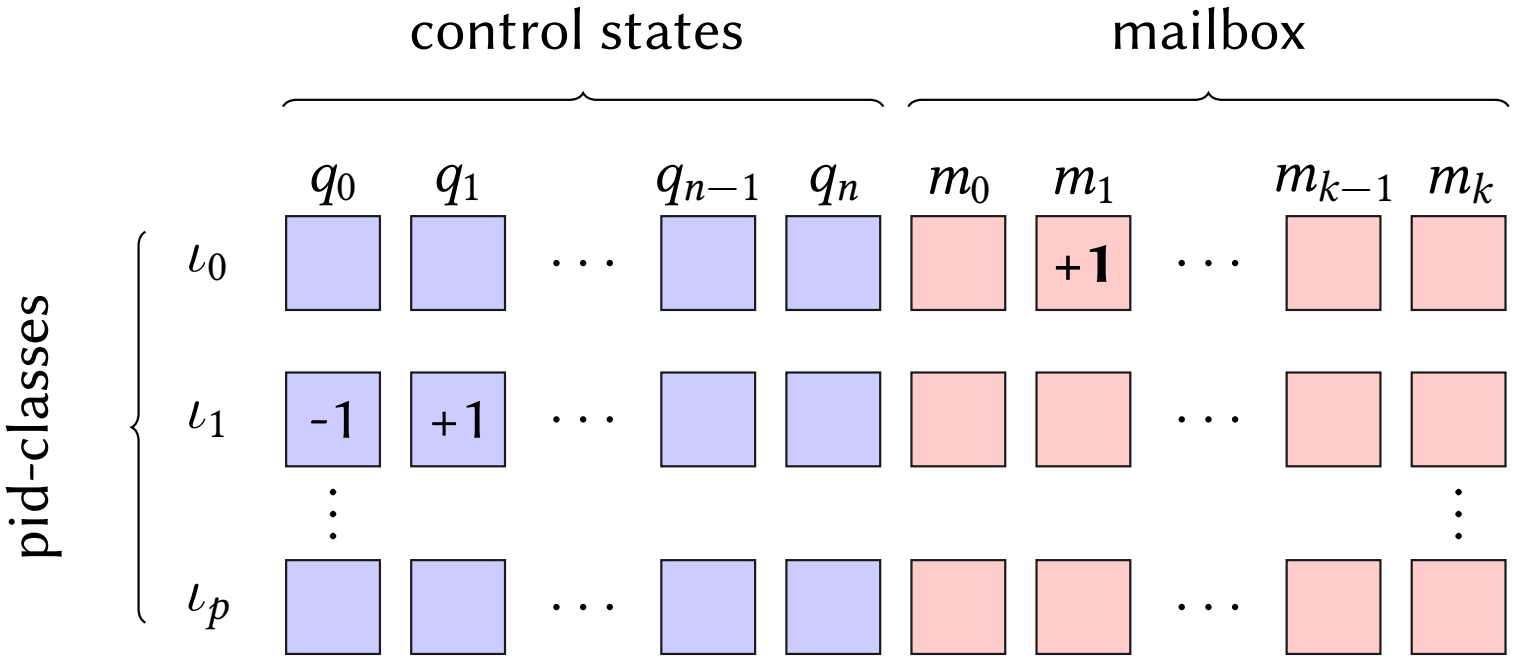


Underlying VAS

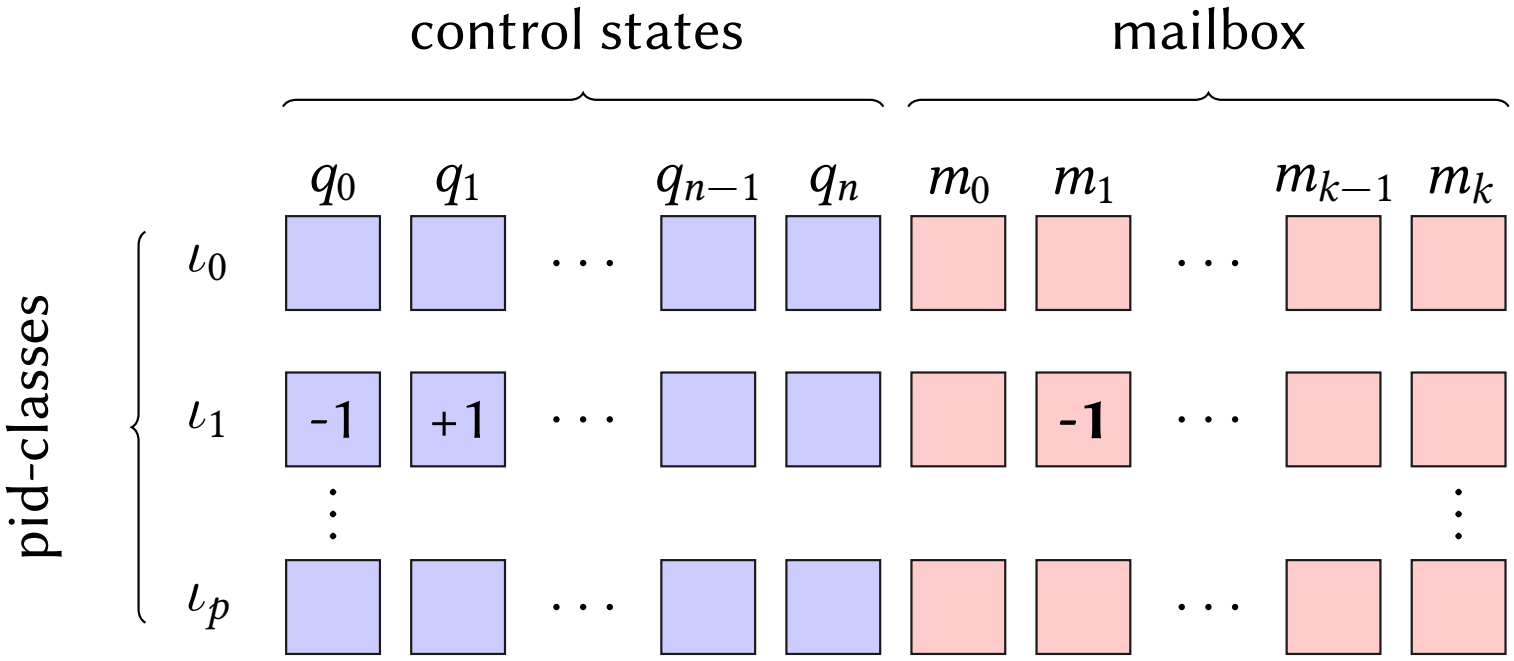


