A Formal specification For Half a Century of Actor Systems

JOERI DE KOSTER, Vrije Universiteit Brussel, Belgium
WOLFGANG DE MEUTER, Vrije Universiteit Brussel, Belgium

The Actor Model is a message passing concurrency model that was originally proposed by Hewitt et al. in 1973. Half a century later a plethora of variations on this model have been explored for various programming languages and systems. So much so that precise definition of actor-based programming languages is lost and that the term *actor* has become highly conflated. The goal of this paper is to disambiguate different actor models by classifying them into four families, namely: *Classic Actors, Active Objects, Processes and Communicating Event Loops*. This paper identifies and defines the *isolated turn principle* as the key and unifying principle among all actor models. This paper provides a formal definition of the core subset of all four families of actor models by means of an operational semantics and proves that the isolated turn principle holds for all of them.

ACM Reference Format:
Joeri De Koster and Wolfgang De Meuter. 2023. A Formal specification For Half a Century of Actor Systems. 1, 1 (August 2023), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Actor model of computation is a model for concurrent computation that treats *actors* as the universal primitives of concurrent computation. Actors are objects that encapsulate both state and behavior, and can communicate with each other asynchronously by sending and receiving messages. The Actor model was first proposed by Carl Hewitt in 1973 [18] as a way to understand and reason about the behavior of large, concurrent systems.

The invention of the Actor Model is rooted in the research on object-oriented programming languages. Motivated to discover novel modularisation techniques to develop complex systems, the early sixties was a hotbed for formative ideas around object-oriented programming. The first widely recognised object-oriented programming language was specified in that same era (Simula, in 1965). Only a few years later, the Planner language was developed as part of Carl Hewitt's doctoral research at MIT's Artificial Intelligence Lab [17]. Planner introduced the notion of procedural embedding of knowledge. Influenced by ideas from object-oriented programming and his work on Planner, Hewitt and his graduate students invented the actor model of computation in 1973 [18].

One of the key features of the Actor model is that each Actor is completely self-contained and has its **own private state**. This means that an Actor's internal state cannot be directly accessed or modified by other Actors,

Authors' addresses: Joeri De Koster, Joeri.De.Koster@vub.be, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, 1050, Belgium; Wolfgang De Meuter, wdemeuter@vub.ac.be, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, 1050, Belgium.

only **messages** can be sent to it. This makes it easy to reason about the behavior of an Actor, as its behavior is determined solely by the messages that it receives and the internal state that it maintains.

Another important aspect of the Actor model is that all communication between Actors is **asynchronous**. This means that when an Actor sends a message to another Actor, it does not wait for a response, but instead continues to process other messages. This allows for high parallelism and scalability, and reduces the likelihood of contention and deadlock, as Actors can continue to process messages even if other Actors are blocked or slow.

Because of the isolation of state and the asynchronicity of its communication mechanism, the Actor Model is guaranteed to be free of low-level data races and deadlocks. This is a key property that unifies all actor model implementations.

However, there are also some challenges in using the Actor model in practice. One is that it can be difficult to reason about the global behavior of a system, since it is often difficult to see the interactions between different actors. Another is that it can be difficult to debug and test systems built using the Actor model since it can be hard to reproduce the same sequence of messages and interactions that led to a particular error or bug.

Despite these challenges, the Actor model has proven to be a powerful and versatile tool for building concurrent and distributed systems. Over the past 50 years, a plethora of programming languages that implement their own variation on the actor model have been developed. Each of these variations on the actor model comes with their own terminology, programming language concepts and run-time semantics. So much so, that the term *actor* has become so conflated that some researchers refrain from using it in their papers [10].

There are many properties and features along which different actor programming languages can be classified. In this paper we focus solely on those that impact their programming model. This paper disambiguates different actor models by classifying them into four families, each with their own programming model, namely: *Classic Actors, Active Objects, Processes and Communicating Event Loops*. Section 2 provides a brief overview of the early history of the actor model. From Section 3 to Section 6, we highlight the key differences between each of the four families and give a formal definition for each by means of an operational semantics. An executable version of each of these semantics in PLT Redex is also available[1]. We define the *isolated turn principle* as the key principle for actors and we prove that it holds for each of the four families of actor models.

2 EARLY HISTORY OF THE ACTOR MODEL

The actor model was first introduced in a 1973 paper authored by Carl Hewitt and two of his graduate students: Peter Bishop and Richard Steiger [18]. The original goal was to have a programming model for safely exploiting concurrency in distributed workstations. The problem domain was modelling parallel communication based problem solvers. The first implementation of the actor model was in a Planner-like programming language that was modeled on actors [15], originally called Planner-73, but later renamed to PLASMA.

In PLASMA, actors communicate with each other via message passing which consists of sending a request from one actor (called the messenger) to another actor (called the target). The request and a reference to the messenger are packaged as an envelope and put into the inbox of the target actor (`request: message; reply-to: messenger`). Given that envelope, the behaviour of the target actor then specifies how the computation continues with respect to the request. The messenger is typically used as the reply address to which a reply to the request should be sent. The simplest control structure that uses this request-reply pattern in most programming

[1] https://gitlab.soft.vub.ac.be/jdekoste/actormodelhistorypltredex

languages is the procedure call and return. A recursive implementation of *factorial* written in PLASMA is given in Listing 1.

```
(factorial ≡
  (≡> [=n]
    (rules n
      (≡> 1
          1)
      (≡> (> 1)
          (n * (factorial <= (n - 1)))))))
```

Listing 1. Factorial function written in PLASMA.

In this example factorial is defined to be an actor of which the behaviour matches the requests of incoming envelopes with one element which will be called `n`. The rules for `n` are, if it is `1`, then we send back `1` to the messenger of the envelope. Note that this is done implicitly. If it is greater than `1`, we send a message to the factorial actor to recursively compute the factorial of `(n - 1)`.

In December 1975, in an attempt to understand the Actor Model described by Hewitt, Sussman and Steele [32] wrote a continuation-based interpreter for a Lisp-like language called Scheme. They came to the conclusion that Hewitt's "actors" were very similar to scheme lambda expressions and had their roots in the lambda calculus [12]. In effect, sending a message to an actor that is in an idle state is very similar to invoking a continuation.

Throughout the history of the actor model, a plethora of different variations of the actor model have been designed. This paper taxonomises these variations of the actor model into four families, namely: Classic Actors, Active Objects, Processes and Communicating Event Loops. The following four sections give an overview of the history for each of these four families and provides an operational semantics for the core set of language features of each variation.

