Automatic Verification of Erlang-Style Concurrency

Emanuele D'Osualdo, Jonathan Kochems, and C.-H. Luke Ong

University of Oxford

Abstract. This paper presents an approach to verify safety properties of Erlang-style, higher-order concurrent programs *automatically*. Inspired by Core Erlang, we introduce λActor, a prototypical functional language with pattern-matching algebraic data types, augmented with process creation and asynchronous message-passing primitives. We formalise an abstract model of λActor programs called *Actor Communicating System* (ACS) which has a natural interpretation as a vector addition system, for which some verification problems are decidable. We give a parametric abstract interpretation framework for λActor and use it to build a polytime computable, flow-based, abstract semantics of λActor programs, which we then use to bootstrap the ACS construction, thus deriving a more accurate abstract model of the input program. We evaluate the method which we implemented in the prototype Soter. We find that in practice our abstraction technique is accurate enough to verify an interesting range of safety properties. Though the ACS coverability problem is Expspace-complete, Soter can analyse non-trivial programs in a matter of seconds.

Keywords: Erlang, Infinite-state Systems Verification, Petri Nets.

1 Introduction

This paper concerns the verification of concurrent programs written in Erlang. Originally designed to program fault-tolerant distributed systems at Ericsson in the late 80s, Erlang is now a widely used, open-sourced language with support for higher-order functions, concurrency, communication, distribution, on-the-fly code reloading, and multiple platforms [3,2]. Largely because of a runtime system that offers highly efficient process creation and message-passing communication, Erlang is a natural fit for programming multicore CPUs, networked servers, parallel databases, GUIs, and monitoring, control and testing tools.

The sequential part of Erlang is a higher order, dynamically typed, call-by-value functional language with pattern-matching algebraic data types. Following the *actor model* [1], a concurrent Erlang computation consists of a dynamic network of processes that communicate by message passing. Every process has a unique process identifier (pid), and is equipped with an unbounded mailbox. Messages are sent asynchronously in the sense that send is non-blocking. Messages are retrieved from the mailbox, not FIFO, but First-In-First-Firable-Out (FIFFO) via pattern-matching. A process may block while waiting for a message that matches a certain pattern to arrive in its mailbox. For a quick and highly readable introduction to Erlang, see Armstrong's *CACM* article [2].

Challenges. Concurrent programs are hard to write. They are just as hard to verify. In the case of Erlang programs, the inherent complexity of the verification task can be seen from several diverse sources of infinity in the state space.

(∞1) General recursion requires a (process local) call-stack.
(∞2) Higher-order functions are first-class values; closures can be passed as parameters or returned.
(∞3) Data domains, and hence the message space, are unbounded: functions may return, and variables may be bound to, terms of an arbitrary size.
(∞4) An unbounded number of processes can be spawned dynamically.
(∞5) Mailboxes have unbounded capacity.

The challenge of verifying Erlang programs is that one must reason about the asynchronous communication of an unbounded set of messages, across an unbounded set of Turing-powerful processes.

Our goal is to verify safety properties of Erlang-like programs *automatically*, using a combination of static analysis and infinite-state model checking. To a large extent, the key decision of which causes of infinity to model as accurately as possible and which to abstract is forced upon us: the class consisting of a fixed set of context-free (equivalently, first-order) processes, each equipped with a mailbox of size one and communicating messages from a finite set, is already Turing powerful [10]. Our strategy is thus to abstract (∞1), (∞2) and (∞3), while seeking to analyse message-passing concurrency, assuming (∞4) and (∞5).

We consider programs of λ_{ACTOR}, a prototypical functional language with actor-style concurrency. λ_{ACTOR} is essentially *Core Erlang* [5]—the official intermediate representation of Erlang code, which exhibits in full the higher-order features of Erlang, with asynchronous message-passing concurrency and dynamic process creation. With decidable infinite-state model checking in mind, we introduce *Actor Communicating System* (ACS), which models the interaction of an unbounded set of communicating processes. An ACS has a finite set of control states Q, a finite set of *pid classes* P, a finite set of messages M, and a finite set of transition rules. An ACS transition rule has the shape $\iota\colon q \xrightarrow{\ell} q'$, which means that a process of pid class ι can transition from state q to state q' with (possible) *communication side effect* ℓ, of which there are four kinds, namely, (i) the process makes an internal transition (ii) it extracts and reads a message m from its mailbox (iii) it sends a message m to a process of pid class ι' (iv) it spawns a process of pid class ι'. ACS models are infinite state: the mailbox of a process has unbounded capacity, and the number of processes in an ACS may grow arbitrarily large. However the set of pid classes is fixed, and processes of the same pid class are not distinguishable.

An ACS can be interpreted naturally as a *vector addition system* (VAS), or equivalently Petri net, using counter abstraction. We consider a particular counter abstraction of ACS, called *VAS semantics*, which models an ACS as a VAS distinguishing two kinds of counters. A counter named by a pair (ι, q) counts the number of processes of pid class ι that are currently in state q; a counter named by (ι, m) counts the sum total of occurrences of a message m currently in the mailbox of p, where p ranges over processes of pid class ι. Using this abstraction, we can conservatively decide properties of the ACS using well-known decision procedures for VAS.

Parametric, Flow-based Abstract Interpretation. The starting point of our verification pathway is the abstraction of the sources of infinity ($\infty 1$), ($\infty 2$) and ($\infty 3$). Methods such as k-CFA [35] can be used to abstract higher-order recursive functions to a finite-state system. Rather than 'baking in' each type of abstraction separately, we develop a general abstract interpretation framework which is *parametric* on a number of basic domains. In the style of Van Horn and Might [36], we devise a machine-based operational semantics of λ_{ACTOR} which is 'generated' from the basic domains of *Time*, *Mailbox* and *Data*. We show that there is a simple notion of *sound abstraction of the basic domains* whereby every such abstraction gives rise to a sound abstract semantics of λ_{ACTOR} programs (Theorem 1). Further if a given sound abstraction of the basic domains is finite and the associated auxiliary operations are computable, then the derived abstract semantics is finite and computable.

Generating an ACS. We show that a sound ACS (Theorem 3) can be constructed in polynomial time by *bootstrapping* from the 0-CFA-like abstract semantics. Further, the dimension of the resulting ACS is polynomial in the length of the input λ_{ACTOR} program. The idea is that the 0-CFA-like abstract (transition) semantics constitutes a sound but rough analysis of the control-flow of the program, which takes higher-order computation into account but communicating behaviour only minimally. The bootstrap construction consists in constraining these rough transitions with guards of the form 'receive a message of this type' or 'send a message of this type' or 'spawn a process', thus resulting in a more accurate abstract model of the input λ_{ACTOR} program in the form of an ACS.

Evaluation. To demonstrate the feasibility of our verification method, we have constructed a prototype implementation called Soter. Our empirical results show that the abstraction framework is accurate enough to verify an interesting range of safety properties of non-trivial Erlang programs.

Outline. In Section 2 we define the syntax of λ_{ACTOR} and informally explain its semantics with the help of an example program. In Section 3, we introduce Actor Communicating System and its VAS semantics. In Section 4 we present a machine-based operational semantics of λ_{ACTOR} and then, in Section 5, we develop a parametric abstract interpretation from it. In Section 6, we use the analysis to bootstrap the ACS construction. In Section 7 we present the experimental results based on our tool implementation Soter, and discuss the limitations of our approach. We omit proofs for lack of space; they can be found in the extended version of the paper [12].

Notation. We write A^* for the set of finite sequences of elements of the set A, and ϵ for the null sequence. Let $a \in A$ and $l, l' \in A^*$, we overload '$\cdot$' so that it means insertion at the top $a \cdot l$, at the bottom $l \cdot a$ or concatenation $l \cdot l'$. We write l_i for the i-th element of l. The set of finite partial functions from A to B is denoted $A \rightharpoonup B$. we define $f[a \mapsto b] := (\lambda x.\, \textbf{if}\ (x = a)\ \textbf{then}\ b\ \textbf{else}\ f(x))$; $[]$ is the everywhere undefined function.

2 A Prototypical Fragment of Erlang

In this section we introduce λActor, a prototypical untyped functional language with actor concurrency. λActor is inspired by single-node *Core Erlang* [5] without built-in functions and fault-tolerant features. It exhibits in full the higher-order features of Erlang, with message-passing concurrency and dynamic process creation. The syntax of λActor is defined as follows:

$$
\begin{aligned}
e \in \mathit{Exp} ::= {} & x \mid \mathsf{c}(e_1,\ldots,e_n) \mid e_0(e_1,\ldots,e_n) \mid \mathit{fun} \\
& \mid \textbf{letrec } f_1(x_1,\ldots,x_{k_1}){=}e_1.\cdots f_n(x_1,\ldots,x_{k_n}){=}e_n. \textbf{ in } e \\
& \mid \textbf{case } e \textbf{ of } \mathit{pat}_1 \to e_1;\ldots;\mathit{pat}_n \to e_n \textbf{ end} \\
& \mid \textbf{receive } \mathit{pat}_1 \to e_1;\ldots;\mathit{pat}_n \to e_n \textbf{ end} \\
& \mid \textbf{send}(e_1,e_2) \mid \textbf{spawn}(e) \mid \textbf{self}() \\
\mathit{fun} ::= {} & \textbf{fun}(x_1,\ldots,x_n) \to e \\
\mathit{pat} ::= {} & x \mid \mathsf{c}(\mathit{pat}_1,\ldots,\mathit{pat}_n)
\end{aligned}
$$

where c ranges over a fixed finite set Σ of constructors.

