University of Kent

Kent Academic Repository

Orchard, Dominic A., Munteanu, Mihail and Torrens, Paulo (2023) *Communicating actor automata - modelling erlang processes as communicating machines.* In: Castellani, Illaria and Scalas, Alceste, eds. Electronic Proceedings in Theoretical Computer Science. Proceedings 14th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2023). 378. pp. 38-48. Open Publishing Association

Downloaded from
https://kar.kent.ac.uk/101035/ The University of Kent's Academic Repository KAR

The version of record is available from
https://doi.org/10.4204/eptcs.378.4

This document version
Publisher pdf

DOI for this version

Licence for this version
CC BY (Attribution)

Additional information

Versions of research works

Versions of Record
If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts
If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in ***Title of Journal*** , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries
If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

Communicating Actor Automata – Modelling Erlang Processes as Communicating Machines

Dominic Orchard
University of Kent, UK
d.a.orchard@kent.ac.uk

Mihail Munteanu
Masabi Ltd.
mihailmunteanu944@gmail.com

Paulo Torrens
University of Kent, UK
paulotorrens@gnu.org

Brand and Zafiropulo's notion of Communicating Finite-State Machines (CFSMs) provides a succinct and powerful model of message-passing concurrency, based around channels. However, a major variant of message-passing concurrency is not readily captured by CFSMs: the actor model. In this work, we define a variant of CFSMs, called Communicating Actor Automata, to capture the actor model of concurrency as provided by Erlang: with mailboxes, from which messages are received according to repeated application of pattern matching. Furthermore, this variant of CFSMs supports dynamic process topologies, capturing common programming idioms in the context of actor-based message-passing concurrency. This gives a new basis for modelling, specifying, and verifying Erlang programs. We also consider a class of CAAs that give rise to freedom from race conditions.

1 Introduction

Modern software development often deviates from the traditional approach of sequential computation and thrives on *concurrency*, where code is written such that several processes may run simultaneously while potentially sharing resources. Out of the increasing complexity of software systems, platforms and programming languages were created to facilitate the development of concurrent, parallel, and distributed computations, such as the Erlang programming language [1, 2]. The need for formal specification of such systems has motivated the design of formal systems that allow programs to be reasoned about.

The Communicating Finite-State Machines (CFSMs) of Brand and Zafiropulo (also known as *communicating automata*) provide a model for describing concurrent, communicating processes in which a notion of *well-formed* communication protocols can be described [7]. The essential idea is to model a (finite) set of concurrent processes as finite automata (one automaton per process) whose labels correspond to sending and receiving messages on channels connecting each pair of automata, collectively called a *protocol*. Through such precise descriptions, properties of communicating processes can be studied, e.g., checking whether every message is received, or whether no process is left awaiting for a message which is never sent.

This model is useful for further studying the decidability, or undecidability, of various properties of concurrent systems (e.g., [16, 17, 11, 13, 14]). For example, Brand and Zafiropulo show that boundedness (i.e., that communication can proceed with bounded queues), deadlock freedom, and unspecified reception (sending messages that aren't received) are all decidable properties when restricted to two machines with a single type of message [7]. Furthermore, deciding these properties can be computed in polynomial time [18], and can be computed when only one machine is restricted to a single type of message. CFSMs have also been employed more recently as a core modelling tool that provides a useful interface between other models of concurrent programs, such as graphical choreographies [15].

In the CFSM model, processes communicate via channels (FIFO queues) linking each process. Therefore a process knows from which other process a message is received. This differs to the actor

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. 38–48, doi:10.4204/EPTCS.378.4

© D. Orchard, M. Munteanu & P. Torrens

model where each process has a 'mailbox' into which other processes deposit messages, not necessarily with any information about the sender. This makes it difficult to capture actor-based approaches in traditional CFSMs. A further limitation of CFSMs is that they capture programs with fixed communication topologies: both sender and receiver of a message are fixed in the model, and a process cannot have its messages dynamically targeted to different processes. However, this is not the predominant programming idiom in concurrent programming settings. CFSMs also prescribe simple models of message reception, and do not capture more fine-grained reception methods, such as Erlang's mailbox semantics which allow processing messages other than the most recent one, leveraging pattern matching at the language level. Even so, CFSMs are tantalisingly close to Erlang's computational model, with every pair of processes representing sequential computation that is able to communicate bidirectionally.

We propose a variant of CFSMs to capture Erlang's asynchronous mailbox semantics, and furthermore allow dynamic topologies through the binding (and rebinding) of variables for process identifiers via a notion of memory within a process' automaton. Section 2 explicates the model including examples of models corresponding to simple Erlang programs. We consider properties of such models in Section 3. Section 4 concludes with a discussion, some related work, and next steps.

1.1 Communicating Finite-State Machines

To facilitate comparison, we briefly recap the formal definition of CFSMs [7]. A system of N-CFSMs is referred to as a *protocol* which comprises four components (usually represented as a 4-tuple):

- $(S_i)_{i=1}^N$ are N (disjoint) finite sets S_i giving the set of states of each process i;
- $(o_i)_{i=1}^N$ where $o_i \in S_i$ are the initial states of each process i;
- $(M_{i,j})_{i,j=1}^N$ are N^2 (disjoint) finite sets where $M_{i,j}$ represents the set of messages that can be sent from process i to process j, and where $M_{i,j} = \emptyset$ when $i = j$;
- $(\mathsf{succ} : (S_i \times \bigcup_{j=1}^N (M_{i,j} \uplus M_{j,i})) \to S_i)_{i=1}^N$ are N state transition functions (*partial* functions) where $\mathsf{succ}(s,l)$ computes the successor state s' for process i from the state s and given a message l that is either being sent from i to j (thus $l \in M_{i,j}$) or being received from j by i (thus $l \in M_{j,i}$).