3 THE CLASSIC ACTOR MODEL

The Actor Model only became more widely regarded as a general-purpose concurrency model in 1986, when it was recast in terms of three simple primitives by one of Carl Hewitt's former PhD students, namely Gul Agha [1, 2]. Agha redefined the Actor Model in terms of three basic actor primitives: *create*, *send* and *become*. His vision of the Actor Model laid the foundations for a host of different other actor systems and these three primitives can still be found in various modern actor languages and libraries today.

The main focus of his work was to produce a platform for distributed problem solving in networked workstations. In his model concurrent objects, i. e. actors, are self-contained, independent components that interact with each other by asynchronous message passing. In his work he presents three basic actor primitives:

- `create`: Creates an actor from a behaviour description. Returns the address of the newly created actor.
- `send`: Asynchronously sends a message from one actor to another by using the address of the receiver. Immediately returns and returns nothing.
- `become`: Replaces the behaviour of an actor. The next message that will be received by that actor is processed by the new behaviour.

```
(define Cell
  (mutable [content]
    [put [newcontent]
      (become Cell newcontent)]
    [get
      (return 'got-content content)]))

(define my-cell (create Cell 0))
(get my-cell)
```

Listing 2. An actor in Rosette.

Definition: Classic Actor Model

> The Classic Actor Model formalises the Actor Model in terms of three primitives: `create`, `send` and `become`. The sequential subset of actor systems that implement this model is typically functional. Changes to the either or both the state and interface of an actor are aggregated in a single become statement.

The example in Listing 2 is written in the Rosette actor language [33] which was based on this model.

The `mutable` form is used to create an *actor generator* that is bound to `Cell`. That generator can be used with the `create` form to create an instance of that actor. Each actor instance has its own mailbox and behaviour. Following the keyword `mutable` is a sequence of identifiers that specify the mutable fields of that actor. In our example, any `Cell` actor will have one mutable field, namely the content of that cell. After that is a specification of all the messages that are understood by the actor. A message is specified by a message name followed by a table of arguments. In this case the `put` message expects a value for the new content. Afterwards follows the body that specifies how each message should be processed. If one wishes to modify the state of a mutable field one can use the `become` form to replace the behaviour of an actor using the actor generator. The `return` form is used to implicitly send back the result of a computation to the sender of the original message.

These three primitives are the basic building blocks for many actor systems today and have been very influential in the development of any actor language that follows this work. A modern implementation of the Actor Model based on Agha's work[2] is the Akka [4] actor library for Scala. However, there are many other library implementations of this model for different languages such as Smalltalk (Actalk [8]) and C++ (ACT++ [21], Broadway [31] and Thal [23]).

The sequential subset of an actor model is the subset of expressions out of which a behaviour can be composed. In the case of the Classic Actor Model this sequential subset is mostly functional. Any state changes are specified by replacing the behaviour of an actor. This has an important advantage over conventional assignment statements as this severely coarsens the granularity of side-effecting operations that need to be considered when analysing a system. On the one hand, an actor can only change its own behaviour, meaning that the state of each actor is fully isolated. On the other hand, changing the behaviour of an actor only comes into effect when processing the next message. This means that the processing of a single message can be regarded as a single isolated operation. Throughout the rest of this paper we refer to this principle as **the Isolated Turn Principle**. This mechanism allows state updates to be aggregated into a single become statement and significantly reduces the amount of control flow dependencies between statements.

3.1 The Isolated Turn Principle

The semantics of the Classic Actor Model enables a *macro-step semantics* [3]. With the macro-step semantics, the Actor Model provides an important property for formal reasoning about program semantics, which also provides additional guarantees to facilitate application development. The macro-step semantics says that in an Actor Model, the granularity of reasoning is at the level of a **turn**, i. e., an actor processing a single message from its mailbox. A turn starts when the actor retrieves the message from its mailbox and ends when that message is fully processed. A single turn can be regarded as being processed in a single isolated step. The Isolated Turn Principle leads to a convenient reduction of the overall state-space that has to be considered in the process of formal reasoning. Furthermore, this principle is directly beneficial to application programmers, because the amount of processing done within a single turn can be made as large or as small as necessary, which reduces the potential for problematic interactions. In other words, this principle guarantees that, during a single turn, an actor has a consistent view about its state and its environment.

To satisfy this principle, an actor system must satisfy both safety and liveness properties:

Safety. To satisfy safety the state of an actor must be *fully isolated*. This property is mainly guaranteed by adopting a *no-shared-state* policy between actors. Any composite value that is transmitted across actor boundaries is either copied, proxied or immutable. This property ensures that the processing of a single message in the Actor Model is free of low-level data races. In addition, the processing of a message cannot be interleaved with the processing of other messages of the same actor unless the execution of those different messages is also fully isolated. For example, an actor for which the behaviour was modified can already act on other incoming messages before fully processing the current message. Or implementations of the actor model can enable parallel execution of read only messages [28] without impacting safety guarantees.

Liveness. To guarantee liveness, the processing of a message cannot contain any blocking operations. Any message is always entirely processed from start to finish. Because of this property, processing a single message is free of deadlocks.

From the two properties above follows that if an Actor Model satisfies the Isolated Turn Principle, it is free of low-level data races and deadlocks. However, this only applies for the processing of a single message, considering the processing of several messages, these properties no longer hold. On the one hand, as the actor model only guarantees isolation within a single turn, high-level race conditions can still occur with bad interleaving of different messages. The general consensus when programming in an actor system is that when an operation spans several messages the programmer must provide a custom synchronisation mechanism to prevent potential bad interleavings and ensure correct execution. On the other hand, high-level deadlocks can still occur when actors are waiting on each other to send a message before progress can be made.

3.2 Operational Semantics of the Classic Actor Model

In this section we present a minimal operational semantics for the Classic Actor Model. An executable implementation in PLT Redex of this operational semantics can be found online[2]. The goal of this operational semantics and the operational semantics presented in subsequent sections is to provide a precise formal definition for each of the different actor families and to highlight their key differences. We borrow some terminology of

[2]https://gitlab.soft.vub.ac.be/jdekoste/actormodelhistorypltredex

object-oriented programming to remain consistent with our usage of terminology throughout this paper. More concretely, we use objects to represent the behaviour of an actor. An actor's behaviour is composed of its interface (i. e. the set of messages it understands) and its state. Both of these can be represented by an object, the interface of an actor's behaviour is represented by the class of the corresponding object and the state by that object's values for its fields.