Seq. reduction $\iota_1 : q_0 \xrightarrow{\tau} q_1$

Underlying VAS

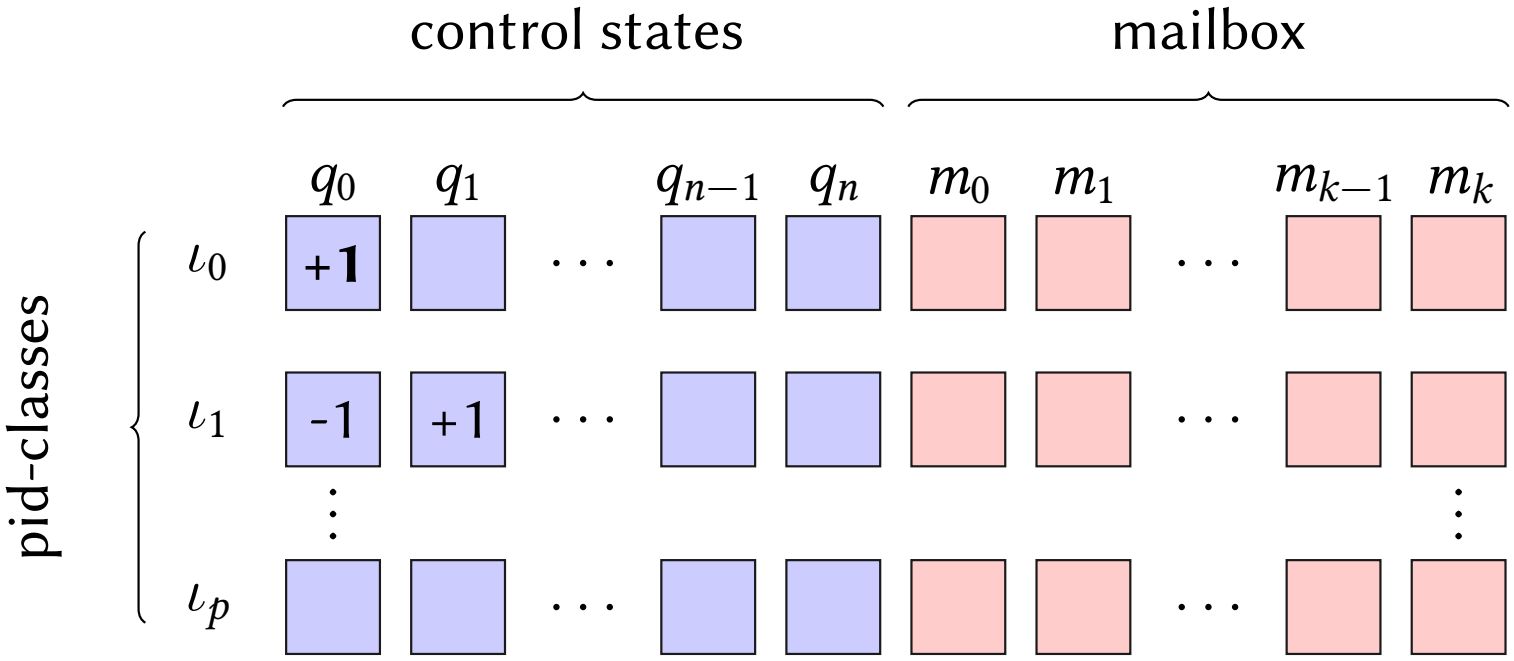