For ease of comparison we keep the syntax close to Core Erlang and use uncurried functions, delimiters, **fun** and **end**. We write '_' for an unnamed unbound variable; using symbols from Σ, we write n-tuples as $\{e_1, \ldots, e_n\}$, the list constructor cons as $[\_|\_]$ and the empty list as $[]$. Sequencing $(e_1,\ e_2)$ is a shorthand for $(\textbf{fun}(\_)\to e_2)(e_1)$ and we we omit brackets for nullary constructors. The character '%' marks the start of a line of comment. Variable names begin with an uppercase letter. We write $\mathrm{fv}(e)$ for the free variables of an expression and we define a λActor program $\mathcal{P}$ to be a closed λActor expression. We associate a unique label l to each sub-expression e of a program and indicate that e is labelled by l by writing $\ell\colon e$. Take a term $\ell\colon(\ell_0\colon e_0(\ell_1\colon e_1,\ldots,\ell_n\colon e_n))$, we define $\ell.\mathrm{arg}_i := \ell_i$ and $\mathrm{arity}(\ell) := n$.

To illustrate λActor's concurrency model we sketch a small-step reduction semantics here. The rewrite rules for function application and λ-abstraction are identical to call-by-value λ-calculus; we write evaluation contexts as $E[\,]$. A state of the computation of a λActor program is a set Π of processes running in parallel. A process $\langle e\rangle^{\iota}_{\mathfrak{m}}$, identified by the pid ι, evaluates an expression e with mailbox $\mathfrak{m}$ holding unconsumed messages. Purely functional reductions performed by each process are independently interleaved. A **spawn** construct, $\textbf{spawn}(\textbf{fun}()\to e)$, evaluates to a fresh pid ι' and creates a new process $\langle e\rangle^{\iota'}_{\epsilon}$, with pid ι':

$$\langle E[\textbf{spawn}(\textbf{fun}()\to e)]\rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \quad\longrightarrow\quad \langle E[\iota']\rangle^{\iota}_{\mathfrak{m}} \parallel \langle e\rangle^{\iota'}_{\epsilon} \parallel \Pi$$

A **send** construct, $\textbf{send}(\iota, v)$, evaluates to the message v with the side-effect of appending it to the mailbox of the receiver process ι; thus send is non-blocking:

$$\langle E[\textbf{send}(\iota, v)]\rangle^{\iota'}_{\mathfrak{m}'} \parallel \langle e\rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \quad\longrightarrow\quad \langle E[v]\rangle^{\iota'}_{\mathfrak{m}'} \parallel \langle e\rangle^{\iota}_{\mathfrak{m}\cdot v} \parallel \Pi$$

The evaluation of a **receive** construct, $\textbf{receive } p_1 \to e_1 \ldots p_n \to e_n \textbf{ end}$, will block if the mailbox of the process in question contains no message that matches

```
letrec
%
res_start (Res) =
    spawn(fun() → res_free (Res)).
res_free (Res) =
    receive {lock, P} →
        send(P, {acquired, self ()}),
        res_locked (Res, P)
    end.
res_locked (Res, P) =
  receive
    {req, P, Cmd} →
      case Res(P, Cmd) of
        {NewRes, ok} →
          res_locked (NewRes, P);
        {NewRes, {reply, A}} →
          send(P, {ans, self (), A}),
          res_locked (NewRes, P)
      end;
    {unlock, P} → res_free (Res)
  end.
%
res_lock (Q)=
  send(Q, {lock, self ()}),
  receive {acquired, Q} → ok end.
res_unlock (Q)=
  send(Q, {unlock, self ()}).
res_request (Q, Cmd) =
  send(Q, {req, self (), Cmd}),
  receive {ans, Q, X} → X end.
res_do(Q, Cmd) =
  send(Q, {req, self (), Cmd}).
%
cell_start () =
  res_start (cell (zero)).
cell (X) = fun(_P, Cmd) →
  case Cmd of
    {write,Y} → {cell (Y), ok};
     read     → {cell (X), {reply, X}}
  end.
%
cell_lock    (C) = res_lock(C).
cell_unlock  (C) = res_unlock(C).
cell_read    (C) = res_request(C, read).
cell_write (C,X)=res_do(C, {write, X}).
%
inc(C) =
  cell_lock (C),
  cell_write (C, {succ, cell_read (C)}),
  cell_unlock (C).
add_to_cell (M, C) =
  case M of
    zero      →   ok;
    {succ, M'} →
      spawn(fun() → inc(C)),
      add_to_cell (M', C)
  end.
%
in  C = cell_start (),
    add_to_cell (N, C).
```

Fig. 1. Locked Resource (running example)

any of the patterns p_i. Otherwise, the first message m that matches a pattern, say p_i, is consumed by the process, and the computation continues with the evaluation of e_i. The pattern-matching variables in e_i are bound by θ to the corresponding matching subterms of the message m; if more than one pattern matches the message, then the first in textual order is fired

$$\langle E[\mathbf{receive}\ p_1 \to e_1 \ldots p_n \to e_n\ \mathbf{end}]\rangle^{\iota}_{\mathtt{m}\cdot m\cdot \mathtt{m}'} \parallel \Pi \quad \longrightarrow \quad \langle E[\theta e_i]\rangle^{\iota'}_{\mathtt{m}\cdot \mathtt{m}'} \parallel \Pi;$$

Note that message passing is *not* First-In-First-Out but rather First-In-First-Fireable Out (FIFFO): incoming messages are queued at the end of the mailbox, and the message that a receive construct extracts is not necessarily the first.

Example 1 (Locked Resource). Figure 1 shows an example λACTOR program. The code has three logical parts, which would constitute three modules in Erlang. The first part defines an Erlang *behaviour*[1] that governs the lock-controlled, concurrent access of a shared resource by a number of clients. A resource is viewed as a generic server implementing the locking protocol, parametrised on

[1] I.e. a module implementing a general purpose protocol, parametrised over another module containing the code specific to a particular instance. Note that we simulate modules with higher-order parameters, which is general enough to express in full the dynamic module system of Erlang.

a function that specifies how to react to requests. Note the use of higher-order arguments and return values. The function res_start creates a new process that runs an unlocked (res_free) instance of the resource. When unlocked, a resource waits for a {lock, P} message to arrive from a client P. Upon receipt of such a message, an acknowledgement message is sent back to the client and the control is yielded to res_locked. When locked (by a client P), a resource can accept requests {req,P,Cmd} from P—and from P only—for an unspecified command Cmd to be executed. After running the requested command, the resource is expected to return the updated resource handler and an answer, which may be the atom ok, which requires no additional action, or a couple {reply, Ans} which signals that the answer Ans should be sent back to the client. When an unlock message is received from P the control is given back to res_free. Note that the mailbox matching mechanism allows multiple locks and requests to be sent asynchronously to the mailbox of the locked resource without causing conflicts: the pattern matching in the locked state ensures that all the pending lock requests get delayed for later consumption once the resource gets unlocked. The functions res_lock, res_unlock, res_request, res_do hide the protocol from the user who can then use this API as if it was purely functional.

The second part implements a simple shared resource that holds a natural number, which is encoded using the constructors {succ, _} and zero, and allows a client to read its value or overwrite it with a new one. Without lock messages, a shared resource with such a protocol easily leads to inconsistencies.

The last part defines the function inc which accesses a locked cell to increment its value. The function add_to_cell adds M to the contents of the cell by spawning M processes incrementing it concurrently. Finally the entry-point of the program sets up a process with a shared locked cell and then calls add_to_cell. Note that N is a free variable; to make the example a program we can either close it by setting N to a constant or make it range over all natural numbers with the extension described in Section 5.

An interesting correctness property of this code is the mutual exclusion of the lock-protected region (i.e. line 49) of the concurrent instances of inc.

3 Actor Communicating Systems

In this section we explore the design space of abstract models of Erlang-style concurrency. We seek a model of computation that should capture the core concurrency and asynchronous communication features of λ_{Actor} and yet enjoys the decidability of interesting verification problems. In the presence of pattern-matching algebraic data types, the (sequential) functional fragment of λ_{Actor} is already Turing powerful [30]. Restricting it to a pushdown (equivalently, first-order) fragment but allowing concurrent execution would enable, using very primitive synchronization, the simulation of a Turing-powerful finite automaton with two stacks. A single finite-control process equipped with a mailbox (required for asynchronous communication) can encode a Turing-powerful queue automaton in the sense of Minsky. Thus constrained, we opt for a model of con-

current computation that has finite control, a finite number of messages, and a finite number of *process classes*.

Definition 1. *An* Actor Communicating System *(ACS) $\mathcal{A}$ is a tuple $\langle P, Q, M, R, \iota_0, q_0\rangle$ where P is a finite set of* pid-classes*, Q is a finite set of control-states, M is a finite set of messages, $\iota_0 \in P$ is the pid-class of the initial process, $q_0 \in Q$ is the initial state of the initial process, and R is a finite set of rules of the form $\iota\colon q \xrightarrow{\ell} q'$ where $\iota \in P$, $q, q' \in Q$ and ℓ is a label that can take one of four forms: τ (local transition), $?m$ with $m \in M$ (receive a message), $\iota'!m$ with $\iota' \in P$, $m \in M$ (send a message), $\nu\iota'.\, q''$ with $\iota' \in P$ and $q'' \in Q$ (spawn a new process in pid-class ι' starting from q'').*

Now we have to give ACS a semantics, but interpreting the ACS mailboxes as FIFFO queues would yield a Turing-powerful model. Our solution is to apply a *counter abstraction* on mailboxes: disregard the ordering of messages, but track the number of occurrences of every message in a mailbox. Since we bound the number of pid-classes, but wish to model dynamic (and hence unbounded) spawning of processes, we apply a second counter abstraction on the control states of each pid-class: we count, for each control-state of each pid-class, the number of processes in that pid-class that are currently in that state.

For soundness, we need to make sure that such an abstraction contains all the behaviours of the semantics with FIFFO mailboxes: if there is a matching term in the mailbox, then the corresponding branch is non-deterministically fired. To see the difference, take the ACS that has one process (named ι), three control states q, q_1 and q_2, and two rules $\iota\colon q \xrightarrow{?a} q_1$, $\iota\colon q \xrightarrow{?b} q_2$. When equipped with a FIFFO mailbox containing the sequence $c\,a\,b$, the process can only evolve from q to q_1 by consuming a from the mailbox, since it can skip c but will find a matching message (and thus not look further) before reaching the message b. In contrast, the counter semantics would let q evolve non-deterministically to both q_1 and q_2, consuming a or b respectively: the mailbox is abstracted to $[a \mapsto 1, b \mapsto 1, c \mapsto 1]$ with no information on whether a or b arrived first. However, the abstracted semantics does contain the traces of the FIFFO semantics.