The typical presentation views the above indexed sets as finite sequences. We use the notation l for messages as we later refer to these as being 'labels' of automata.

For a protocol, a *global state* (or configuration) is a pair (S,C) of a sequence of states for each process $S = \langle s_1 \in S_1, \ldots s_N \in S_N \rangle$ and C is an $N \times N$ matrix whose elements $c_{i,j} \in C$ are finite sequences drawn from $M_{i,j}$ representing a FIFO queue (channel) of messages between process i and j (here and later we use $\cdot$ to concatenate such sequences and $[l_1, \ldots, l_m]$ for an instance of a sequence with m messages).

A binary-relation *step* captures when one global state (S,C) can evolve into another global state (S',C') due to a single succ function. That is, $(S',C') \in \mathsf{step}(S,C)$ iff there exists i, k, $l_{i,k}$ and either:

	1.	(i sends to k):	$s'_i \in S' = \mathsf{succ}_i(s_i, l_{i,k})$ and $c'_{i,k} \in C' = c_{i,k} \cdot [l_{i,k}]$
or	2.	(reception from k by i):	$s'_k \in S' = \mathsf{succ}_k(s_k, l_{i,k})$ and $c_{i,k} \in C = [l_{i,k}] \cdot c'_{i,k}$

We adopt similar terminology and structure, but vary enough to capture the actor model of Erlang.

2 A variant of CFSMs for Erlang

Our main goal is to define a CFSM variant, which we call Communicating Actor Automata (CAA), by borrowing from Erlang's mailbox semantics. We first review some core Erlang concepts, and follow by describing CAA and their composition into protocols. While the traditional definition of CFSMs immediately considers a global configuration (state) of some processes, we take each process (representing an Erlang actor) as a separate state machine, which gives a local "in-isolation" characterisation from which the global "protocol" characterisation is derived.

2.1 Erlang basic definitions

A key concept in Erlang is that of a *mailbox*: instead of processing messages strictly in FIFO order, each process (also referred to as an *actor*) possesses a queue of incoming messages from which they may *match*: given a sequence of patterns, a process picks the first message from the queue that unifies with one of those patterns, in an ordered fashion. If no pattern matches the first message in the queue, the next message is tried for all patterns, and so on [8]. In the following, we will use Erlang's term syntax in an abstract way (for details, see Carlsson et al. [9]), whose actual choice may vary. Concrete syntax is given in `monospace` font, e.g., `{1, 2}` is an Erlang tuple of two integer terms.

Definition 2.1 (Syntactic categories)**.** Let *Term* be the set of terms, ranged over by t. Let $\mathit{Var} \subset \mathit{Term}$ be the set of variables, ranged over by uppercase letters. Let $\mathit{Proc} \subset \mathit{Var}$ be the set of process identifiers which uniquely identify processes. Let *Pat* be the set of patterns, ranged over by pat. As Erlang has a call-by-value semantics, we define a subset $\mathit{Value} \subset \mathit{Term}$ of terms which are normal forms (called *values*), ranged over by v. Notice that $\mathit{Var} \subset \mathit{Value}$.

We note that, in regard to the semantics proposed in this paper, we mostly focus on four basic kinds of terms: process identifiers, variables, atoms and tuples. While identifiers and variables allow us to control the topology, atoms and tuples are useful for structuring message patterns (aiding in identifying which message is to be sent or received). In the example that will be given in Section 2.3, we shall also consider integers and arithmetic operators, but that's not necessary. It would also be possible to consider functions in the term syntax, but we won't entertain this possibility in this paper as we intend to focus on the expressivity of the process automaton itself and not on the term language.

In order to mimic Erlang's method of receiving messages, we need a notion of unification: incoming messages are matched against a set of patterns, and will proceed only if one of those is accepted.

Definition 2.2 (Unification)**.** We define the notion of *unification* [8, 9] between a term and a pattern written $t \rhd \mathit{pat}$ which either yields $\perp$ representing failure to pattern match or it yields an *environment* which is a finite map from a subset of variables $V \subseteq \mathit{Var}$ to terms, i.e., $\Gamma_V : V \to \mathit{Term}$ represents the binding context of a successful pattern match. Such maps are ranged over by γ. If $t \rhd \mathit{pat} = \gamma$, with $\gamma \in \Gamma_V$ for some V, then pat becomes equal to t if we replace every variable $v \in V$ in it by $\gamma_V(v)$. We write $\{X_1 \mapsto t_1,\ \ldots,\ X_n \mapsto t_n\}$ to denote the environment that maps X_i to t_i (for all $1 \le i \le n$).

2.2 CAAs for individual processes

Just as in CFSMs, we describe actors by finite-state automata. Each automaton is described individually and represents a single Erlang process, having its own unique identifier. Our main intention is to capture possible states in an Erlang process, by saying which messages it's allowed to receive and to send at a given point during execution, and what it should do after it.