Underlying VAS



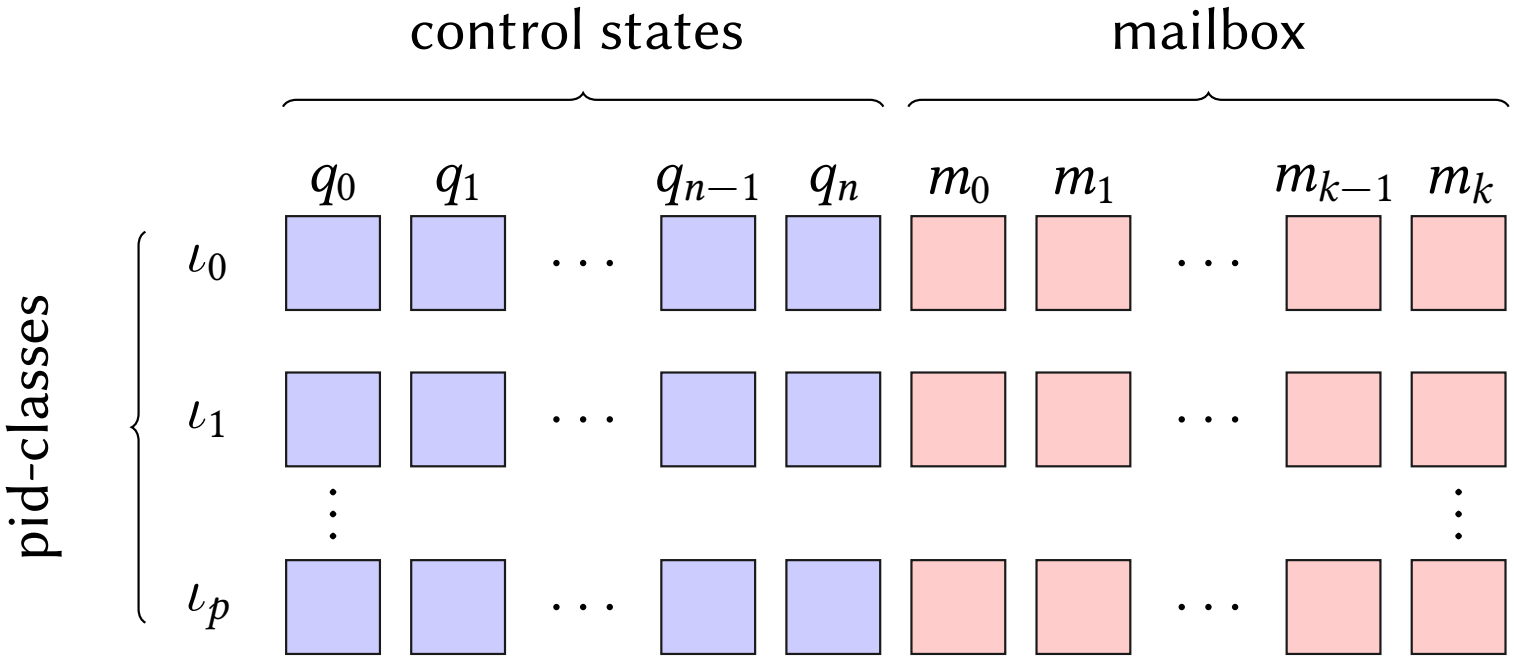
Receive $\iota_1 : q_0 \xrightarrow{?m_1} q_1$