The VAS semantics of an ACS is a state transition system equipped with counters that support increment and decrement (when non-zero) operations. Such infinite-state systems are known as *vector addition systems* (VAS).

Definition 2 (Vector Addition System). *A* vector addition system *(VAS) $\mathcal{V}$ is a pair (I, R) where I is a finite set of indices (called the* places *of the VAS) and $R \subseteq \mathbb{Z}^I$ is a finite set of* rules*. Thus a rule is just a vector of integers of dimension $|I|$, whose components are indexed (i.e. named) by the elements of I.*

The state transition system $[\![\mathcal{V}]\!]$ *induced by a VAS $\mathcal{V} = (I, R)$ has state-set $\mathbb{N}^I$ and transition relation $\{(\mathbf{v}, \mathbf{v}+\mathbf{r}) \mid \mathbf{v} \in \mathbb{N}^I, \mathbf{r} \in R, \mathbf{v}+\mathbf{r} \in \mathbb{N}^I\}$. We write $\mathbf{v} \leq \mathbf{v}'$ just if for all i in I, $\mathbf{v}(i) \leq \mathbf{v}'(i)$.*

The semantics of an ACS can now be given easily in terms of the underlying vector addition system.

Definition 3 (VAS semantics). *The semantics of an ACS* $\mathcal{A} = (P, Q, M, R, \iota_0, q_0)$ *is the transition system induced by the VAS* $\mathcal{V} = (I, \mathbf{R})$ *where* $I = P \times (Q \uplus M)$*. Each ACS rule in* R *is translated into a VAS rule in* $\mathbf{R}$ *as follows:* $\iota\colon q \xrightarrow{\tau} q'$ *is the vector that decrements* (ι, q) *and increments* (ι, q')*,* $\iota\colon q \xrightarrow{?m} q'$ *decrements* (ι, q) *and* (ι, m) *while incrementing* (ι, q')*,* $\iota\colon q \xrightarrow{\iota'!m} q'$ *decrements* (ι, q) *and increments both* (ι, q') *and* (ι', m)*,* $\iota\colon q \xrightarrow{\nu\iota'.\, q''} q'$ *decrements* (ι, q) *while incrementing both* (ι, q') *and* (ι', q'')*. Given a* $[\![\mathcal{V}]\!]$*-state* $\mathbf{v} \in \mathbb{N}^I$*, the component* $\mathbf{v}(\iota, q)$ *counts the number of processes in the pid-class* ι *currently in state* q*, while the component* $\mathbf{v}(\iota, m)$ *is the sum of the number of occurrences of the message* m *in the mailboxes of the processes of the pid-class* ι*.*

While infinite-state, many non-trivial properties are decidable on VAS including reachability, coverability and place boundedness; for more details see [14]. In this paper we focus on coverability, which is EXPSPACE-complete [33]: given two states s and t, is it possible to reach from s a state t' that covers t (i.e. $t' \leq t$)?

Which kinds of correctness properties of λActor programs can one specify by coverability of an ACS? We will be using ACS to *over-approximate* the semantics of a λActor program, so if a state of the ACS is not coverable, then it is not reachable in any execution of the program. It follows that we can use coverability to express safety properties such as: (i) unreachability of error program locations (ii) mutual exclusion (iii) boundedness of mailboxes: is it possible to reach a state where the mailbox of pid-class ι has more than k messages? If not we can allocate just k memory cells for that mailbox.

4 An Operational Semantics for λActor

In this section, we define an operational semantics for λActor using a *time-stamped CESK* machine*, following an approach by Van Horn and Might [36]. An unusual feature of such machines are *store-allocated continuations* which allow the recursion in a programs's control flow and data structure to be separated from the recursive structure in its state space.

A Concrete Machine Semantics. Without loss of generality, we assume that in a λActor program, variables are distinct, and constructors and cases are only applied to variables. The λActor machine defines a transition system on *(global) states* $s \in \mathit{State} := \mathit{Procs} \times \mathit{Mailboxes} \times \mathit{Store}$. An element π of $\mathit{Procs} := \mathit{Pid} \rightharpoonup \mathit{ProcState}$ associates a process with its *(local) state*, and an element μ of $\mathit{Mailboxes} := \mathit{Pid} \rightharpoonup \mathit{Mailbox}$ associates a process with its mailbox. We split a store σ into two partitions $\mathit{Store} := (\mathit{VAddr} \rightharpoonup \mathit{Value}) \times (\mathit{KAddr} \rightharpoonup \mathit{Kont})$ each with its address space, to separate *values* and *continuations*. By abuse of notation $\sigma(x)$ shall mean the application of the first component when $x \in \mathit{VAddr}$ and of the second when $x \in \mathit{KAddr}$.

The *local state* q of a process is a tuple in $\mathit{ProcState} := (\mathit{ProgLoc} \uplus \mathit{Pid}) \times \mathit{Env} \times \mathit{KAddr} \times \mathit{Time}$ consisting of (i) a pid, or a *program location* which is a subterm of the program, labelled with its occurrence; whenever it is clear from

Functional reductions

FunEval
if $\pi(\iota) = \langle \ell\colon (e_0(e_1, \ldots, e_n)), \rho, a, t\rangle$
$\quad b := \mathrm{new}_{\mathrm{kpush}}(\iota, \pi(\iota))$
then $\pi' = \pi[\iota \mapsto \langle e_0, \rho, b, t\rangle]$
$\quad \sigma' = \sigma[b \mapsto \mathsf{Arg}_0\langle \ell, \epsilon, \rho, a\rangle]$

ArgEval
if $\pi(\iota) = \langle v, \rho, a, t\rangle$
$\quad \sigma(a) = \kappa = \mathsf{Arg}_i\langle \ell, d_0 \ldots d_{i-1}, \rho', c\rangle$
$\quad\quad d_i := (v, \rho)$
$\quad\quad b := \mathrm{new}_{\mathrm{kpop}}(\iota, \kappa, \pi(\iota))$
then $\pi' = \pi[\iota \mapsto \langle \ell.\mathrm{arg}_{i+1}, \rho', b, t\rangle]$
$\quad \sigma' = \sigma[b \mapsto \mathsf{Arg}_{i+1}\langle \ell, d_0 \ldots d_i, \rho', c\rangle]$

Vars
if $\pi(\iota) = \langle x, \rho, a, t\rangle$
$\quad \sigma(\rho(x)) = (v, \rho')$
then $\pi' = \pi[\iota \mapsto \langle v, \rho', a, t\rangle]$

Apply
if $\pi(\iota) = \langle v, \rho, a, t\rangle$, $\mathrm{arity}(\ell) = n$
$\quad \sigma(a) = \kappa = \mathsf{Arg}_n\langle \ell, d_0 \ldots d_{n-1}, \rho', c\rangle$
$\quad\quad d_0 = (\mathbf{fun}(x_1 \ldots x_n) \to e, \rho_0) \quad d_n := (v, \rho)$
$\quad\quad b_i := \mathrm{new}_{\mathrm{va}}(\iota, x_i, \mathrm{res}(\sigma, d_i), \pi(\iota))$
$\quad\quad t' := \mathrm{tick}(\ell, \pi(\iota))$
then $\pi' = \pi[\iota \mapsto \langle e, \rho'[x_1 \to b_1 \ldots x_n \to b_n], c, t'\rangle]$
$\quad \sigma' = \sigma[b_1 \mapsto d_1 \ldots b_n \mapsto d_n]$

Fig. 2. Concrete Semantics rules for the functional primitives. The tables define the transition relation $s = \langle \pi, \mu, \sigma, \vartheta\rangle \to \langle \pi', \mu', \sigma', \vartheta'\rangle = s'$ by cases; the primed components of the state are identical to the non-primed components, unless indicated otherwise in the "then" part of the rule. The meta-variable v stands for terms that cannot be further rewritten such as λ-abstractions, constructor applications and un-applied primitives.

the context, we shall omit the label; (ii) an environment, which is a map from variables to pointers to values $\rho \in \mathit{Env} := \mathit{Var} \rightharpoonup \mathit{VAddr}$; (iii) a pointer to a continuation, which indicates what to evaluate next when the current evaluation returns a value; (iv) a time-stamp, which will be described later.

Values are either closures $d \in \mathit{Value} := \mathit{Closure} \uplus \mathit{Pid}$ or pids $\mathit{Closure} := \mathit{ProgLoc} \times \mathit{Env}$. Note that closures include both functions and constructor terms. All the above domains are naturally partially ordered: *ProgLoc* and *Var* are discrete partial orders, all others are defined by pointwise extension.

A *mailbox* is a finite sequence of values: $\mathfrak{m} \in \mathit{Mailbox} := \mathit{Value}^*$. We denote the empty mailbox by ϵ. A mailbox is supported by two operations:

$$\mathrm{mmatch}\colon \mathit{pat}^* \times \mathit{Mailbox} \times \mathit{Env} \times \mathit{Store} \to (\mathbb{N} \times (\mathit{Var} \rightharpoonup \mathit{Value}) \times \mathit{Mailbox})_\perp$$
$$\mathrm{enq}\colon \mathit{Value} \times \mathit{Mailbox} \to \mathit{Mailbox}$$

The function mmatch takes a list of patterns, a mailbox, the current environment and a store (for resolving pointers in the values stored in the mailbox) and returns the index of the matching pattern, a substitution witnessing the match, and the mailbox resulting from the extraction of the matched message. To model *Erlang-style* FIFFO mailboxes we set $\mathrm{enq}(d, \mathfrak{m}) := \mathfrak{m} \cdot d$ and define $\mathrm{mmatch}(p_1 \ldots p_n, \mathfrak{m}, \rho, \sigma) := (i, \theta, \mathfrak{m}_1 \cdot \mathfrak{m}_2)$ such that $\mathfrak{m} = \mathfrak{m}_1 \cdot d \cdot \mathfrak{m}_2$ with $\forall d' \in \mathfrak{m}_1$ and $\forall j\,.\,\mathrm{match}_{\rho,\sigma}(p_j, d') = \perp$, and $\theta = \mathrm{match}_{\rho,\sigma}(p_i, d)$ with $\forall j < i\,.\,\mathrm{match}_{\rho,\sigma}(p_j, d) = \perp$ where $\mathrm{match}_{\rho,\sigma}(p, d)$ pattern-matches term d against pattern p, using the environment ρ and store σ where necessary, and returns a substitution if successful and $\perp$ otherwise.