We formally define our notion of actor as follows (note the addition of final states, not always included for CFSMs).

Definition 2.3. A Communicating Actor Automata (CAA) is a 7-tuple $(S,\ o,\ F,\ \mathscr{L}_!,\ \mathscr{L}_?,\ \delta, <)$, which includes a finite set of states S, an initial state $o \in S$, a possibly empty set of final states $F \subseteq S$, a set of send labels $\mathscr{L}_! \subseteq \mathit{Term}$, a finite set of receive labels $\mathscr{L}_? \subseteq \mathit{Pat}$, a function $\delta : S \times (\mathscr{L}_? \cup (\mathit{Var} \times \mathscr{L}_!)) \to S$, describing transitions, and an S-indexed family of order relations $<$ on $\mathscr{L}_?$. We impose a restriction on the domain of δ such that a state may only have either some number of receive labels or a single send label (but never both). We write $\delta(s, ?pat)$ for any transitions which receive a message matching pat, and $\delta(s, X!t)$ for transitions which send a message, where $X!t$ denotes the pair (X,t) of a message given by term t being sent to process X. For an state $s \in S$, we write $<_s$ as the order relation on such state.

Notice a CAA is essentially a deterministic finite automata (DFA) state machine with the alphabet defined as $\Sigma = \mathscr{L}_? \cup (\mathit{Var} \times \mathscr{L}_!)$. Non-determinism is exposed by interaction of many CAAs in a protocol, which will be described in Section 2.3. As such, a notion of a "static CAA" (as opposed to mobile) can be conceived of as having transitions $\delta : S \times (\mathscr{L}_? \cup (\mathit{Proc} \times \mathscr{L}_!)) \to S$ where we replace the variables associated with send labels by concrete process identifiers, i.e., the target of a send is always known ahead of time. We do not explore this notion further here.

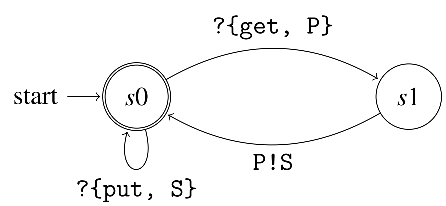
Example 2.1. Consider the following Erlang code which defines a function `mem` which emulates a memory cell via recursion:

```
mem(S) -> receive
            {get, P} -> P!S, mem(S);
            {put, X} -> mem(X)
          end.
```

The state of the memory cell is given by variable `S`. The process receives either a pair `{get, P}`, after which it sends `S` to the process identifier `P`, or a pair `{put, X}`, after which it recurses with `X` as the argument (the 'updated' memory cell state). Note that lowercase terms in Erlang are atoms.

Once spawned, this function can be modelled as a CAA with states $S = \{s_0, s_1\}$, initial and final states $o = s_0$ and $F = \{s_0\}$, send labels $\mathscr{L}_! = \mathit{Terms}$, receive labels $\mathscr{L}_? = \{$`{get, P}`, `{put, S}`$\}$, order stating `{get, P}` $<_{s_0}$ `{put, S}`, and the following transition (with corresponding automaton):

$$
\begin{aligned}
\delta(s_0,\ ?\texttt{\{get, P\}}) &= s_1 \\
\delta(s_0,\ ?\texttt{\{put, S\}}) &= s_0 \\
\delta(s_1,\ \texttt{P!S}) &= s_0
\end{aligned}
$$



Notice we don't use the variable X in the above example for the 'put' message, as we want to rebind the received value (in the second component of the pair) to S in the recursive call. We do not consider the additional aspect of Erlang's semantics in which already bound variables may appear in pattern matches, incurring a unification, which is a further complication not considered in this paper.

The above definition is enough to capture the static semantics of an actor. However, during execution, further information is needed to represent its dynamic behaviour at runtime: namely the mailbox and the internal state of the actor. We proceed to formally define this.

Definition 2.4. A *local state* (or *machine configuration*) is a triple (s, m, γ), being comprised of a state s, a finite sequence of terms m, and an environment γ. The sequence of terms m models *message queues*, also called actor *mailboxes*, of unreceived messages, and γ is the actor's memory. We write ε for the empty sequence and $[t_1, \ldots, t_n]$ for the sequence comprising n elements with t_1 being the head of the queue. Two sequences m, m' can be appended, written $m \cdot m'$, e.g., $[1,2,3] \cdot [4,5] = [1,2,3,4,5]$.

2.3 Systems of CAAs: protocols, states, and traces

As with a CFSM model or an Erlang program, computation is described by the communication among concurrent actors. In order to formally define that, we give an operational semantics to the combination of several CAAs, called a protocol, through an evaluation step relation between states.

Definition 2.5 (Protocol)**.** A *protocol* is an indexed family of CAAs, of finite size (or arity) N, written $\langle (S_i,\ o_i,\ F_i,\ \mathscr{L}_{!i},\ \mathscr{L}_{?i},\ \delta_i)\rangle_{i=1}^{N}$. Each index i represents the unique process identifier of each process.