Underlying VAS



Spawn $\quad \iota_1 : q_0 \xrightarrow{\nu \iota_0 . q_0} q_1$

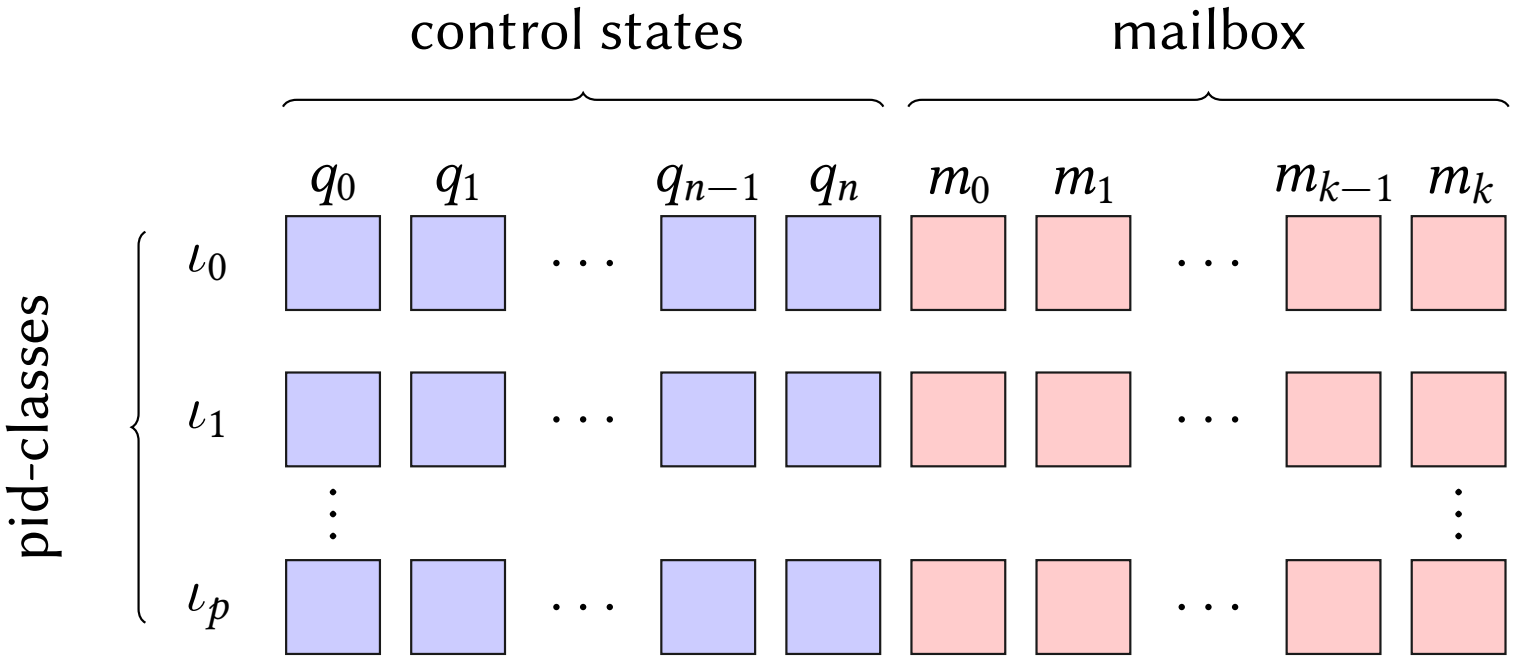
Underlying VAS



Encoding properties:

Mutual Exclusion $\mathbf{v}(\iota_1, q_1) < 2$

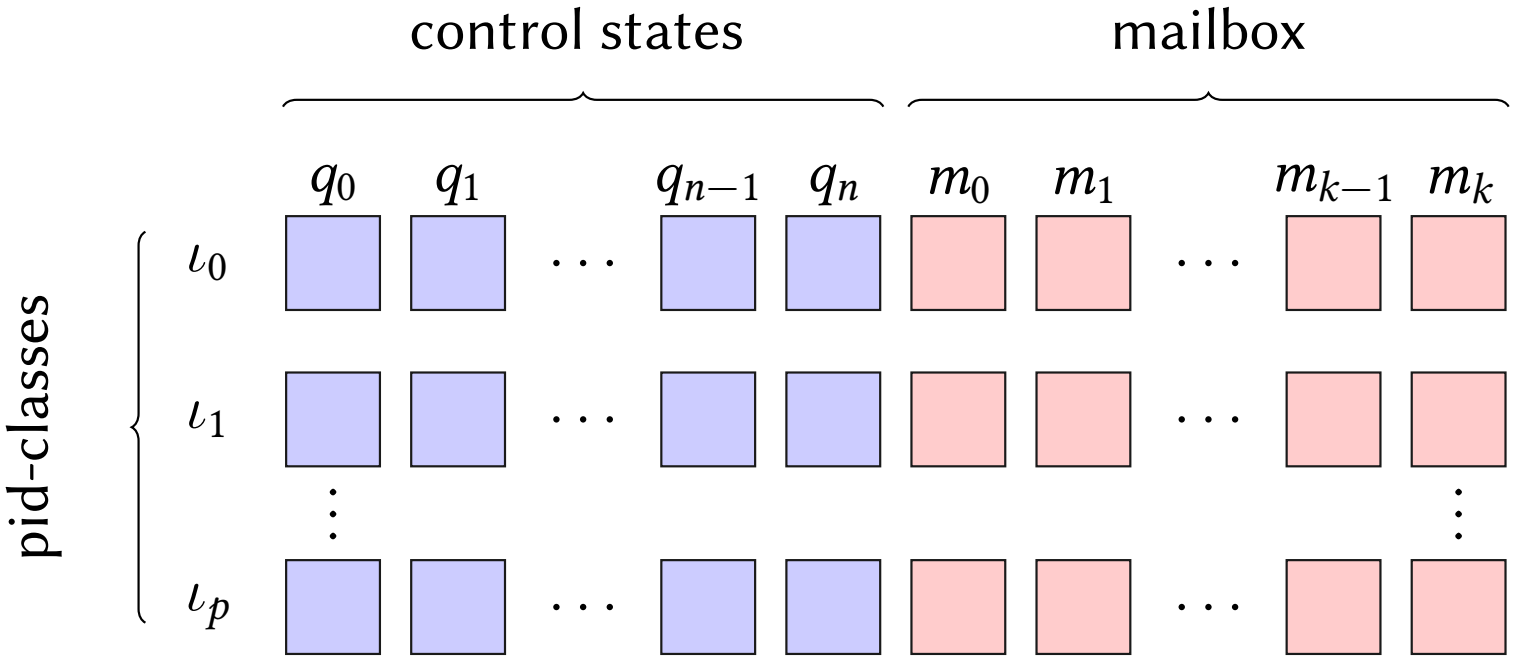
Underlying VAS



Encoding properties:

Absence of error $\mathbf{v}(\iota_1, q_1) = 0$

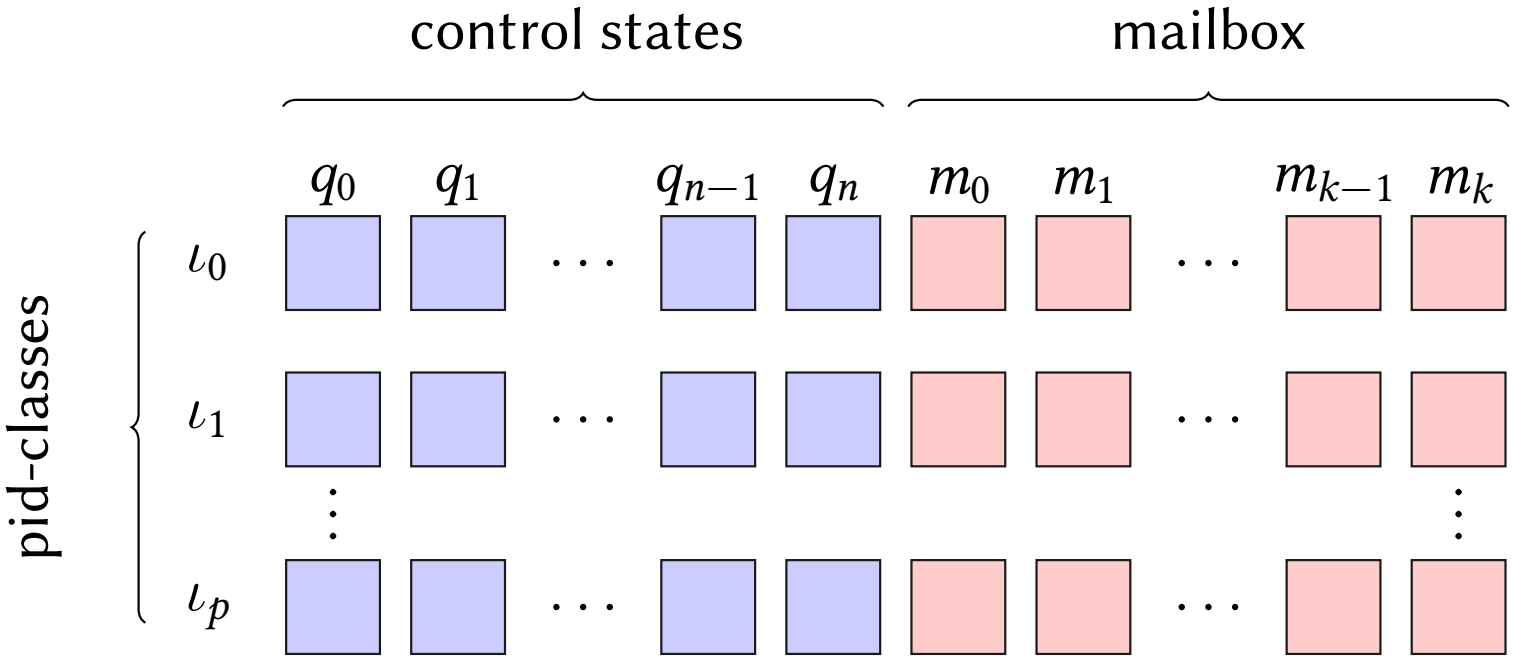
Underlying VAS



Encoding properties:

Bound on mailbox $\sum_{0 \leq i \leq k} \mathbf{v}(\iota_1, m_i) \leq B$

Underlying VAS



Encoding properties:

Coverability queries (EXPSPACE-complete)

Contributions:

- we handle the core features of Erlang
- the algorithm is sound and terminating
- the abstraction is parametric and models message-passing
- we use a combination of *static analysis* and *infinite-state model-checking*

Future Work

Planned work includes

- support for full Erlang
- extension to distributed/fault tolerant systems
- link with Type&Effect systems and work on π-calculus
- find ways to handle open systems
- counter-example guided abstraction refinement (CEGAR)
- more precise static analysis techniques
- support for liveness and LTL-properties

Related Work

Error-detection tools:

Related Work

Error-detection tools:

- **Dialyzer**
  Based on control-flow-analysis, *success types* and ad-hoc detection of wrong use of built-ins.
  Range of static analyses for specific concurrent properties (data-races on built-ins, unused receive patterns, ...).

- **QuickCheck** / **PropEr**
  Unit-testing for APIs.
  Detection of race conditions through clever generation of interleavings.

Related Work

Verification tools:

Related Work

Verification tools:

- **McErlang**
  On-the-fly software model-checker for Erlang. Employs an instrumented custom runtime which extensively explores all the traces. It is parametrised by user-provided abstractions for all components of the state space.
- **EtomCRL2** Verification tool for Erlang. Model-checking by translation to Turing-complete process algebra μCRL.

Thank you for your attention!