3.2.1 Semantic Entities.

$$
\begin{array}{rcl}
k \in \textbf{Configuration} & ::= & A \\
a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}\langle \iota_a, \mu, e, o \rangle \\
\mu \in \textbf{Mailbox} & ::= & \overline{m} \\
m \in \textbf{Message} & ::= & \mathcal{M}\langle msg, \overline{v} \rangle \\
o \in \textbf{Object} & ::= & \mathcal{O}\langle cls, \overline{v} \rangle \\
v \in \textbf{Value} & ::= & r \mid \texttt{null} \\
r \in \textbf{Reference} & ::= & \iota_a
\end{array}
$$

$$\iota_a \in \textbf{ActorId}$$

Fig. 1. Semantic entities of the Classic Actor Model

Figure 1 lists the semantic entities for the Classic Actor Model. Caligraphic letters like $\mathcal{A}$ and $\mathcal{M}$ are used as *constructors* to distinguish the different semantic entities syntactically instead of using bare cartesian products.

In the Classic Actor Model a **Configuration**, K, consists of a set of running actors, A. A configuration represents the whole state of a program in a single step. Each **Actor**, a, is represented by an identifier, ι_a, a mailbox, μ, the expression it is currently evaluating, e, and an object that represents its behaviour, o. The object that represents an actor's behaviour represents both that actor's interface as well as its state. Classic Actors can change their behaviour using a `become` statement that replaces this object, thereby changing either their interface, their state, or both. A **Mailbox**, μ, is an ordered list of pending messages. A **Message**, m, has a message name, msg, that acts as a selector to invoke the correct method of the behaviour of the receiving actor and an ordered list of values, $\overline{v}$ that represent the arguments to the message. An **Object**, o, has a classname cls that defines the actor's interface and a set of fields, $\overline{v}$ that represent the actor's state. For simplicity, we have restricted the set of possible values to either be a reference to an actor, r, or the null value, `null`. A **Reference** is always represented by an Actor's Id, ι_a.

3.2.2 Syntax.

Figure 2 lists the syntax, evaluation contexts and initial configuration for Classic Actor programs.

Syntax. A Classic Actor **Program**, p, is a set of class definitions, C, of which one class is the $Main$ class that has to implement the *run* method. A **Class** has a classname, cls, a list of attribute variables, $\overline{x_f}$ and a set of methods, M. Each **Method** has a methodname, msg, and an associated lambda, $\lambda\overline{x_m}.e$, for which the body consists of a single expression, e. The expressions of the Classic Actor Model features a minimal set of syntactical elements. Local variables can be introduced using a *let* expression. The pseudovariable $self$ always refers to the enclosing actor. Expression sequences, $e\,;\,e$ are only introduced for convenience and are syntactic sugar for

nested *let* expressions. Finally, it has syntax for the three main primitives of the Classic Actor model, namely: `spawn`, `send`, and `become`. `spawn` creates a new actor from the behaviour defined by the class with classname *cls* and initialises it's attributes, `send` is used to send a message to an actor, and `become` is used to replace the behaviour of an actor (i. e. its interface can be replaced with a different class and its state can be replaced with different values).

Syntax

$$
\begin{array}{rcl}
p \in \textbf{Program} & ::= & C \\
C \subseteq \textbf{Class} & ::= & \texttt{class}\ cls\ \{\overline{x_f}; M\} \\
M \subseteq \textbf{Method} & ::= & msg \rightarrow \lambda\overline{x_m}.e \\
e \in \textbf{Expression} & ::= & \texttt{self} \mid x \mid \texttt{null} \mid e\ ;\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \\
 & & \texttt{spawn}(cls, \overline{e}) \mid \texttt{send}(e, msg, \overline{e}) \mid \texttt{become}(cls, \overline{e})
\end{array}
$$

$x \in \textbf{VarName}, cls \in \textbf{ClassName}, msg \in \textbf{MessageName}$

Runtime Syntax

$$e ::= \ldots \mid r$$

Evaluation Contexts

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \square \mid \texttt{let}\ x\ =\ \mathcal{E}\ \texttt{in}\ e \mid \texttt{spawn}(cls, \overline{v}, \mathcal{E}, \overline{e}) \mid \texttt{send}(\mathcal{E}, msg, \overline{e}) \mid \\
 & & \texttt{send}(v, msg, \overline{v}, \mathcal{E}, \overline{e}) \mid \texttt{become}(cls, \overline{v}, \mathcal{E}, \overline{e})
\end{array}
$$

Syntactic Sugar

$$e\ ;\ e' \stackrel{\text{def}}{=} \text{let}\ x = e\ \text{in}\ e' \qquad x \notin \text{FV}(e')$$

Initial Configuration

$$K_{init} = \{\mathcal{A}\langle \iota_a, \varnothing, \texttt{send}(\iota_a, run, \varnothing), \mathcal{O}\langle Main, \varnothing\rangle\rangle\}$$

Fig. 2. Syntax of the Classic Actor Model

Runtime Syntax. Our reduction rules operate on so-called run-time expressions; these are a superset of source-syntax phrases. The additional form represents references, r, that can be used as return values for the aforementioned primitives.

Evaluation Contexts. We use evaluation contexts [14] to indicate what subexpressions of an expression should be fully reduced before the compound expression itself can be further reduced. $\mathcal{E}$ denotes an expression with a "hole". Each appearance of $\mathcal{E}$ indicates a subexpression with a possible hole. The intent is for the hole to identify the next subexpression to reduce in a compound expression.

Initial Configuration. For any program, p, the set of class definitions, C is constant. We therefore do not include it as part of the run-time configuration. The initial configuration is always a singleton set with the single main actor. The main actor is initialised with an empty mailbox, a single `send` expression to self-send the *run* method, and an object that is an instance of the *Main* class. That object represents the main actor's interface.

3.2.3 Reduction Rules.

Notation. A configuration, A, is a set of actors. To lookup and extract values from a set we use the notation $A = A' \uplus \{a\}$. This splits the set A into a singleton set containing the desired actor a and the disjoint set $A' = A \setminus \{a\}$. The notation $\mu = \mu' \cdot \mathrm{m}$ deconstructs a sequence μ into a subsequence μ' and the last element m. We denote both the empty set and the empty sequence using $\varnothing$. The notation $\mathcal{E}[e]$ indicates that the expression e is part of a compound expression $\mathcal{E}$, and should be reduced first before the compound expression can be reduced further. We use the notation $[v/x]e$ to denote a variable substitution where all occurrences of x in e are replaced by v.