Definition 2.6 (Global state)**.** A *global state* (or *system configuration*) for a protocol comprises a finite sequence of N local states, written $\langle (s_i, m_i, \gamma_i)\rangle_{i=1}^{N}$ where every $s_i \in S_i$. We denote the set of global states for a protocol with arity N as G_N.

Given a protocol, we may derive what we call an initial global state: before starting, each process has an empty mailbox, and is in its initial state as defined in its own CAA. This initial state is deterministic: for any given protocol, there's a single possible initial state.

Definition 2.7 (Initial global state)**.** For a protocol, the *initial global state* is $\langle (o_i,\ \varepsilon,\ \emptyset)\rangle_{i=1}^{N} \in G_N$, i.e., every machine is in its initial state with an empty mailbox, and its mapping from variables to process identifiers is empty.

In order to use the mailbox semantics, we define a partial function that is defined only if a message may be accepted at the moment. As in Erlang, we look for each message that's in the mailbox in order, and only try the next message if no currently accepting pattern matches. While checking each message, patterns are checked in the defined order and only the first one that matches will be accepted.

Definition 2.8 (Pick function)**.** We implictly assume a CAA $(S,\ o,\ F,\ \mathscr{L}_!,\ \mathscr{L}_?,\ \delta,\ <)$ as our context. Then, the partial function $pick(s,m,v) = \langle pat, \gamma\rangle$ is defined if and only if:

- For all $pat' \in \mathscr{L}_?$, if $\delta(s, ?pat)$ is defined, then for all $v_k \in m$, we have that $v_k \rhd pat = \bot$;
- For all $pat' <_s pat$, we have $v \rhd pat' = \bot$; and
- There is a γ such that $v \rhd pat = \gamma$.

Finally, we now can define our semantics through a step relation, which defines what happens to a global state once one of the current possible transitions is performed.

Definition 2.9 (Step relation)**.** Given a protocol $\langle (S_i,\ o_i,\ F_i,\ \mathscr{L}_{!i},\ \mathscr{L}_{?i},\ \delta_i, <_i)\rangle_{i=1}^{N}$, the relation[1] *step* denotes the non-deterministic transitions of the overall concurrent system, of type $step : G_N \to \mathscr{P}(G_N)$. We write $step(c_1) \ni c_2$ if c_2 is amongst the possible outcomes of $step(c_1)$, as defined by the following two rules:

$$\frac{\delta_i(s_i, P!t) = s_i' \qquad \gamma_i(P) = j \qquad \gamma_i(t) \to^* v}{step(\langle (s_i, m_i, \gamma_i)\rangle_{i=1}^{N}) \quad \ni \quad \langle \ldots, (s_i', m_i, \gamma_i), \ldots, (s_j, m_j \cdot [v], \gamma_j), \ldots\rangle} \text{ (SEND)}$$

$$\frac{\delta_j(s_j, ?pat) = s_j' \qquad m_j = m \cdot [v] \cdot m' \qquad pick(s_j, m, v) = \langle pat, \gamma\rangle}{step(\langle (s_i, m_i, \gamma_i)\rangle_{i=1}^{N}) \quad \ni \quad \langle \ldots, (s_j', m \cdot m', \gamma_j \cup \gamma), \ldots\rangle} \text{ (RECV)}$$

[1]Notice *step* can be equivalently thought of as a non-deterministic function (producing many possible global states) or as a relation from a single input global state to many outcome global states. The definition here declaratively defines this relation.

The first rule, (SEND), actions a *send* transition with label $P!t$ for a process i in state s_i resulting in the state s'_i. We lookup the variable P from the process identifier environment γ_i to get the process identifier we are sending to, i.e., $\gamma_i(P) = j$, which means we are sending to process j. Notice that as γ_i is a finite map, $\gamma_i(P) = P$ if P is not in the domain of γ_i. By abuse of notation, $\gamma(t)$ replaces occurrences of variables of γ in t, allowing for the use of internal state, and then we evaluate the resulting term to a value v (which is denoted by the reduction relation $\rightarrow^*$). Subsequently in the resulting global configuration, process i is now in state s'_i and process j has the message v enqueued onto the end of its mailbox in its configuration, while the states for any other processes is kept the same.

The second rule, (RECV), actions a *receive* transition with label *pat* for a process j in state s_j resulting in the process j being in state s'_j under the condition that the mailbox m_j contains a message v at some point with prefix m and suffix m'. We use the *pick* function to check the semantic constraints: we want to pick the first message v for which some possible pattern matches, and the first one that does so. If this is the correct pattern, given state, prefix and value, then the value term v will unify with *pat* to produce an binding γ. Subsequently, process j is now in state s'_j with v removed from its mailbox and its process identifier environment updated to $\gamma_j \cup \gamma$: its internal state gets updated by any terms matched while receiving the message. Any states for processes other than j remains the same.

Following that, a way to describe a possible result for computation is through a trace, which represents a sequence of steps from the initial global state into one possible final state (as *step* is non-deterministic, several outcomes are possible). We proceed by formally defining a notion of trace.

Definition 2.10 (Trace)**.** A *trace* is a sequence of system configurations such that the first one is the initial global state, and the subsequent states are obtained from applying the *step* relation onto the previous configuration. We call a sequence of global states T a trace if it has the form $T = \langle t_0, t_1, ..., t_x \rangle$ and satisfies the following conditions:

- $t_0 = \langle (o_i, \varepsilon, \emptyset) \rangle_{i=1}^{N}$
- $step(t_i) \ni t_{i+1}$
- $step(t_x) = \emptyset$ where t_x is the last term in the trace sequence.