Evaluation Contexts as Continuations. Next we represent (in an inside-out manner) evaluation contexts as continuations. A continuation consists of a *tag* indicating the shape of the evaluation context, a pointer to a continuation representing the enclosing evaluation context, and, in some cases, a program location and an environment. Thus $\kappa \in \mathit{Kont}$ consists of the following constructs:

Concurrency

Receive
if $\pi(\iota) = \langle e, \rho, a, t\rangle$
$e = \textbf{receive}\ p_1 \to e_1 \dots p_n \to e_n\ \textbf{end}$
$\text{mmatch}(p_1 \dots p_n, \mu(\iota), \rho, \sigma) = (i, \theta, \mathfrak{m})$
$\theta = [x_1 \mapsto d_1 \dots x_k \mapsto d_k]$
$b_j := \text{new}_{\text{va}}(\iota, x_j, \text{res}(\sigma, d_j), \pi(\iota))$
$\rho' := \rho[x_1 \mapsto b_1 \dots x_k \mapsto b_k]$
then $\pi' = \pi[\iota \mapsto \langle e_i, \rho', a, t\rangle]$
$\mu' = \mu[\iota \mapsto \mathfrak{m}]$
$\sigma' = \sigma[b_1 \mapsto d_1 \dots b_k \mapsto d_k]$

Self
if $\pi(\iota) = \langle\, \textbf{self}\,(), \rho, a, t\rangle$
then $\pi' = \pi[\iota \mapsto \langle \iota, \rho, a, t\rangle]$

Send
if $\pi(\iota) = \langle v, \rho, a, t\rangle \quad \sigma(a) = \mathsf{Arg}_2\langle \ell, d, \iota', \_, c\rangle$
$d = (\textbf{send}, \_)$
then $\pi' = \pi[\iota \mapsto \langle v, \rho, c, t\rangle]$
$\mu' = \mu[\iota' \mapsto \text{enq}((v, \rho), \mu(\iota'))]$

Spawn
if $\pi(\iota) = \langle \textbf{fun}() \to e, \rho, a, t\rangle \quad d = (\textbf{spawn}, \_)$
$\sigma(a) = \mathsf{Arg}_1\langle \ell, d, \rho', c\rangle \quad \iota' := \text{new}_{\text{pid}}(\iota, \ell, t)$
then
$\pi' = \pi \left[\begin{array}{l} \iota \mapsto \langle \iota', \rho', c, t\rangle, \\ \iota' \mapsto \langle e, \rho, *, t_0\rangle \end{array}\right]$
$\mu' = \mu[\iota' \mapsto \epsilon]$

Fig. 3. Concrete Semantic Rules for Concurrency primitives.

- Stop represents the empty context.
- $\mathsf{Arg}_i\langle \ell, v_0 \dots v_{i-1}, \rho, a\rangle$ represents the context $E[v_0(v_1, \dots, v_{i-1}, [\,], e'_{i+1}, \dots, e'_n)]$ where $e_0(e_1, \dots, e_n)$ is the subterm located at ℓ; ρ closes the terms $e_{i+1}, \dots, e_n$ to $e'_{i+1}, \dots, e'_n$ respectively; the address a points to the continuation representing the enclosing evaluation context E.

Addresses, Pids and Time-Stamps. While the machine supports arbitrary concrete representations of time-stamps, addresses and pids, we present here an instance based on *contours* [35] which shall serve as the reference semantics of λ_{ACTOR}, and the basis for the abstraction.

A way to represent a *dynamic* occurrence of a symbol is the history of the computation at the point of its creation. We record history as *contours* which are strings of program locations $t \in \mathit{Time} := \mathit{ProgLoc}^*$. The initial contour is just the empty sequence $t_0 := \epsilon$, while the function tick: $\mathit{ProgLoc} \times \mathit{Time} \to \mathit{Time}$ updates the contour of the process in question by prepending the current program location $\text{tick}(\ell, t) := \ell \cdot t$. Addresses for values $b \in \mathit{VAddr} := \mathit{Pid} \times \mathit{Var} \times \mathit{Data} \times \mathit{Time}$ are represented by tuples comprising the current pid, the variable in question, the bound value and the current time stamp. Addresses for continuations $a, c \in \mathit{KAddr} := (\mathit{Pid} \times \mathit{ProgLoc} \times \mathit{Env} \times \mathit{Time}) \uplus \{*\}$ are represented by tuples comprising the current pid, program location, environment and time; or $*$ which is the address of the initial continuation (Stop).

The *data domain* ($\delta \in \mathit{Data}$) is the set of closed λ_{ACTOR} terms; the function $\text{res}: \mathit{Store} \times \mathit{Value} \to \mathit{Data}$ resolves all the pointers of a value through the store σ, returning the corresponding closed term $\text{res}(\sigma, (e, \rho)) := e[x \mapsto \text{res}(\sigma, \sigma(\rho(x))) \mid x \in \text{fv}(e)]$ or, when the value is a pid it just returns it $\text{res}(\sigma, \iota) := \iota$.
We extract the relevant components from the context to generate new addresses:

$$\text{new}_{\text{kpush}}: \mathit{Pid} \times \mathit{ProcState} \to \mathit{KAddr}$$
$$\text{new}_{\text{kpush}}(\iota, \langle \ell, \rho, \_, t\rangle) := (\iota, \ell.\text{arg}_0, \rho, t)$$
$$\text{new}_{\text{kpop}}: \mathit{Pid} \times \mathit{Kont} \times \mathit{ProcState} \to \mathit{KAddr}$$
$$\text{new}_{\text{kpop}}(\iota, \kappa, \langle \_, \_, \_, t\rangle) := (\iota, \ell.\text{arg}_{i+1}, \rho, t) \text{ where } \kappa = \mathsf{Arg}_i\langle \ell, \dots, \rho, \_\rangle$$

$$\text{new}_{\text{va}} \colon \mathit{Pid} \times \mathit{Var} \times \mathit{Data} \times \mathit{ProcState} \to \mathit{VAddr}$$
$$\text{new}_{\text{va}}(\iota, x, \delta, \langle \_, \_, \_, t \rangle) := (\iota, x, \delta, t)$$

To enable data abstraction in our framework, the address of a value contains the data to which the variable is bound: by making appropriate use of the embedded information in the abstract semantics, we can fine-tune the data sensitivity of our analysis.

Following the same scheme, pids ($\iota \in \mathit{Pid}$) can be identified with the contour of the **spawn** that generated them: $\mathit{Pid} := (\mathit{ProgLoc} \times \mathit{Time})$. Thus the generation of a new pid is defined as

$$\text{new}_{\text{pid}} \colon \mathit{Pid} \times \mathit{ProgLoc} \times \mathit{Time} \to \mathit{Pid}$$
$$\text{new}_{\text{pid}}((\ell', t'), \ell, t) := (\ell, \text{tick}^*(t, \text{tick}(\ell', t'))$$

where tick^* is just the simple extension of tick that prepends a whole sequence to another. Note that the new pid contains the pid that created it as a sub-sequence: it is indeed part of its history. The pid $\iota_0 := (\ell_0, \epsilon)$ is the pid associated with the starting process, where ℓ_0 is just the root of the program.

Remark 1. Note that the only sources of infinity for the state space are time, mailboxes and the data component of value addresses. If these domains are finite then the state space is finite.

Definition 4 (Concrete Semantics). *We define a (non-deterministic) transition relation on states* $(\to) \subseteq \mathit{State} \times \mathit{State}$. *In Figures 2 and 3 we present the rules for application, message passing and process creation; we omit rules for letrec, case and returning pids since they follow the same shape. The transition* $s \to s'$ *is defined by a case analysis of the shape of* s. *The initial state associated with a program* $\mathcal{P}$ *is* $s_{\mathcal{P}} := \langle \pi_0, \mu_0, \sigma_0 \rangle$ *where* $\pi_0 = [\iota_0 \mapsto \langle \mathcal{P}, [], *, t_0 \rangle]$, $\mu_0 = [\iota_0 \mapsto \epsilon]$ *and* $\sigma_0 = [* \mapsto \mathit{Stop}]$.

The rules for the purely functional reductions are a simple lifting of the corresponding rules for the sequential CESK* machine: when the currently selected process is evaluating a variable **Vars** its address is looked up in the environment and the corresponding value is fetched from the store and returned. **Apply**: When evaluating an application, control is given to each argument in turn—including the function to be applied; **FunEval** and **ArgEval** are then applied, collecting the values in the continuation. When the machine has evaluated all arguments, it records the new values in the environment and store, and passes control to the function-body. The rule **Receive** fires if mmatch returns a valid match from the process' mailbox and passes control to the expression in the matching clause with the pattern-variables populated by the matching substitution θ. When the machine applies rule **Send** it extracts the recipient's pid from the continuation, and calls enq to dispatch the message. Rule **Spawn** is enabled if the argument evaluates to a nullary function; the machine then creates a new process with a fresh pid running the body of the function.

One can easily add rules for run-time errors such as wrong arity in function application, non-exhaustive patterns in cases, sending to a non-pid and spawning a non-function.