The reduction rules are split into two layers. The *actor-local reduction rules* ($\longrightarrow_a$) in Figure 3 define the rules that reduce an actor's expression an that do not involve any additional actors (i. e. the expression can be reduced in isolation). The *actor-global reduction rules* ($\longrightarrow_k$) in Figure 4 define rules that involve multiple actors.

(LET)
$$\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{let}\ x = v\ \texttt{in}\ e], o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[[v/x]e], o\rangle$$

(BECOME)
$$\frac{o' = \mathcal{O}\langle cls, \overline{v}\rangle}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{become}(cls, \overline{v})], o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{null}], o'\rangle}$$

(SELF-SEND)
$$\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{send}(\iota_a, msg, \overline{v})], o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu \cdot \mathcal{M}\langle msg, \overline{v}\rangle, \mathcal{E}[\texttt{null}], o\rangle$$

(RECEIVE)
$$\frac{o = \mathcal{O}\langle cls, \overline{v_f}\rangle \qquad \texttt{class}\ cls\ \{\overline{x_f}; M\} \in C \qquad e, \mu' = \text{match}(M, \mu)}{\mathcal{A}\langle \iota_a, \mu, v, o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu', [\iota_a/\texttt{self}][\overline{v_f}/\overline{x_f}]e, o\rangle}$$

Fig. 3. Local Reduction Rules for Classic Actor Model

Actor-local reductions. Actors operate by perpetually taking the first message that matches their interface from their mailbox and then evaluating (reducing) the associated expression to a value. Messages are matched from right to left. Classic Actors have out of order message processing, if a message does not currently match the actor's interface, it is simply skipped and remains in the actor's mailbox. Classic Actors also have a flexible interface (that can be changed using a `become` expression), this facilitates what is known as "conditional synchronisation" [9] (e.g. implementing a blocking bounded buffer, or other more complex forms of synchronisation).

If no reduction rule is applicable to further reduce a reducible expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its current expression is fully reduced to a value and no messages in its mailbox matches with its current interface. A value cannot be further reduced and the actor sits idle until it receives a new message that does match its interface.

We now explain the actor-local reduction rules ($\longrightarrow_a$) found in Figure 3:

- LET: Reducing a "let"-expression simply substitutes the value of x for v in e.
- BECOME: This rule describes how actors can change their behaviour using the `become` primitive. A new object is created and entirely replaces the original behaviour of that actor. The become expression itself reduces to `null`.

- SELF-SEND: An asynchronous message sent to the same actor simply appends a new message to the end of that actor's own mailbox. The message send itself immediately reduces to `null`.
- RECEIVE: This rule describes the processing of messages in the mailbox of an actor. A new message can be processed only if two conditions are satisfied: Firstly, one of the messages in the mailbox of the actor matches its interface, and secondly, the current expression of the actor cannot be reduced any further (the expression is a value, v). To match a message from the actor's mailbox, μ, with that actor's interface, M, the auxiliary function `match` is used. This function returns the body expression. e, of the matched method and an updated mailbox, μ, where the matched message was removed.

$$\begin{array}{l}\text{(SEND)}\\ \quad A \uplus \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathsf{send}(\iota_{a'}, msg, \overline{v})], o\rangle\} \uplus \{\mathcal{A}\langle \iota_{a'}, \mu', e', o'\rangle\}\\ \longrightarrow_k A \cup \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathtt{null}], o\rangle\} \cup \{\mathcal{A}\langle \iota_{a'}, \mu' \cdot \mathcal{M}\langle msg, \overline{v}\rangle, e', o'\rangle\}\end{array}$$

$$\text{(SPAWN)}\quad \frac{\iota_{a'} \text{ fresh} \qquad o' = \mathcal{O}\langle cls, \overline{v}\rangle}{\begin{array}{l}A \uplus \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathsf{spawn}(cls, \overline{v})], o\rangle\}\\ \longrightarrow_k A \cup \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_{a'}], o\rangle\} \cup \{\mathcal{A}\langle \iota_{a'}, \varnothing, \mathtt{null}, o'\rangle\}\end{array}}$$

$$\text{(CONGRUENCE)}\quad \frac{a \longrightarrow_a a'}{A \uplus \{a\} \longrightarrow_k A \cup \{a'\}}$$

Fig. 4. Global Reduction Rules for Classic Actor Model

Actor-global reductions. We now explain the actor-global reduction rules ($\longrightarrow_k$) found in Figure 4:

- SEND: An asynchronous message sent to a different actor appends that message to the front of the mailbox of the recipient actor (the actor with identifier $\iota_{a'}$). The send expression itself reduces to `null`.
- SPAWN: Reducing a spawn expression adds a new actor to the set of actors of the configuration. The newly created actor is initialised with an empty mailbox, `null` as its expression (the actor will be in an idle state), and a newly created object as its behaviour. The spawn expression itself reduces to a reference to that newly created actor, $\iota_{a'}$.
- CONGRUENCE: this rule connects the actor-local reduction rules to the global configuration reduction rules.

$$\begin{array}{rcll}\text{match}(M, \mathcal{M}\langle msg, \overline{v}\rangle \cdot \mu) & \stackrel{def}{=} & [\overline{v}/\overline{x_m}]e, \mu & \text{if } msg \to \lambda\overline{x_m}.e \in M\\ \text{match}(M, m \cdot \mu) & \stackrel{def}{=} & e, m \cdot \mu' & \text{if } e, \mu' = \text{match}(M, \mu)\end{array}$$

Fig. 5. Auxiliary Functions

Auxiliary functions. The auxiliary function `match` finds the first message (going from left to right) in the mailbox, μ, that matches the selector msg. To find a match, the mailbox cannot be empty ($\mu \neq \varnothing$). It has two return values: the body expression of the associated lambda where all parameters, $\overline{x_m}$ have been substituted by the arguments of the message, $\overline{v}$, and and updated mailbox where the selected message has been removed.

3.3 Proof that the Isolated Turn Principle holds for the Classic Actor Model

In this section we provide a formal proof that the Isolated Turn principle holds for the Classic Actor Model. For this property to hold, an actor system must satisfy both safety and liveness for the execution of a turn. For each actor, a new turn starts when a message is retrieved from its mailbox and ends when that message is fully processed (See Section 3.1). In addition to this informal definition, we start by formally defining what constitues a turn here.