Example 2.2. Recall the `mem` function from Example 2.1, to which we assign process id 0. We consider a protocol comprises four machines, with three further machines with process identifiers 1, 2 and 3, given by the following definitions:

$$S_1 = \{c_0, c_1, c_2, c_3\} \quad o_1 = c_0 \quad F_1 = \{c_3\} \quad \mathscr{L}_{!1} = \{\texttt{\{get, 1\}}, \texttt{\{put, X+1\}}\} \quad \mathscr{L}_{?1} = \{\texttt{X}\}$$
$$\delta_1(c_0, 0!\texttt{\{get, 1\}}) = c_1 \quad \delta_1(c_1, ?\texttt{X}) = c_2 \quad \delta_1(c_2, 0!\texttt{\{put, X+1\}}) = c_3$$

$$S_2 = \{d_0, d_1, d_2, d_3\} \quad o_2 = d_0 \quad F_2 = \{d_3\} \quad \mathscr{L}_{!2} = \{\texttt{\{get, 2\}}, \texttt{\{put, X+2\}}\} \quad \mathscr{L}_{?2} = \{\texttt{X}\}$$
$$\delta_2(d_0, 0!\texttt{\{get, 2\}}) = d_1 \quad \delta_2(d_1, ?\texttt{X}) = d_2 \quad \delta_2(d_2, 0!\texttt{\{put, X+2\}}) = d_3$$

$$S_3 = \{e_0, e_1\} \quad o_3 = e_0 \quad F_3 = \{e_1\} \quad \mathscr{L}_{!3} = \{\texttt{\{put, 0\}}\} \quad \mathscr{L}_{?3} = \emptyset$$
$$\delta_3(e_0, 0!\texttt{\{put, 0\}}) = e_1$$

The first machine above (process id 1) makes a 'get' request to process 0 then receives the result `X` and sends back to 0 a 'put' message with `X+1`. The second machine above (process id 2) is similar to the first, requesting the value from process 0 but then sending back a 'put' message with `X+2`. The third machine (process id 3) sends to 0 a 'put' message with the initial value 0.

We can then get the following trace for the protocol $\langle (S_i, o_i, F_i, \mathscr{L}_{!i}, \mathscr{L}_{?i}, \delta_i) \rangle_{i=1}^{3}$, one of the many possibilities, demonstrating mobility and the ability to store internal state (we underline the parts of the configuration which have changed at each step of the trace for clarity):

$$
\begin{array}{llll}
\langle(s_0,\varepsilon,\emptyset), & (c_0,\varepsilon,\emptyset), & (d_0,\varepsilon,\emptyset), & (e_0,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,[\{\mathtt{put},0\}],\emptyset)}, & (c_0,\varepsilon,\emptyset), & (d_0,\varepsilon,\emptyset), & \underline{(e_1,\varepsilon,\emptyset)}\rangle \\
\langle\underline{(s_0,\varepsilon,\{S\mapsto 0\})}, & (c_0,\varepsilon,\emptyset), & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,[\{\mathtt{get},1\}],\{S\mapsto 0\})}, & \underline{(c_1,\varepsilon,\emptyset)}, & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_1,\varepsilon,\{S\mapsto 0,P\mapsto 1\})}, & (c_1,\varepsilon,\emptyset), & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,\varepsilon,\{S\mapsto 0,P\mapsto 1\})}, & \underline{(c_1,[0],\emptyset)}, & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle(s_0,\varepsilon,\{S\mapsto 0,P\mapsto 1\}), & \underline{(c_2,\varepsilon,\{X\mapsto 0\})}, & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,[\{\mathtt{put},1\}],\{S\mapsto 0,P\mapsto 1\})}, & \underline{(c_3,\varepsilon,\{X\mapsto 0\})}, & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,[\{\mathtt{put},1\},\{\mathtt{get},2\}],\{S\mapsto 0,P\mapsto 1\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & \underline{(d_1,\varepsilon,\emptyset)}, & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,[\{\mathtt{get},2\}],\{S\mapsto 1,P\mapsto 1\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & (d_1,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_1,\varepsilon,\{S\mapsto 1,P\mapsto 2\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & (d_1,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,\varepsilon,\{S\mapsto 1,P\mapsto 2\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & \underline{(d_1,[1],\emptyset)}, & (e_1,\varepsilon,\emptyset)\rangle \\
\langle(s_0,\varepsilon,\{S\mapsto 1,P\mapsto 2\}), & (c_3,\varepsilon,\{X\mapsto 0\}), & \underline{(d_2,\varepsilon,\{X\mapsto 1\})}, & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,[\{\mathtt{put},3\}],\{S\mapsto 1,P\mapsto 2\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & \underline{(d_3,\varepsilon,\{X\mapsto 1\})}, & (e_1,\varepsilon,\emptyset)\rangle \\
\langle\underline{(s_0,\varepsilon,\{S\mapsto 3,P\mapsto 2\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & (d_3,\varepsilon,\{X\mapsto 1\}), & (e_1,\varepsilon,\emptyset)\rangle
\end{array}
$$

3 Characterising CAA systems: race freedom and convergence

We consider how to characterise race conditions between CAAs and identify a subclass of CAA systems which is race free, or exhibiting *convergence*. To characterise race conditions, we first define the notion of what possible messages can be observed as 'incoming' to a process at a particular step in a trace. We define this notion via multisets of messages:

Definition 3.1 (Multiset of messages in a mailbox)**.** Given a mailbox m, we denote by A_m the multiset of elements in m, i.e., there is a way of 'arranging' the elements of A_m to obtain m.