Concurrent abstract reductions

AbsReceive

$$\begin{array}{l}\text{if } \widehat{\pi}(\widehat{\iota}) \ni \widehat{q} = \langle e, \widehat{\rho}, \widehat{a}, \widehat{t}\,\rangle \\ \quad e = \textbf{receive } p_1 \to e_1 \dots p_n \to e_n \textbf{ end} \\ \quad \widehat{\text{mmatch}}(p_1 \dots p_n, \widehat{\mu}(\widehat{\iota}), \widehat{\rho}, \widehat{\sigma}) \ni (i, \widehat{\theta}, \widehat{\mathfrak{m}}) \\ \quad \widehat{\theta} = [x_1 \mapsto \widehat{d}_1 \dots x_k \mapsto \widehat{d}_k] \\ \quad \widehat{\delta}_j \in \widehat{\text{res}}(\widehat{\sigma}, \widehat{d}_j) \\ \quad \widehat{b}_j := \widehat{\text{new}_{\text{va}}}(\widehat{\iota}, x_j, \widehat{\delta}_j, \widehat{q}\,) \\ \quad \widehat{\rho}' := \widehat{\rho}[x_1 \mapsto \widehat{b}_1 \dots x_k \mapsto \widehat{b}_k] \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\widehat{\iota} \mapsto \{\langle e_i, \widehat{\rho}', \widehat{a}, \widehat{t}\,\rangle\}] \\ \quad \widehat{\mu}' = \widehat{\mu}[\widehat{\iota} \mapsto \widehat{\mathfrak{m}}] \\ \quad \widehat{\sigma}' = \widehat{\sigma} \sqcup [\widehat{b}_1 \mapsto \{\widehat{d}_1\} \dots \widehat{b}_k \mapsto \{\widehat{d}_k\}]\end{array}$$

AbsSelf

$$\begin{array}{l}\text{if } \widehat{\pi}(\widehat{\iota}) \ni \langle\, \textbf{self}\,(), \widehat{\rho}, \widehat{a}, \widehat{t}\,\rangle \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\widehat{\iota} \mapsto \{\langle \widehat{\iota}, \widehat{\rho}, \widehat{a}, \widehat{t}\,\rangle\}]\end{array}$$

AbsSend

$$\begin{array}{l}\text{if } \widehat{\pi}(\widehat{\iota}) \ni \langle v, \widehat{\rho}, \widehat{a}, \widehat{t}\,\rangle \quad \widehat{\sigma}(\widehat{a}) \ni \mathsf{Arg}_2\langle \ell, \widehat{d}, \widehat{\iota}', \_, \widehat{c}\rangle \\ \quad \widehat{d} = (\textbf{send}, \_) \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\widehat{\iota} \mapsto \{\langle v, \widehat{\rho}, \widehat{c}, \widehat{t}\,\rangle\}] \\ \quad \widehat{\mu}' = \widehat{\mu}[\widehat{\iota}' \mapsto \widehat{\text{enq}}((v, \widehat{\rho}), \widehat{\mu}(\widehat{\iota}'))]\end{array}$$

AbsSpawn

$$\begin{array}{l}\text{if } \widehat{\pi}(\widehat{\iota}) \ni \langle \textbf{fun}() \to e, \widehat{\rho}, \widehat{a}, \widehat{t}\,\rangle \\ \quad \widehat{\sigma}(\widehat{a}) \ni \mathsf{Arg}_1\langle \ell, \widehat{d}, \widehat{\rho}', \widehat{c}\rangle \\ \quad \widehat{d} = (\textbf{spawn}, \_) \\ \quad \widehat{\iota}' := \widehat{\text{new}_{\text{pid}}}(\widehat{\iota}, \ell, \widehat{t}\,) \\ \text{then} \\ \quad \widehat{\pi}' = \widehat{\pi} \sqcup \begin{bmatrix} \widehat{\iota} \mapsto \{\langle \widehat{\iota}', \widehat{\rho}', \widehat{c}, \widehat{t}\,\rangle\}, \\ \widehat{\iota}' \mapsto \{\langle e, \widehat{\rho}, *, \widehat{t}_0\rangle\} \end{bmatrix} \\ \quad \widehat{\mu}' = \widehat{\mu} \sqcup [\widehat{\iota}' \mapsto \widehat{\epsilon}\,]\end{array}$$

Fig. 4. Abstract Semantic Rules for Concurrency primitives. We write $\sqcup$ for the join operation of the appropriate domain.

5 Parametric Abstract Interpretation

We aim to abstract the concrete operational semantics of Section 4 isolating the least set of domains that need to be made finite in order for the abstraction to be decidable. In Remark 1 we identify *Time*, *Mailbox* and *Data* as responsible for the unboundedness of the state space. Our abstract semantics is thus parametric on the abstraction of these basic domains.

Definition 5 (Basic domains abstraction). *A* data abstraction *is a triple* $\mathcal{D} = \langle \widehat{Data}, \alpha_d, \widehat{\text{res}}\rangle$ *where* $\widehat{Data}$ *is a flat (i.e. discretely ordered) domain of abstract data values,* $\alpha_d \colon Data \to \widehat{Data}$ *and* $\widehat{\text{res}} \colon \widehat{Store} \times \widehat{Value} \to \mathscr{P}(\widehat{Data})$*. A* time abstraction *is a tuple* $\mathcal{T} = \langle \widehat{Time}, \alpha_t, \widehat{\text{tick}}, \widehat{t}_0\rangle$ *where* $\widehat{Time}$ *is a flat domain of abstract contours,* $\alpha_t \colon Time \to \widehat{Time}$*,* $\widehat{t}_0 \in \widehat{Time}$*, and* $\widehat{\text{tick}} \colon ProgLoc \times \widehat{Time} \to \widehat{Time}$*. A* mailbox abstraction *is a tuple* $\mathcal{M} = \langle \widehat{Mailbox}, \leq_m, \sqcup_m, \alpha_m, \widehat{\text{enq}}, \widehat{\epsilon}, \widehat{\text{mmatch}}\rangle$ *where* $(\widehat{Mailbox}, \leq_m, \sqcup_m)$ *is a join-semilattice with least element* $\widehat{\epsilon} \in \widehat{Mailbox}$*,* $\alpha_m \colon Mailbox \to \widehat{Mailbox}$*,* $\widehat{\text{enq}} \colon \widehat{Value} \times \widehat{Mailbox} \to \widehat{Mailbox}$ *are monotone in mailboxes and* $\widehat{\text{mmatch}} \colon pat^* \times \widehat{Mailbox} \times \widehat{Env} \times \widehat{Store} \to \mathscr{P}(\mathbb{N} \times (Var \rightharpoonup \widehat{Value}) \times \widehat{Mailbox})$*. A* basic domains abstraction *is a triple* $\mathcal{I} = \langle \mathcal{D}, \mathcal{T}, \mathcal{M}\rangle$ *consisting of a data, a time and a mailbox abstraction.*

An abstract interpretation of the basic domains determines an interpretation of the other abstract domains as follows.

$$\widehat{State} := \widehat{Procs} \times \widehat{Mailboxes} \times \widehat{Store} \qquad \widehat{Procs} := \widehat{Pid} \to \mathscr{P}(\widehat{ProcState})$$
$$\widehat{ProcState} := (ProgLoc \uplus \widehat{Pid}) \times \widehat{Env} \times \widehat{KAddr} \times \widehat{Time}$$
$$\widehat{Store} := (\widehat{VAddr} \to \mathscr{P}(\widehat{Value})) \times (\widehat{KAddr} \to \mathscr{P}(\widehat{Kont}))$$

$$\widehat{Mailboxes} := \widehat{Pid} \to \widehat{Mailbox} \qquad \widehat{Pid} := (ProgLoc \times \widehat{Time}) \uplus \{\widehat{\iota_0}\} \qquad \widehat{\iota_0} := \widehat{t_0}$$
$$\widehat{Env} := Var \rightharpoonup \widehat{VAddr} \qquad \widehat{Value} := \widehat{Closure} \uplus \widehat{Pid} \qquad \widehat{Closure} := ProgLoc \times \widehat{Env}$$

each equipped with an abstraction function defined by an appropriate pointwise extension. We will call all of them α since it will not introduce ambiguities. The abstract domain $\widehat{Kont}$ is the pointwise abstraction of $Kont$, and we will use the same tags as those in the concrete domain. The abstract functions $\widehat{\text{new}_{\text{kpush}}}$, $\widehat{\text{new}_{\text{kpop}}}$, $\widehat{\text{new}_{\text{va}}}$ and $\widehat{\text{new}_{\text{pid}}}$, are defined exactly as their concrete versions, but on the abstract domains. When B is a flat domain, the abstraction of a partial map $C = A \rightharpoonup B$ to $\widehat{C} = \widehat{A} \to \mathscr{P}(\widehat{B})$, where $\widehat{f} \leq_{\widehat{C}} \widehat{g} \Leftrightarrow \forall \widehat{a}.\ \widehat{f}(\widehat{a}) \subseteq g(\widehat{a})$, is defined as $\alpha_C(f) := \lambda \widehat{a} \in \widehat{A}.\ \{\alpha_B(b) \mid (a, b) \in f \text{ and } \alpha_A(a) = \widehat{a}\}$.

The operations on the parameter domains need to 'behave' with respect to the abstraction functions: the standard correctness conditions listed below must be satisfied by their instances. These conditions amount to requiring that what we get from an application of a concrete auxiliary function is adequately represented by the abstract result of the application of the abstract counterpart of that auxiliary function. The partial orders on the domains are standard pointwise extensions of partial orders of the parameter domains.