Definition 3.1 (Turn). Let a turn, $\xrightarrow{\iota_a}{}^*_k$, be any sequence of reduction rules such that $A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, o\rangle\} \xrightarrow{\iota_a}{}^*_k A \uplus \{\mathcal{A}\langle \iota_a, \mu', v, o'\rangle\}$ and the (RECEIVE) reduction rule for the actor with id, ι_a, $\notin \xrightarrow{\iota_a}{}^*_k$.

More informally, for any actor, a turn is defined as any sequence of reduction rules that are applied between two applications of the (RECEIVE) reduction rule for that actor. During that turn, reduction rules involving other actors can be applied (i. e. A and A' can differ in the above definition).

THEOREM 3.2 (THEOREM OF SAFETY). *If $A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, o\rangle\} \xrightarrow{\iota_a}{}^*_k A' \uplus \{\mathcal{A}\langle \iota_a, \mu', v, o'\rangle\}$ then all possible values for v are α-equivalent.*

PROOF OF SAFETY. The evaluation contexts defined in Figure 2 determine a strict total order in which compound expressions can be reduced. Therefore, This proof follows by induction over the the reduction of expressions. Cases (BECOME), (SELF-SEND) and (SEND) are trivial as they all reduce to `null`. Case (SPAWN) reduces to a fresh α-equivalent identifier $\iota_{a'}$. Case (LET) follows by induction, if all possible v are α-equivalent then the substitution $[v/x]e$ will also be α-equivalent. Case (CONGRUENCE) follows by induction over the actor-local reduction rules. Case (RECEIVE) does not apply as (RECEIVE) $\notin \xrightarrow{\iota_a}{}^*_k$ using definition 3.1. □

Definition 3.3 (Error). If no reduction rule is applicable to further reduce a reducible expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its current expression is fully reduced to a value and no messages in its mailbox matches with its current interface.

THEOREM 3.4 (THEOREM OF LIVENESS). $\forall K = A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, o\rangle\}, e \notin \textbf{Value}, \exists K' : K \xrightarrow{\iota_a}_k K'$

PROOF OF LIVENESS. This proof follows by induction over the reduction of expressions. Cases (LET), (BECOME), (SELF-SEND), (SEND) and (SPAWN) all operate on reducible expressions and can be applied in any error-free program following definition 3.3. Case (CONGRUENCE) follows by induction over the actor-local reduction rules. Case (RECEIVE) does not apply as (RECEIVE) $\notin \xrightarrow{\iota_a}_k$ using definition 3.1. □

In theorem 3.2 and theorem 3.4 we have shown that classic actors satisfy both safety and liveness for the execution of a single turn and therefore the Isolated Turn Principle holds.

4 THE ACTIVE OBJECT MODEL

Around the same time that Agha reformulated Hewitt's actors in terms of OOP, another PhD student of Carl Hewitt, Yonezawa, worked on a object-oriented concurrent programming language called ABCL/1 [38]. In this language, each object has its own thread of control and may have its own local mutable memory. In this model state changes are not specified in terms of behaviour updates (`become`) but rather by traditional assignment statements. To maintain actor isolation, the mutable state of each active object is only accessible and mutable

by the object's own thread of control. This means that state updates are also isolated and because messages are processed entirely sequentially the Isolated Turn Principle also holds for active objects.

```
[object Cell
  (state [contents := nil])
  (script
    (=> [:put newContent]
      contents := newContent)
    (=> [:get] @ From
      From <= contents))]

Cell <= [:get]
```

Listing 3. An active object in ABCL/1.

While this paper considers ABCL/1 to be the first active objects programming language, the term *active object* was only coined much later. Independently from Yonezawa's work, in the early 2000s a number of other actor programming languages within the family of active object languages have been proposed [6], namely ASP [11], Rebeca [29], ABS [20] and Encore [7]. The distinguishing feature of these programming languages is that they implement a double-layered programming model where actors are composed out of *active* and *passive* objects and all passive objects are owned by exactly one actor. Each actor has a single root object called the *active* object. Every other object that is encapsulated by that actor is called a *passive* object. Different actor do not share memory, the active objects' whole object graph is deep-copied into the actor. When actors send a message to another actor, any passive object that is transmitted (and the transitive closure of all objects referenced by that passive object) is also copied, thus maintaining a no-shared-state policy and therefore guaranteeing the Isolated Turn Principle. Other examples of actor languages based on the Active Object Model include SALSA [37] and Orleans [26].

4.1 Operational Semantics

In this section we provide a minimal operational semantics for the Active Object Model. An executable implementation in PLT Redex of this operational semantics can be found online[3]. Because the operational semantics for the Active Object Model shares some similarities with the one for the Classic Actor Model we will only highlight (in red) the differences between both formalisms.

Figure 6 lists the semantic entities for the Active Object Model. In the Active Object Model, each **Actor**, a, is represented by an identifier, ι_a, a mailbox, μ, the expression it is currently evaluating, e, a heap of passive

[3] https://gitlab.soft.vub.ac.be/jdekoste/actormodelhistorypltredex

Definition: Active Object

Every active object has a single entry point that defines a fixed interface of messages that are understood. The sequential subset of actor systems that implement this model is typically imperative. Changes to the state of an actor can be done imperatively and isolation is guaranteed by sending composite values (passive objects) between active objects by copy.

$$
\begin{array}{rcl}
k \in \textbf{Configuration} & ::= & A \\
a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}\langle \iota_a, \mu, e, O, o \rangle \\
\mu \in \textbf{Mailbox} & ::= & \overline{m} \\
m \in \textbf{Message} & ::= & \mathcal{M}\langle msg, \overline{v} \rangle \\
o \in O \subseteq \textbf{Object} & ::= & \mathcal{O}\langle \iota_o, cls, \overline{v} \rangle \\
v \in \textbf{Value} & ::= & r \mid \texttt{null} \\
r \in \textbf{Reference} & ::= & \iota_a \mid \iota_o
\end{array}
$$

$$\iota_a \in \textbf{ActorId}, \iota_o \in \textbf{ObjectId}$$

Fig. 6. Semantic entities of the Classic Actor Model

objects, O, and an active object that represents its interface, o. Contrary to the Classic Actor Model, in this model there is no `become` primitive to change the behaviour of an actor. The Active Object Model is a double-layered actor model where the interface of an actor is defined by its active object (i. e. facade of the actor), o, and its state by a heap of passive objects, O. The interface of an active object does not change throughout the lifetime of the actor (i. e. the interface is fixed). However, because passive objects are mutable, an actor can change its own state and can therefore change how it reacts to certain messages depending on that state. Any **Object**, o, in this model also has an identifier, ι_o, that uniquely identifies that object. Because objects are now also first-class entities, we extend **References** to also include object identifiers, ι_o.