Definition 3.2 (Incoming messages multiset)**.** Let $G_N = \langle c_1,...,(s_i,m_i,\gamma_i),...,c_N\rangle$ be a global configuration with process i at state s_i, where the next step of the system gives M possible configurations:

$$step(G_N) = \{\langle c_{11},...,(s_{i1},m_{i1},\gamma_{i1}),...,c_{N1}\rangle,...\langle c_{1M},...,(s_{iM},m_{iM},\gamma_{iM}),...,c_{NM}\rangle\}$$

The *incoming message multiset* I_i represents all the possible incoming messages for process i defined:

$$I_i = (\bigcup_{y=1}^{M} A_{m_{iy}}) - A_{m_i}$$

i.e., we take the union of all the possible mailbox multisets $A_{m_{iy}}$ for i obtained after a step is taken, from which is subtracted (multiset difference) the messages in the mailbox of i before that step. That is, the incoming messages I_i are all possible messages that can be added to the mailbox of i after taking a step.

The definition of I is implicitly parameterised by the starting configuration G_N. We will typically superscript I to denote it occurring at some position x in a trace, i.e., I_i^x.

Remark. We use overloaded notation $m_i = \langle I_i^0, I_i^1, ..., I_i^n\rangle$ to represent a mailbox where the first message is drawn from I_i^0, second from I_i^1 and so on. That is, $m_i = \langle l_{0i}, l_{1i}, ..., l_{ni}\rangle$ where $l_{xi} \in I_i^x$.

The mailbox of a process i can now be written as a sequence of multisets of incoming messages, more precisely a subsequence of $\langle I_i^0, I_i^1, ..., I_i^n\rangle$. Thus, $m_i = \langle I_i^{x_0}, I_i^{x_1}, ..., I_i^{x_m}\rangle$, where $x_0 < ... < x_m$. But why a subsequence? If message l_{xi} is consumed, then multiset I_i^x disappears from the mailbox.

Remark (Transition function δ notation overloading). Usually a transition function δ has as its argument a pair of a state and a label. We overload the second part of this pair to allow a multiset such that $\delta(s, ?I_i^x) = \{s' | , \forall l \in I_i^x, \delta(s, ?l) = s'\}$, i.e., the set of target states for any of the possible messages in I_i^x.

Definition 3.3 (Race condition)**.** For a protocol, a *race condition* represents the scenario in which there exists a state that can consume 2 or more messages from the same I_i^x and cannot consume any messages from the previous mailbox sets. That is:

$$\forall y \in [0, x-1].\ |\delta(s, ?I_i^y)| = 0 \wedge |\delta(s, ?I_i^x)| \geq 2$$

Intuitively, if in the current state for a process we can receive 2 or more messages from set I_i^x we are faced with a race condition, since these messages were sent at the same step and could arrive in any order. This is represented by the second part of the above conjunction. However, in order for us to reach I_i^x, we need to not consume any messages before that, hence the first part of the conjunction. If a previous multiset of messages has just one message we can consume, the race condition will not take place.

Trace convergence and race freedom for systems of two automata We consider a class of binary CAA systems (i.e., where $N = 2$) that is race free by showing that its traces always converge, that is, the system is deterministic. Furthermore, this makes the testing of a two automata system much easier, as we need only examine one trace to determine the final state of the system.

We recall that our definition of CAAs allows only for a restricted set of transitions: if a state has a transition, it can only be either some number of receive transition, properly ordered, or a single send transition. In fact, Erlang's semantics is such that transitions cannot be mixed in other ways and should have at most one send transition from any state, a condition we refer to as *affine sends*. We use this condition in order to reason about the possibility of non-determinism in a trace.

In the following, we will assume that *self-messaging* is disallowed: i.e., given any local state (s_i, m_i, γ_i) for $s_i \in S_i$, $\delta(s_i, P!t)$ is undefined for any P such that $\gamma_i(P) = i$[2]. For a class of binary models where no self messaging is allowed we observe (and formally prove below) that there will be no race conditions. A pre-requisite of a race condition (Definition 3.3) is that an actor can receive two incoming messages at the same trace step. By the stated conditions, there can only be one sender of messages at a time and therefore there is at most one transition for every state of an actor, across all global configurations. Thus, if a final state is reached then there will only be one possible trace to it.

To prove that we can have at most one transition at a specific time for an actor, we need to look at the mailbox. Send actions are affine (i.e., deterministic) and so we only need to show that receive transitions are deterministic. The proof considers two possible scenarios where we have two different global configurations for the system:

- Automata in both configurations are in states about to send, or both about to receive. Since both automata are going to perform the same type of action, they will only affect one mailbox; the mailbox configuration cannot diverge in this case.
- One automata is in a state about to send and the other to receive. If the actor in the receive state does not have a valid message to consume, the only valid action is the send (i.e., the other configuration progresses), otherwise both actions would impact the same mailbox making these two configurations diverge. However the send state is a "constructive" action, which will append a message at the end of the mailbox, while the receive action is "destructive" consuming a message from the mailbox, their disjoint nature will result in the convergence of the mailbox.