Definition 6 (Sound basic domains abstraction). *A basic domains abstraction $\mathcal{I}$ is* sound *just if the conditions below are met by the auxiliary operations:*

$$\alpha_t(\text{tick}(\ell, t)) \leq \widehat{\text{tick}}(\ell, \alpha_t(t)) \tag{1}$$
$$\widehat{\sigma} \leq \widehat{\sigma}' \wedge \widehat{d} \leq \widehat{d}' \implies \widehat{\text{res}}(\widehat{\sigma}, \widehat{d}) \leq \widehat{\text{res}}(\widehat{\sigma}', \widehat{d}') \tag{2}$$
$$\forall \widehat{\sigma} \geq \alpha(\sigma).\ \alpha_d(\text{res}(\sigma, d)) \in \widehat{\text{res}}(\widehat{\sigma}, \alpha(d)) \tag{3}$$
$$\alpha_m(\text{enq}(d, \mathfrak{m})) \leq \widehat{\text{enq}}(\alpha(d), \alpha_m(\mathfrak{m})) \quad \alpha_m(\epsilon) = \widehat{\epsilon} \tag{4}$$

if $\text{mmatch}(\boldsymbol{p}, \mathfrak{m}, \rho, \sigma) = (i, \theta, \mathfrak{m}')$ *then* $\forall \widehat{\mathfrak{m}} \geq \alpha(\mathfrak{m})$, $\forall \widehat{\sigma} \geq \alpha(\sigma)$

$$\exists \widehat{\mathfrak{m}}' \geq \alpha(\mathfrak{m}') \text{ such that } (i, \alpha(\theta), \widehat{\mathfrak{m}}') \in \widehat{\text{mmatch}}(\boldsymbol{p}, \widehat{\mathfrak{m}}, \alpha(\rho), \widehat{\sigma}) \tag{5}$$

Following the Abstract Interpretation framework, one can exploit the soundness constraints to derive, by algebraic manipulation, the definitions of the abstract auxiliary functions which would then be correct by construction [26].

Definition 7 (Abstract Semantics). *Once the abstract domains are fixed, the rules that define the abstract transition relation are straightforward abstractions of the original ones. In Figure 4, we present the abstract counterparts of the concurrency rules for the operational semantics of Figure 3; the full list of the abstract rules can be found in the extended paper [12]. defining the nondeterministic abstract transition relation on abstract states $(\rightsquigarrow) \subseteq \widehat{State} \times \widehat{State}$. When referring to a particular program $\mathcal{P}$, the abstract semantics is the portion of the graph reachable from $s_{\mathcal{P}}$.*

Theorem 1 (Soundness of Analysis). *Given a sound abstraction of the basic domains, if $s \to s'$ and $\alpha_{cfa}(s) \leq u$, then there exists $u' \in \widehat{State}$ such that $\alpha_{cfa}(s') \leq u'$ and $u \rightsquigarrow u'$.*

Now that we have defined a sound abstract semantics we give sufficient conditions for its computability.

Theorem 2 (Decidability of Analysis). *If a given (sound) abstraction of the basic domains is finite, then the derived abstract semantics is finite; it is also decidable if the associated auxiliary operations (in Definition 6) are computable.*

Abstracting Mailboxes For the analysis to be computable abstract mailboxes need to be finite too. Abstracting addresses (and data) to a finite set, values, and thus messages, become finite. We abstract a mailbox by an un-ordered set of messages in the static analysis overcoming the potential unbounded length of mailboxes but loosing information about the sequence and removal of messages. This abstraction is formalised in the domain $\mathcal{M}_{\text{set}} := \langle \mathscr{P}(\widehat{Value}), \subseteq, \cup, \alpha_{\text{set}}, \widehat{\text{enq}}_{\text{set}}, \emptyset, \widehat{\text{mmatch}}_{\text{set}} \rangle$ where the abstract versions of enq and the matching function can be derived from the correctness condition: $\alpha_{\text{set}}(\mathfrak{m}) := \{\alpha(d) \mid \exists i.\ \mathfrak{m}_i = d\}$, $\widehat{\text{enq}}_{\text{set}}(\widehat{d}, \widehat{\mathfrak{m}}) := \{\widehat{d}\} \cup \widehat{\mathfrak{m}}$ and

$$\widehat{\text{mmatch}}_{\text{set}}(p_1 \dots p_n, \widehat{\mathfrak{m}}, \widehat{\rho}, \widehat{\sigma}) := \left\{ (i, \widehat{\theta}, \widehat{\mathfrak{m}}) \,\middle|\, \widehat{d} \in \widehat{\mathfrak{m}}, \widehat{\theta} \in \widehat{\text{match}}_{\widehat{\rho},\widehat{\sigma}}(p_i, \widehat{d}) \right\}$$

Abstracting Data. We included data in the value addresses in the definition of *VAddr* in order to allow for sensitivity towards data in the analysis. However, cutting contours is no longer sufficient to make *VAddr* finite. A simple solution is to use the trivial data abstraction $Data_0 := \{\_\}$, discarding the value, or if more precision is required, any finite data-abstraction would do: the analysis will then distinguish states that differ because of different bindings in their frame.

A data abstraction particularly well-suited to languages with algebraic data-types is the abstraction that discards every sub-term of a constructor term that is nested at a deeper level than a parameter D. We call $Data_D$ such an abstraction, the formal definition of which can be found in [12].

Abstracting Time. k-CFA is a specific time abstraction which yields an analysis that distinguishes dynamic contexts up to a given bound k; this is achieved by truncating contours at length k to obtain their abstract counterparts obtaining the abstract domain $Time_k := \bigcup_{0 \le i \le k} ProgLoc^i$, $\alpha^k_{\text{t}}(\ell_0 \dots \ell_k \cdot t) := \ell_0 \dots \ell_k$. The simplest analysis we can then define is a 0-CFA with the basic domains abstraction $\langle Data_0, Time_0, \widehat{Mailbox}_{\text{set}} \rangle$. With this instantiation many of the domains collapse into singletons. However, the analysis keeps a separate store and mailboxes for each abstract state and leads to an exponential algorithm. To improve the complexity we apply a widening along the lines of [36, Section 7]: we replace the separate store and separate mailboxes for each state by a global copy of each. This reduces significantly the state-space we need to explore: the algorithm becomes polynomial in the size of the program.

Considering other abstractions for the basic domains easily leads to exponential algorithms; in particular, the state-space grows linearly wrt the size of abstract data so the complexity of the analysis using $Data_D$ is exponential in D.

Open programs. Often it is useful to verify an open expression where its input is taken from a regular set of terms [30]. For this purpose we introduce a new primitive **choice** that non-deterministically calls one of its arguments. For instance, an interesting way of closing N in Example 1 is to bind it to any_num():

any_num() = **choice**(**fun**() → zero, **fun**() → {succ, any_num()}).

If the state running two or more instances of inc's critical section is uncoverable, then mutual exclusion is ensured for arbitrarily many instances of inc.

6 Generating the Actor Communicating System

The CFA algorithm allows us to derive a sound representation of the control-flow of the program taking into account higher-order computation and some information about synchronization. The abstract transition relation gives us a rough scheme of the possible transitions that we can 'guard' with communication and process creation actions. These guarded rules will form the definition of an ACS that simulates the semantics of the input λActor program.

Terminology. We identify a common pattern of the rules of the abstract semantics. In each rule **R**, the premise distinguishes an abstract pid $\widehat{\iota}$ and an abstract process state $\widehat{q} = \langle e, \widehat{\rho}, \widehat{a}, \widehat{t}\,\rangle$ associated with $\widehat{\iota}$ i.e. $\widehat{q} \in \widehat{\pi}(\widehat{\iota})$ and the conclusion of the rule associates a new abstract process state—call it $\widehat{q}'$—with $\widehat{\iota}$ i.e. $\widehat{q}' \in \widehat{\pi}'(\widehat{\iota})$. Henceforth we shall refer to $(\widehat{\iota}, \widehat{q}, \widehat{q}')$ as the *active components* of the rule **R**.

Definition 8 (Generated ACS). *Given a λActor program $\mathcal{P}$, a sound basic domains abstraction $\mathcal{I} = \langle \mathcal{T}, \mathcal{M}, \mathcal{D}\rangle$ and a sound data abstraction for messages $\mathcal{D}_{msg} = \langle \widehat{Msg}, \alpha_{msg}, \widehat{\text{res}}_{msg}\rangle$ the* Actor communicating system *generated by $\mathcal{P}$, $\mathcal{I}$ and $\mathcal{D}_{msg}$ is $\mathcal{A}_{\mathcal{P}} := \langle \widehat{Pid}, \widehat{ProcState}, \widehat{Msg}, R, \alpha(\iota_0), \alpha(\pi_0(\iota_0))\rangle$ where $s_{\mathcal{P}} = \langle \pi_0, \mu_0, \sigma_0, t_0\rangle$ is the initial state with $\pi_0 = [\iota_0 \mapsto \langle \mathcal{P}, [], *, t_0\rangle]$ and the rules in R are defined by induction over the following rules.*

(AcsRec) *If $\widehat{s} \rightsquigarrow \widehat{s}'$ is proved by* ***AbsReceive*** *with active components $(\widehat{\iota}, \widehat{q}, \widehat{q}')$ where $\widehat{d} = (p_i, \widehat{\rho}')$ is the abstract message matched by* $\widehat{\text{mmatch}}$ *and $\widehat{m} \in \widehat{\text{res}}_{msg}(\widehat{\sigma}, \widehat{d})$, then $\widehat{\iota}\colon \widehat{q} \xrightarrow{?\widehat{m}} \widehat{q}'$ is in R.*

(AcsSend) *If $\widehat{s} \rightsquigarrow \widehat{s}'$ is proved by* ***AbsSend*** *with active components $(\widehat{\iota}, \widehat{q}, \widehat{q}')$ where $\widehat{d}$ is the sent abstract value and $\widehat{m} \in \widehat{\text{res}}_{msg}(\widehat{\sigma}, \widehat{d})$, then $\widehat{\iota}\colon \widehat{q} \xrightarrow{\widehat{\iota}'!\widehat{m}} \widehat{q}'$ is in R.*