4.1.1 Syntax.

Figure 7 lists the syntax, evaluation contexts and initial configuration for Active Object programs.

Syntax. A **Program**, p, is a set of class definitions, C, of which one class is the *Main* class that has to implement the *run* method. The definition of **Classes** and **Methods** are identical to the ones for the Classic Actor Model. The expressions of the Active Object Model have been extended with new syntax to create and modify passive objects and the `become` primitive has been removed. The `new` primitive creates a new instance of a class with classname, cls, and initialises its field with the values for expressions, $\overline{e}$. Referencing and modifying a field in an object is done through the $e.x$ and $e.x = e$ syntax respectively. The dot notation is also used for invoking a method on an object, $e.msg(\overline{e})$.

Evaluation Contexts. Evaluation contexts for the `new` syntax have been added to ensure the newly added compound expressions are also reduced from left to right.

Initial Configuration. The initial configuration is changed such that the single *Main* actor is now initialised with an empty heap of passive objects and a single active object with a unique identifier, ι_o.

The reduction rules are split into two layers. The *actor-local reduction rules* ($\longrightarrow_a$) in Figure 8 define the rules that reduce an actor's expression an that do not involve any additional actors (i. e. the expression can be reduced in isolation). The *actor-global reduction rules* ($\longrightarrow_k$) in Figure 9 define rules that involve multiple actors.

Actor-local reductions. Actors operate by perpetually taking the first message from their mailbox and processing it. If the message matches the actor's interface, the associated expression is evaluated (reduced) to a value. When the expression is fully reduced, the next message is processed. Contrary to the Classic Actor

Syntax

$$
\begin{array}{rcl}
p \in \textbf{Program} & ::= & C \\
C \subseteq \textbf{Class} & ::= & \texttt{class}\ cls\ \{\overline{x_f}; M\} \\
M \subseteq \textbf{Method} & ::= & msg \rightarrow \lambda\overline{x_m}.e \\
e \in \textbf{Expression} & ::= & \texttt{self} \mid x \mid \texttt{null} \mid e\ ;\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{spawn}(cls, \overline{e}) \mid \texttt{send}(e, msg, \overline{e}) \mid \\
 & & \texttt{new}(cls, \overline{e}) \mid e.x \mid e.x = e \mid e.msg(\overline{e})
\end{array}
$$

$x \in \textbf{VarName}, cls \in \textbf{ClassName}, msg \in \textbf{MessageName}$

Runtime Syntax

$$e ::= \ldots \mid r$$

Evaluation Contexts

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \square \mid \texttt{let}\ x\ =\ \mathcal{E}\ \texttt{in}\ e \mid \texttt{spawn}(cls, \overline{v}, \mathcal{E}, \overline{e}) \mid \texttt{send}(\mathcal{E}, msg, \overline{e}) \mid \texttt{send}(v, msg, \overline{v}, \mathcal{E}, \overline{e}) \mid \\
 & & \texttt{new}(cls, \overline{v}, \mathcal{E}, \overline{e}) \mid \mathcal{E}.x \mid \mathcal{E}.x = e \mid v.x = \mathcal{E} \mid \mathcal{E}.msg(\overline{e}) \mid v.msg(\overline{v}, \mathcal{E}, \overline{e})
\end{array}
$$

Syntactic Sugar

$$e\ ;\ e' \quad \stackrel{\text{def}}{=} \quad \text{let}\ x = e\ \text{in}\ e' \qquad x \notin \text{FV}(e')$$

Initial Configuration

$$K_{init} = \{\mathcal{A}\langle \iota_a, \varnothing, \texttt{send}(\iota_a, run, \varnothing), \varnothing, \mathcal{O}\langle \iota_o, Main, \varnothing\rangle\rangle\}$$

Fig. 7. Syntax of the Classic Actor Model

Model, messages are always processed in FIFO order. If the first message in the mailbox of an actor does not match the interface of the active object, and if no actor-local reduction rule is applicable to further reduce an expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its current expression is fully reduced to a value and and its mailbox is empty. A value cannot be further reduced and the actor sits idle until it receives a new message.

We now explain the actor-local reduction rules ($\longrightarrow_a$) found in Figure 8:

- LET: Reducing a "let"-expression simply substitutes the value of x for v in e.
- NEW: Newly created passive objects are always owned by the actor that creates them, i. e. the newly created object is added to the heap of the actor for which the `new` expression was reduced. A new object is created with a fresh identifier, ι_o, its classname, cls, and, the initial values for its field, $\overline{v}$. That new object is added to the heap of the actor and the expression reduces to a reference to that object.
- INVOKE: Actors can only synchronously invoke methods on their own local passive objects, i. e. the object has to be part of the heap of that actor. The class of the object and the corresponding method are looked up in C and M respectively. Invoking a method reduces to the body expression of the corresponding method. In this expression, the method parameters, $\overline{x_m}$ are substituted with their respective message arguments, $\overline{v}$. The field variables, $\overline{x_f}$ are substituted with the respective field values, $\overline{v_f}$ of the passive object, and the pseudovariable `self` is substituted with the reference to the actor, ι_a.
- FIELD-ACCESS: For accessing a field of a passive object that object has to be part of the actor's heap. The auxiliary function `lookup` is used to retrieve the value of the corresponding field and the expression reduces to that value.