[2]This restriction is given to avoid both static messages to self, such as in $1!t$, and dynamic ones, such as $P!t$ where P is bound to 1. A more strict approach would be to require that each process only sends static messages to each other.

Proposition 3.1 (Convergence)**.** *Let $X_1 = (S_1, o_1, F_1, \mathscr{L}_{!_1}, \mathscr{L}_{?_1}, \delta_1)$ and $X_2 = (S_2, o_2, F_2, \mathscr{L}_{!_2}, \mathscr{L}_{?_2}, \delta_2)$ form a protocol of two actors, where there are no mixed transitions, only affine sends, and no self messaging. Such a protocol, if it converges to a final state, will do so deterministically, i.e., any global configuration with converge to the same final global configuration, and as a consequence, be race free.*

Proof. We prove our desired goal by showing that any two traces T_1 and T_2 of the same size are the same. We note that a possible sequence of states must follow a pattern $\langle s_0, ..., s_n\rangle$, for some n, and show that for any $k \leq n$, the prefix of size k of T_1 and T_2 is the same. Taking k to be n, they are the same. This proof follows by well-founded induction on k (which has an upper bound):

1. Base case: we take $k = 1$. Both prefixes should then be $\langle s\rangle$, where s is the initial global state. This follows by definition of a trace, as it deterministically specifies which is the first state for any given configuration.
2. Inductive step: our inductive assumption says that our prefixes $\langle s_0...s_k\rangle$ match. If $k = n$, then we are done. If, however, $k \leq n$, then s_k is not a final state and we have that $T_1 = \langle s_0...s_k t_1\rangle$ and $T_2 = \langle s_0...s_k t_2\rangle$. We must now show that $t_1 = t_2$. Since we can only have one other process sending messages, we have that $\forall j \in [1, k+1], |I_i^{j+1}| \leq 1$. Now, since the size of the sets is at most one, consuming a message would nullify the set, therefore we can represents the mailbox as a subsequence of the set $\{I_i^0, I_i^1, .., I_i^k\}$, with $m_i = \langle I_i^{x_1}, I_i^{x_2}, ...I_i^{x_y}\rangle$ where $x_1 < x_2 < ... < x_y < k+1$. We find the greatest z such that $\forall j \in [1, z], |\delta(s_k, ?I_i^j)| = 0$. Let P_i be the set of all possible states after a receive transition. If $j = y$, then $|P_i| = 0$, so no messages can be consumed and we stay put at the same state, otherwise $P_i = \{s' | \forall m \in I_i^{z+1}, \delta(s_k, ?m) = s'\}$. Since we can receive at most one message, $|I_i^{z+1}| \leq 1$ therefore $|P_i| \leq 1$, having at most one possible transition, so just one possible new state. This can't be zero as we have t_1 and t_2: thus the list of possible next states has only one member, and $t_1 = t_2$ as expected.

□

Compatibility in Erlang Early work on CFSMs identified the notion of *compatibility* between machines as a key step towards guaranteeing progress of systems [14]. Compatibility is the automata analogue of what is commonly known as duality: that every receive has a corresponding send and vice versa. A pair of compatible, deterministic machines is then free from deadlock and unspecified receptions [14].

Due to Erlang's design principles, we might not always care about ending up with an empty mailbox and admit systems as being compatible even if some messages are not received, leaving remaining messages in the mailbox. For future work we thus propose 'tiers' of final conditions on a system which characterise, roughly, different levels of compatibility.

In the following, let the set of all possible traces for a system of CAAs be Y.

Definition 3.4. (Tier 1) A system is *strongly compatible* iff, $\forall T \in Y, t_{|T|} = \langle (f_i, \varepsilon, \gamma_i)\rangle_{i=1}^N$ where $f_i \in F_i$, i.e., no process has any message left in their mailboxes and they have reached final states.

Definition 3.5. (Tier 2) A system is *weakly compatible* iff, $\forall T \in Y, t_{|T|} = \langle (f_i, m_i, \gamma_i)\rangle_{i=1}^N$ where $f_i \in F_i$, i.e., some processes may have some messages left in their mailbox, but all have reached final states.

Definition 3.6. (Tier 3) A system is *communication-lacking* iff, $\forall T \in Y, t_{|T|} = \langle (s_i, \varepsilon, \gamma_i)\rangle_{i=1}^N$ where $s_i \in S_i/F_i$, i.e., processes didn't reach their final states but can't continue because of lack of input.

Definition 3.7. (Tier 4) If none of the previous conditions are met, the system is said to be *incompatible*.

We remark that the system in Example 2.2 is strongly compatible, as all possible traces will end up with empty mailboxes and in final states.

4 Discussion and Related work

Fowler describes a framework for generating runtime monitors for Erlang/OTP's `gen_server` behaviours from multiparty session types as conceived of in the Scribble language [12]. This leverages the idea, due to Deniélou and Yoshida [10] of projecting Scribble's global types (multiparty session types) into local types, and then implementing local types as CFSMs. It is not clear however to what extent this models Erlang's general mailbox semantics. This warrants a further investigation.