(AcsSp) *If $\widehat{s} \rightsquigarrow \widehat{s}'$ is proved by* ***AbsSpawn*** *with active component $(\widehat{\iota}, \widehat{q}, \widehat{q}')$ where $\widehat{\iota}'$ is the new abstract pid that is generated in the premise of the rule, which gets associated with the process state $\widehat{q}'' = \langle e, \widehat{\rho}, *\rangle$ then $\widehat{\iota}\colon \widehat{q} \xrightarrow{\nu\widehat{\iota}'.\widehat{q}''} \widehat{q}'$ is in R.*

(AcsTau) *If $\widehat{s} \rightsquigarrow \widehat{s}'$ is proved by any other rule with active components $(\widehat{\iota}, \widehat{q}, \widehat{q}')$, then $\widehat{\iota}\colon \widehat{q} \xrightarrow{\tau} \widehat{q}'$ is in R.*

As we will make precise later, keeping $\widehat{Pid}$ and $\widehat{ProcState}$ small is of paramount importance for the model checking of the generated ACS to be feasible. This is the main reason why we keep the message abstraction independent from

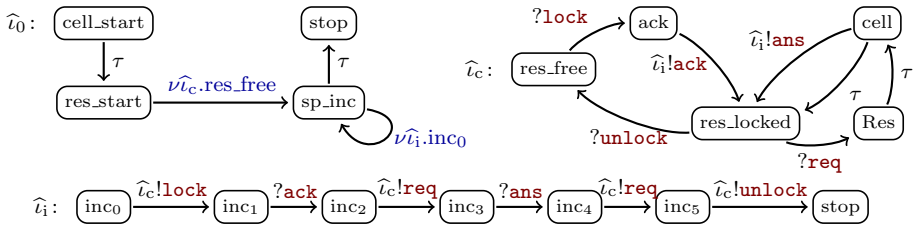

Fig. 5. ACS generated by the algorithm from Example 1

the data abstraction: this allows us to increase precision with respect to types of messages, which is computationally cheap, and keep the expensive precision on data as low as possible. It is important to note that these two 'dimensions' are in fact independent and a more precise message space enhances the precision of the ACS even when using $Data_0$ as the data abstraction.

In our examples (and in our implementation) we use a $Data_D$ abstraction for messages where D is the maximum depth of the receive patterns of the program.

Definition 9. *The function* $\alpha_{acs}\colon State \to (\widehat{Pid} \times (\widehat{ProcState} \uplus \widehat{Msg}) \to \mathbb{N})$ *relating concrete states and states of the ACS is defined as*

$$\alpha_{acs}(\langle \pi, \mu, \sigma \rangle) := \begin{cases} (\widehat{\iota}, \widehat{q}) & \mapsto \left|\{\iota \mid \alpha(\iota) = \widehat{\iota}, \alpha(\pi(\iota)) = \widehat{q}\}\right| \\ (\widehat{\iota}, \widehat{m}) & \mapsto \left|\left\{(\iota, i) \;\middle|\; \begin{array}{l} \alpha(\iota) = \widehat{\iota}, \\ \alpha_{msg}(\mathrm{res}(\sigma, \mu(\iota)_i)) = \widehat{m} \end{array}\right\}\right| \end{cases}$$

It is important to note that most of the decidable properties of the generated ACS are not even expressible on the CFA graph alone: being able to predicate on the contents of the counters means we can decide boundedness, mutual exclusion and many other expressive properties.

Theorem 3 (Soundness of generated ACS). *For all choices of* $\mathcal{I}$ *and* $\mathcal{D}_{msg}$*, for all concrete states* s *and* s'*, if* $s \to s'$ *and* $\alpha_{acs}(s) \leq \mathbf{v}$ *then there exists* $\mathbf{v}'$ *such that* $\alpha_{acs}(s') \leq \mathbf{v}'$*, and* $\mathbf{v} \to_{\mathrm{acs}} \mathbf{v}'$*.*

Let $\mathcal{A}_{\mathcal{P}}$ be the ACS derived from a given λActor program $\mathcal{P}$. From Theorem 3 we have that $[\![\mathcal{A}_{\mathcal{P}}]\!]$ simulates the semantics of $\mathcal{P}$: for each run $s \to s_1 \to s_2 \to \ldots$ of $\mathcal{P}$, there exists a $[\![\mathcal{A}_{\mathcal{P}}]\!]$-run $\mathbf{v} \to_{\mathrm{acs}} \mathbf{v}_1 \to_{\mathrm{acs}} \mathbf{v}_2 \to_{\mathrm{acs}} \ldots$ such that $\alpha_{\mathrm{acs}}(s) = \mathbf{v}$ and for all i, $\alpha_{\mathrm{acs}}(s_i) \leq \mathbf{v}_i$. Simulation preserves all paths so reachability (and coverability) is preserved.

Example 2. Figure 5 shows a pictorial representation of the ACS generated by our procedure from the program in Example 1 (with the parametric entry point of Section 5) using a 0-CFA analysis.The three pid-classes correspond to the starting process $\widehat{\iota_0}$ and the two static calls of spawn in the program, the one for the shared cell process $\widehat{\iota_c}$ and the other, $\widehat{\iota_i}$, for all the processes running inc.

The entry point is $(\widehat{\iota_0}, \mathrm{cell\_start})$. The second component represents the locking protocol quite faithfully. The VAS semantics is accurate enough to prove

mutual exclusion of state 'inc_2', which is protected by locks. This property can be stated as a coverability problem for VAS: can $\mathrm{inc}_2 = 2$ be covered? We can answer this question algorithmically: in this case the answer is negative and soundness allows us to conclude that our input program satisfies the property.

Complexity of the Generation. Generating an ACS from a program amounts to calculating the analysis of Section 5 and aggregating the relevant ACS rules for each transition of the analysis. Since we are adding $O(1)$ rules to R for each transition, the complexity of the generation is the same as the complexity of the analysis itself. The only reason for adding more than one rule to R for a single transition is the cardinality of $\widehat{Msg}$ but since this costs only a constant overhead, increasing the precision with respect to message types is not as expensive as adopting more precise data abstractions.

Dimension of the Abstract Model. The complexity of coverability on VAS is EXPSPACE in the dimension of the VAS; hence for the approach to be practical, it is critical to keep the number of components of the VAS underlying the generated ACS small; in what follows we call *dimension* of an ACS the dimension of the VAS underlying its VAS semantics.

Our algorithm produces an ACS with dimension $(|\widehat{ProcState}|+|\widehat{Msg}|)\times|\widehat{Pid}|$. With the 0-CFA abstraction described at the end of Section 5, $\widehat{ProcState}$ is polynomial in the size of the program and $\widehat{Pid}$ is linear in the size of the program so, assuming $|\widehat{Msg}|$ to be a constant, the dimension of the generated ACS is polynomial in the size of the program, in the worst case. Due to the parametricity of the abstract interpretation we can adjust for the right levels of precision and speed. For example, if the property at hand is not sensitive to pids, one can choose a coarser pid abstraction. It is also possible to greatly reduce $\widehat{ProcState}$: we observe that many of the control states result from intermediate functional reductions; such reductions performed by different processes are independent, thanks to the actor model paradigm. This allows for the use of preorder reductions. In our prototype, as described in Section 7, we implemented a simple reduction that collapses internal functional transitions, if irrelevant to the property at hand. This has proven to be a simple yet effective transformation yielding a significant speedup. We conjecture that, after the reduction, the cardinality of $\widehat{ProcState}$ is quadratic only in the number of **send**, **spawn** and **receive** of the program.

7 Evaluation, Limitations and Extensions

To evaluate the feasibility of the approach, we have constructed Soter, a prototype implementation of our method. Written in Haskell, Soter takes as input a single Erlang module annotated with safety properties in the form of simple assertions. Soter supports the full higher-order fragment and the (single-node) concurrency and communication primitives of Erlang; for more details about the tool see [11]. The annotated Erlang module is first compiled to Core Erlang by the Erlang compiler. A 0-CFA-like analysis, with support for the $Data_D$ data

Name	LOC	PRP	ORD	SAFE?	ABS D	ABS M	ACS Places	SIZE Ratio	TIME Analysis	TIME Simpl	TIME BFC	TIME Total
reslock	356	1	2	yes	0	2	40	10%	0.56	0.08	0.82	1.48
sieve	230	3	2	yes	0	2	47	19%	0.26	0.03	2.46	2.76
concdb	321	1	1	yes	0	2	67	12%	1.10	0.16	5.19	6.46
state_factory	295	2	2	yes	0	1	22	4%	0.59	0.13	0.02	0.75
pipe	173	1	2	yes	0	0	18	8%	0.15	0.03	0.00	0.18
ring	211	1	2	yes	0	2	36	9%	0.55	0.07	0.25	0.88
parikh	101	1	1	yes	0	2	42	41%	0.05	0.01	0.07	0.13
unsafe_send	49	1	1	no	0	1	10	38%	0.02	0.00	0.00	0.02
safe_send	82	1	1	no*	0	1	33	36%	0.05	0.01	0.00	0.06
safe_send	82	4	1	yes	1	2	82	34%	0.23	0.03	0.06	0.32
firewall	236	1	2	no*	0	2	35	10%	0.36	0.05	0.02	0.44
firewall	236	1	2	yes	1	3	74	10%	2.38	0.30	0.00	2.69
finite_leader	555	1	2	no*	0	2	56	20%	0.35	0.03	0.01	0.40
finite_leader	555	1	2	yes	1	3	97	23%	0.75	0.07	0.86	1.70
stutter	115	1	1	no*	0	0	15	19%	0.04	0.00	0.00	0.05
howait	187	1	2	no*	0	2	29	14%	0.19	0.02	0.00	0.22

Table 1. Soter Benchmarks. LOC is the number of lines of compiled Core Erlang. PRP is the number of properties to be proven. ORD is the order of the program. D and M are the data and message abstraction depth. In the "Safe?" column, "no*" means that the verification was inconclusive but the program is safe; "no" means that the program is not safe and Soter finds a genuine counterexample. "Places" is the number of places of the underlying Petri net after simplification; "Ratio" is the ratio between the number of places before and after simplification. All times are in seconds.

and message abstraction, is then performed on the compile; subsequently an ACS is generated. The ACS is simplified and then fed to the backend model-checker along with coverability queries translated from the annotations in the input Erlang program. Soter's backend is the tool BFC [20] which features a fast coverability engine for a variant of VAS. At the end of the verification pathway, if the answer is YES then the program is safe with respect to the input property, otherwise the analysis is inconclusive.