$$\frac{}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{let}\ x = v\ \texttt{in}\ e], O, o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[[v/x]e], O, o\rangle}\ (\textsc{let})$$

$$\frac{\iota_o \text{ fresh} \qquad o' = \mathcal{O}\langle \iota_o, cls, \overline{v}\rangle}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{new}(cls, \overline{v})], O, o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_o], O \cup o', o\rangle}\ (\textsc{new})$$

$$\frac{\mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \in O \qquad \texttt{class}\ cls\ \{\overline{x_f}; M\} \in C \qquad msg \to \lambda \overline{x_m}.e \in M}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_o.msg(\overline{v})], O, o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[[\iota_a/\texttt{self}][\overline{v_f}/\overline{x_f}][\overline{v}/\overline{x_m}]e], O, o\rangle}\ (\textsc{invoke})$$

$$\frac{o' = \mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \qquad o' \in O \qquad v = \text{lookup}(o', x_f)}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_o.x_f], O, o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[v], O, o\rangle}\ (\textsc{field-access})$$

$$\frac{o' = \mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \qquad o'' = \text{update-field}(o', x_f, v)}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_o.x_f = v], O \cup o', o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[v], O \cup o'', o\rangle}\ (\textsc{field-update})$$

$$\frac{}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{send}(\iota_a, msg, \overline{v})], O, o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu \cdot \mathcal{M}\langle msg, \overline{v}\rangle, \mathcal{E}[\texttt{null}], O, o\rangle}\ (\textsc{self-send})$$

$$\frac{o = \mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \qquad \texttt{class}\ cls\ \{\overline{x_f}; M\} \in C \qquad msg \to \lambda \overline{x_m}.e \in M}{\mathcal{A}\langle \iota_a, \mathcal{M}\langle msg, \overline{v}\rangle \cdot \mu, v, O, o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, [\iota_a/\texttt{self}][\overline{v_f}/\overline{x_f}][\overline{v}/\overline{x_m}]e, O, o\rangle}\ (\textsc{receive})$$

Fig. 8. Local Reduction Rules for the Active Object Model

- FIELD-UPDATE: For updating a field we first extract the corresponding passive object from the heap using $O \cup o'$ and then use the auxiliary function `update-field` to change the value of the corresponding field in the object. The modified object is added again to the heap of the actor using $O \cup o''$ and the expression reduces to the value that was assigned to the field, v.
- SELF-SEND: An asynchronous message sent to the same actor simply appends a new message to the end of that actor's own mailbox. The message send itself immediately reduces to `null`.
- RECEIVE: For Active Object actors, a new message can only be processed only if the current expression of the actor cannot be reduced any further (the expression is a value v). Unlike the Classic Actor Model, messages are always processed from left to right because the interface of an active object is fixed. The first message is extracted from the mailbox of the actor using $\mathcal{M}\langle msg, \overline{v}\rangle \cdot \mu$. The corresponding method is looked up in the interface of the active object, o. If there is no match, the evaluation is *stuck* and this signifies an error in the program. Finally, the current expression of the actor is replaced with the body expression of the corresponding method.

Actor-global reductions. We now explain the actor-global reduction rules ($\longrightarrow_k$) found in Figure 9:

- SPAWN: Reducing a spawn expression adds a new actor to the set of actors of the configuration. The newly created actor is initialised with an empty mailbox, `null` as its expression (i. e. the actor will be in an idle state), an empty heap of passive objects, and a newly created object as its active object. The spawn expression itself reduces to a reference to that newly created actor $\iota_{a'}$.

$$\text{(SPAWN)}\quad \frac{\iota_{a'}, \iota_o \text{ fresh} \qquad o' = \mathcal{O}\langle \iota_o, cls, \overline{v}\rangle}{\begin{array}{c} A \uplus \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathsf{spawn}(cls, \overline{v})], O, o\rangle\} \\ \longrightarrow_k A \cup \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_{a'}], O, o\rangle\} \cup \{\mathcal{A}\langle \iota_{a'}, \varnothing, \mathtt{null}, \varnothing, o'\rangle\} \end{array}}$$

$$\text{(SEND)}\quad \frac{\overline{v'}, O' = \mathrm{pass}(\overline{v}, O)}{\begin{array}{c} A \uplus \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathsf{send}(\iota_{a'}, msg, \overline{v})], O, o\rangle\} \uplus \{\mathcal{A}\langle \iota_{a'}, \mu', e', O'', o'\rangle\} \\ \longrightarrow_k A \cup \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathtt{null}], O, o\rangle\} \cup \{\mathcal{A}\langle \iota_{a'}, \mu' \cdot \mathcal{M}\langle msg, \overline{v'}\rangle, e', O'' \cup O', o'\rangle\} \end{array}}$$

$$\text{(CONGRUENCE)}\quad \frac{a \longrightarrow_a a'}{A \uplus \{a\} \longrightarrow_k A \cup \{a'\}}$$

Fig. 9. Global Reduction Rules for the Active Object Model

- SEND: An asynchronous message sent to a different actor appends that message to the front of the mailbox of the recipient actor (the actor with identifier $\iota_{a'}$). To preserve actor isolation, active objects adopt a strict no-shared-state policy. As such, any passive object that is referenced by an argument argument of a message must be deep copied. The auxiliary function `pass` is used to create a set of passive objects, O', that contains copies for all the objects that are in the transitive closure of the objects referenced by the message arguments, $\overline{v}$. It also returns a list, $\overline{v'}$, with all the fresh references to the copied objects in $\overline{v}$. The set of copied objects is added to the heap of the recipient actor using $O'' \cup O'$. The copied objects are added to the receiving actor before the message is processed. Because all references in $\overline{v'}$ are fresh and references cannot be forged, these objects will only be accessible once the message is taken from the actor's mailbox and processed. The send expression itself reduces to `null`.
- CONGRUENCE: this rule connects the actor-local reduction rules to the global configuration reduction rules.

Auxiliary functions. The auxiliary functions `lookup` and `update-field` simply lookup a field or modify a field of a given passive object respectively.

The auxiliary function `reach` constructs a set of objects that is the transitive closure of all objects referenced by $\overline{v}$. Matching objects are removed from the original heap one by one using $O \uplus o$. Every time this is done, all references to passive objects by the fields of removed object, $\overline{v_f}$, are added to the list of *reachable* object references. If a reference is no longer part of the set of passive objects because it was previously removed (e. g. because of a circular reference), the reference is simply ignored and removed from the list.

The auxiliary function `pass` constructs a new set of copied objects that is the transitive closure of all objects referenced by $\overline{v}$. It uses the auxiliary function `reach` to first construct the set of all reachable objects, O', and then constructs a new set, O'', by replacing each object in O' with a copy for which its identifier and fields are replaced by fresh object identifiers given by σ. σ is a function that maps object identifiers, ι_o, in the original heap to fresh object identifiers, $\iota_{o'}$. All other values (i. e. actor identifier, ι_a) are left unchanged.

4.2 Proving the Isolated Turn Principle for the Active Object Model

In this section we provide a formal proof that the Isolated Turn principle holds for the Active Object Model.