A classic mantra of Erlang is to "let it crash" (or "let it fail"). Our model here does not deal with process failure, although recent models have incorporated such aspects [4]. In the model of Bocchi et al. [4], actors communicate via unidirectional links to their mailboxes, similar to the structure of CFSMs, but with increased flexibility in the way that steps occur. The approach doesn't integrate pattern matching or dynamic topologies. Mailbox MSCs (Message Sequence Charts) [5, 6], build a message sequent chart model of processes with a single incoming channel and matching semantics similar in philosophy to our model but not directly based in the CFSM tradition. A deeper comparison with our approach is further work. One considerable difference in our approach is the integration of dynamic topologies by the 'memory' environment γ for each process, which enables messages to be sent to a variable which is a process identifier bound by a preceding receive.

In preliminary work, we have created a tool for extracting a CAA model from the Erlang code, and also the other way around, generating an Erlang skeleton of concurrent communicating code from a description of a CAA protocol. Further work includes developing this into a tool for analysis and specification of Erlang programs. We also note that due to the possibility to describe mobile processes, we conjecture that Milner's CPS translations from the λ-calculus into the π-calculus could be adapted to use CAAs as a target language. If that's the case, then it follows that CAAs describe a Turing-complete model of computation, and we intend to investigate this possibility.

Acknowledgments With thanks to Kartik Jalal for insights coming from model extraction of CAAs from Erlang, Simon Thompson for his Erlang expertise in the early days of this idea, and University of Kent undergraduates taking *CO545: Functional and Concurrent Programming* between 2017-20 for being a test audience for this model and its use in diagnosing race conditions and deadlocks. This work was partly supported by EPSRC project EP/T013516/1 (Granule). Orchard is also supported in part by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program.

References

[1] Joe Armstrong (1997): *The development of Erlang*. In: *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pp. 196–203, doi:10.1145/258948.258967.

[2] Joe Armstrong (2007): *A history of Erlang*. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 6–1, doi:10.1145/1238844.1238850.

[3] Samik Basu, Tevfik Bultan & Meriem Ouederni (2012): *Deciding choreography realizability*. *ACM SIGPLAN Notices* 47(1), pp. 191–202, doi:10.1145/2103621.2103680.

[4] Laura Bocchi, Julien Lange, Simon Thompson & A. Laura Voinea (2022): *A Model of Actors and Grey Failures*. In Maurice H. ter Beek & Marjan Sirjani, editors: *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, *Lecture Notes in Computer Science* 13271, Springer, pp. 140–158, doi:10.1007/978-3-031-08143-9_9.

[5] Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Etienne Lozes & Amrita Suresh (2021): *A unifying framework for deciding synchronizability*. In: *CONCUR 2021-32nd International Conference on Concurrency Theory*, pp. 1–33, doi:10.4230/LIPIcs.CONCUR.2021.14.

[6] Ahmed Bouajjani, Constantin Enea, Kailiang Ji & Shaz Qadeer (2018): *On the completeness of verifying message passing programs under bounded asynchrony*. In: *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, Springer, pp. 372–391, doi:10.1007/978-3-319-96142-2_23.

[7] Daniel Brand & Pitro Zafiropulo (1983): *On communicating finite-state machines*. *Journal of the ACM (JACM)* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[8] Richard Carlsson (2001): *An introduction to Core Erlang*. In: *Proceedings of the PLI*, 1, Citeseer.

[9] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson & Robert Virding (2000): *Core Erlang 1.0 language specification*. *Information Technology Department, Uppsala University, Tech. Rep.*

[10] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty session types meet communicating automata*. In: *European Symposium on Programming*, Springer, pp. 194–213, doi:10.1007/978-3-642-28869-2_10.

[11] Alain Finkel & Etienne Lozes (2017): *Synchronizability of communicating finite state machines is not decidable*. *arXiv preprint arXiv:1702.07213*, doi:10.48550/arXiv.1702.07213.

[12] Simon Fowler (2016): *An Erlang implementation of multiparty session actors*. *arXiv preprint arXiv:1608.03321*, doi:10.4204/EPTCS.223.3.

[13] Mohamed G Gouda (1984): *Closed covers: to verify progress for communicating finite state machines*. *IEEE transactions on software engineering* (6), pp. 846–855, doi:10.1109/TSE.1984.5010313.

[14] Mohamed G Gouda, Eric G Manning & Yao-Tin Yu (1984): *On the progress of communication between two finite state machines*. *Information and control* 63(3), pp. 200–216, doi:10.1016/S0019-9958(84)80014-5.

[15] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In Sriram K. Rajamani & David Walker, editors: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, ACM, pp. 221–232, doi:10.1145/2676726.2676964.

[16] Jan Pachl (2003): *Reachability problems for communicating finite state machines*. *arXiv preprint cs/0306121*, doi:10.48550/arXiv.cs/0306121.

[17] Wuxu Peng & S Purushothaman (1992): *Analysis of a class of communicating finite state machines*. *Acta Informatica* 29(6-7), pp. 499–522, doi:10.1007/BF01185558.

[18] Louis E Rosier & Mohamed G Gouda (1984): *Deciding progress for a class of communicating finite state machines*.