In Table 1 we summarise our experimental results. Many of the examples are higher-order, use dynamic (and unbounded) process creation and non-trivial synchronization. Example 1 appears as reslock and Soter proves mutual exclusion of the clients' critical section. concdb is the example program of [18] for which we prove mutual exclusion. pipe is inspired by the 'pipe' example of [21]; the property proved here is boundedness of mailboxes. sieve is a dynamically spawning higher-order concurrent implementation of Erathostene's sieve inspired by a program by Rob Pike [32]; Soter can prove all the mailboxes are bounded. safe_send, firewall and finite_leader could be successfully verified after refining the data abstraction. All example programs, annotated with coverability queries, can be viewed and verified using Soter at http://mjolnir.cs.ox.ac.uk/soter/.

There are programs and correctness properties that cannot be proved using any of the presented abstractions. Programs whose correctness depends on the order in which messages are delivered are abstracted too coarsely by the counter abstraction on mailboxes; however this is an uncommon pattern. Properties that assume the precise identification of processes are also not amenable to our approach because of the abstraction on pids. Finally, stack-based reasoning is out of reach of the current abstractions. The examples stutter and howait were designed specifically to illustrate Soter's limitations, see [12] for details.

Refinement and Extensions. The parametric definition of our abstract semantics allows us to tune the precision of the analysis. For safety properties, the counter-example witnessing a no-instance is a finite run of the abstract model. We conjecture that, given a spurious counter-example, a suitable refinement of the basic domains abstraction is computable which eliminates the spurious run of the corresponding abstract semantics. The development of a fully-fledged CEGAR loop is a topic of ongoing research.

The general architecture of our approach, combining static analysis and abstract model generation, can be adapted to accommodate different language features and different abstract models. By appropriate decoration of the analysis, it is possible to derive even more complex models for which semi-decision verification procedures have been developed [4,23].

8 Related Work and Conclusions

Verification or bug-finding tools for Erlang [24,29,22,7,8,6] typically rely on static analysis. The information obtained, usually in the form of a call graph, is then used to extract type constraints or infer runtime properties. Examples of static analyses of Erlang programs in the literature include data-flow [6], control-flow [29,22] and escape [7] analyses. Reppy and Xiao [34] and Colby [9] analyse the communication patterns of CML, which is based on typed channels and synchronous message passing, unlike Erlang's Actor-based model. To our knowledge, none of these analyses derives an *infinite-state* system.

Van Horn and Might [27] derive a CFA for a multithreaded extension of Scheme, using the same methodology [36] that we follow. The concurrency model therein is thread-based, and uses a compare-and-swap primitive. Our contribution, in addition to extending the methodology to Actor concurrency, is to use the derived parametric abstract interpretation to bootstrap the construction of an infinite-state abstract model for automated verification.

Venet [37] proposed an abstract interpretation framework for the sanalysis of π-calculus, later extended to other process algebras by Feret [13] and applied to CAP, a process calculus based on the Actor model, by Garoche [17]. In particular, Feret's non-standard semantics can be seen as an alternative to Van Horn and Might's methodology, but tailored for process calculi.

Huch [18] uses abstract interpretation and model checking to verify LTL-definable properties of a restricted fragment of Erlang programs: (i) order-one (ii) tail-recursive, (iii) mailboxes are bounded (iv) programs spawn a fixed, statically computable, number of processes. Given a data abstraction function, his method transforms a program to an abstract, finite-state model. In contrast, our method can verify Erlang programs of every finite order, with no restriction on the size of mailboxes, or the number of processes that may be spawned. Since our method of verification is by transformation to a *decidable infinite-state system* that simulates the input program, it is capable of greater accuracy.

McErlang is a model checker for Erlang programs developed by Fredlund and Svensson [15]. Given a program, a Büchi automaton, and an abstraction

function, McErlang explores on-the-fly a product of an abstract model of the program and the Büchi automaton. When the abstracted model has infinitely many reachable states, McErlang's exploration will not terminate. McErlang implements a fully-fledged Erlang runtime system, and it supports a substantial part of the language, including distributed and fault-tolerant features.

Asynchronous Programs, i.e. first-order recursive procedures with finite data which can make an unbounded number of asynchronous calls, can be encoded precisely into VAS and thus verified using reachability [19,16]. This infinite-state model supports call-stacks, through Parikh images, but not message-passing.

ACS can be expressed as processes in a suitable variant of CCS [28]. Decidable fragments of process calculi have been used in the literature to verify concurrent systems. Meyer [25] isolated a rich fragment of the π-calculus called *depth-bounded*. For certain patterns of communication, this fragment can be the basis of an abstract model that avoids the "merging" of mailboxes of the processes belonging to the same pid-class. Erlang programs however can express processes which are not depth bounded. We plan to address the automatic abstraction of arbitrary Erlang programs as depth-bounded process elsewhere.

Dialyzer [22,7,8] is a popular bug finding tool, included in the standard Erlang / OTP distribution. Given an Erlang program, the tool uses flow and escape [31] analyses to detect specific error patterns. Building on top of Dialyzer's static analysis, *success types* are derived. Lindahl and Sagonas' success types [22] 'never disallow the use of a function that will not result in a type clash during runtime' and thus never generate false positives. Dialyzer puts to good use the type annotations that programmers do use in practice; it scales well and is effective in detecting 'discrepancies' in Erlang code. However, success typing cannot be used to *verify* program correctness.

Conclusion. We have defined a generic analysis for λ_{ACTOR}, and a way of extracting from the analysis a simulating infinite-state abstract model in the form of an ACS, which can be automatically verified for coverability: if a state of the abstract model is not coverable then the corresponding concrete states of the input λ_{ACTOR} program are not reachable. Our constructions are parametric thus enabling different analyses to be easily instantiated. In particular, with a 0-CFA-like specialisation of the framework, the analysis and generation of the ACS are computable in polynomial time. Further, the dimension of the resulting ACS is polynomial in the length of the input program, small enough for the verification problem to be tractable in many useful cases. The empirical results using our prototype implementation Soter are encouraging. They demonstrate that the abstraction framework can be used to prove interesting safety properties of non-trivial programs automatically.

References

1. G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.
2. J. Armstrong. Erlang. *CACM*, 53(9):68, 2010.

3. J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in Erlang*. Prentice Hall, 1993.
4. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *ACM SIGPLAN Notices*, volume 38, pages 62–73, 2003.
5. R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, 2001.
6. R. Carlsson, K. Sagonas, and J. Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM TOPLAS*, 2006.
7. M. Christakis and K. Sagonas. Static detection of race conditions in erlang. *PADL*, pages 119–133, 2010.
8. M. Christakis and K. Sagonas. Detection of asynchronous message passing errors using static analysis. *PADL*, pages 5–18, 2011.
9. C. Colby. Analyzing the communication topology of concurrent programs. In *PEPM*, pages 202–213, 1995.
10. E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Verifying Erlang-style concurrency automatically. Technical report, University of Oxford DCS Technical Report, 2011. Available at http://mjolnir.cs.ox.ac.uk/soter/cpmrs.pdf.
11. E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Soter: an automatic safety verifier for Erlang. AGERE!'12, pages 137–140. ACM, 2012.
12. E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Automatic verification of Erlang-style concurrency. *CoRR*, abs/1303.2201, 2013. Available at http://arxiv.org/abs/1303.2201.
13. J. Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63(1):59130, 2005.
14. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.
15. L. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *ICFP*, pages 125–136, 2007.
16. P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *TOPLAS*, 34(1), 2012.
17. P. Garoche, M Pantel, and X. Thirioux. Static safety for an actor dedicated process calculus by abstract interpretation. In *FMOODS*, pages 78–92, 2006.
18. F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *ICFP*, pages 261–272, 1999.
19. R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. POPL'07, pages 339–350, New York, NY, USA, 2007. ACM.
20. A. Kaiser, D. Kroening, and T. Wahl. Efficient coverability analysis by proof minimization. In *CONCUR*, 2012. www.cprover.org/bfc/.
21. N. Kobayashi, M. Nakade, and A. Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. *Static Analysis*, pages 225–242, 1995.
22. T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178, 2006.
23. Z. Long, G. Calin, R. Majumdar, and R. Meyer. Language-Theoretic abstraction refinement. In *FASE*, pages 362–376, 2012.
24. S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *ICFP*, pages 136–149, 1997.
25. R. Meyer. On boundedness in depth in the π-calculus. In *Fifth Ifip International Conference On Theoretical Computer Science*, pages 477–489, 2008.
26. J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. *Static Analysis*, pages 347–362, 2008.
27. M. Might and D. Van Horn. A family of abstract interpretations for static analysis of concurrent higher-order programs. *Static Analysis*, pages 180–197, 2011.
28. R. Milner. *A calculus of communicating systems*, volume 92. Springer-Verlag Germany, 1980.
29. S. Nyström. A soft-typing system for Erlang. In *ACM Sigplan Erlang Workshop*, pages 56–71, 2003.
30. C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL*, pages 587–598, 2011.
31. Y. G. Park and B. Goldberg. Escape analysis on lists. In *ACM SIGPLAN Notices*, volume 27, pages 116–127, 1992.
32. R. Pike. Concurrency and message passing in Newsqueak. Google Talks Archive. Available at http://youtu.be/hB05UFqOtFA.
33. C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6:223–231, 1978.
34. J. H. Reppy and Y. Xiao. Specialization of CML message-passing primitives. In *POPL*, pages 315–326, 2007.
35. O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.
36. D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP*, pages 51–62, 2010.
37. Arnaud Venet. Abstract interpretation of the pi-calculus. In *LOMAPS*, pages 51–75, 1996.