Definition 4.1 (Turn). Let a turn, $\xrightarrow{\iota_a}{}^*_k$, be any sequence of reduction rules such that $A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, O, o\rangle\} \xrightarrow{\iota_a}{}^*_k A \uplus \{\mathcal{A}\langle \iota_a, \mu', v, O', o'\rangle\}$ and the (RECEIVE) reduction rule for the actor with id, ι_a, $\notin \xrightarrow{\iota_a}{}^*_k$.

$$
\begin{array}{rcll}
\text{lookup}(\mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle, x_f) & \stackrel{def}{=} & \text{lookup}(\overline{x_f}, \overline{v_f}, x_f) & \text{if } \texttt{class } cls\ \{\overline{x_f}; M\} \in C \\
\text{lookup}(x_f \cdot \overline{x_f}, v_f \cdot \overline{v_f}, x_f) & \stackrel{def}{=} & v & \\
\text{lookup}(x'_f \cdot \overline{x_f}, v_f \cdot \overline{v_f}, x_f) & \stackrel{def}{=} & \text{lookup}(\overline{x_f}, \overline{v_f}, x_f) & \\
 & & & \\
\text{update-field}(\mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle, x_f, v) & \stackrel{def}{=} & \mathcal{O}\langle \iota_o, cls, \text{update-field}(\overline{x_f}, \overline{v_f}, x_f, v)\rangle & \text{if } \texttt{class } cls\ \{\overline{x_f}; M\} \in C \\
\text{update-field}(x_f \cdot \overline{x_f}, v_f \cdot \overline{v_f}, x_f, v) & \stackrel{def}{=} & v \cdot \overline{v_f} & \\
\text{update-field}(x'_f \cdot \overline{x_f}, v_f \cdot \overline{v_f}, x_f, v) & \stackrel{def}{=} & v_f \cdot \text{update-field}(\overline{x_f}, \overline{v_f}, x_f, v) & \\
 & & & \\
\text{reach}(\varnothing, \overline{v}) & \stackrel{def}{=} & \varnothing & \\
\text{reach}(O, \varnothing) & \stackrel{def}{=} & \varnothing & \\
\text{reach}(O \uplus o, \overline{v} \cdot \iota_o) & \stackrel{def}{=} & o \cup \text{reach}(O, \overline{v} \cdot \overline{v_f}) & \text{if } o = \mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \\
\text{reach}(O, \overline{v} \cdot v) & \stackrel{def}{=} & \text{reach}(O, \overline{v}) & \text{otherwise} \\
 & & & \\
\text{pass}(\overline{v}, O) & \stackrel{def}{=} & \overline{\sigma(v)}, O'' & \\
 & & \text{where } O' = reach(O, \overline{v}) & \sigma(v) \begin{cases} \sigma'(v) & \text{if } v = \iota_o \\ v & \text{otherwise} \end{cases} \\
 & & O'' = \{\mathcal{O}\langle \sigma(\iota_o), cls, \overline{\sigma(v_f)}\rangle \mid \mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \in O'\} & \sigma' = \{\iota_o \mapsto \iota'_o \mid \mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \in O', \iota'_o \text{ fresh}\}
\end{array}
$$

Fig. 10. Auxiliary Functions

Theorem 4.2 (Theorem of Safety). *If* $A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, O, o\rangle\} \xrightarrow{\iota_a}{}^{*}_{k} A' \uplus \{\mathcal{A}\langle \iota_a, \mu', v, O', o'\rangle\}$ *then all possible values for* v *are* α*-equivalent.*

Proof of Safety. The evaluation contexts defined in Figure 7 determine a strict total order in which compound expressions can be reduced. Therefore, This proof follows by induction over the the reduction of expressions. Cases (SELF-SEND) and (SEND) are trivial as they both reduce to `null`. Case (NEW) reduces to a α-equivalent object reference. An α-equivalent object o' is also added to the heap of the actor. Cases (INVOKE), (FIELD-ACCESS), and (FIELD-UPDATE) follow by induction. If ι_o is α-equivalent and the referenced object in the heap is also α-equivalent, then resulting expressions will also be α-equivalent. Case (LET) follows by induction, if all possible v are α-equivalent then the substitution $[v/x]e$ will also be α-equivalent. Case (SPAWN) reduces to a fresh α-equivalent identifier $\iota_{a'}$. Case (CONGRUENCE) follows by induction over the actor-local reduction rules. Case (RECEIVE) does not apply as (RECEIVE) $\notin \xrightarrow{\iota_a}{}^{*}_{k}$ using definition 4.1. □

Definition 4.3 (Error). If no reduction rule is applicable to further reduce a reducible expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its current expression is fully reduced to a value and its mailbox is empty.

Theorem 4.4 (Theorem of Liveness). $\forall K = A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, O, o\rangle\}, e \notin \textbf{Value}, \exists K' : K \xrightarrow{\iota_a}_{k} K'$

Proof of Liveness. This proof follows by induction over the reduction of expressions. Cases (LET), (NEW), (INVOKE), (FIELD-ACCESS), (FIELD-UPDATE), (SELF-SEND), (SEND) and (SPAWN) all operate on reducible expressions and can be applied in any error-free program following definition 4.3. Case (CONGRUENCE) follows

by induction over the actor-local reduction rules. Case (RECEIVE) does not apply as (RECEIVE) $\notin \xrightarrow{\iota a}_k$ using definition 4.1. □

In theorem 4.2 and theorem 4.4 we have shown that active objects satisfy both safety and liveness for the execution of a single turn and therefore the Isolated Turn Principle holds.

5 PROCESSES

Independently from Gul Agha and Akinori Yonezawa, T. Hoare, was also inspired by Carl Hewitt's Actor Model of computation. This inspiration lead him, in 1978, to introduce the formal language communicating sequential processes [19] to study the interactions between concurrent processes. Almost a decade later, in 1986, Joe Armstrong developed the first version of the Erlang programming language [5] while working for Ericsson and Ellemtel Computer Science Laboratories. The sequential parts of Erlang are heavily inspired by Prolog. Erlang draws a lot of inspiration from CSP for its concurrency model. Although none of the early papers about Erlang directly reference the Actor Model, its influence on the concurrency model of Erlang is undeniable. We therefore consider Erlang to be the first industry-strength language to adopt the actor model as its model of concurrency. It was developed as a declarative language for programming large industrial telecommunications switching systems.

While the communication mechanism of Erlang's processes is very close to that of the Classic Actor Model, different mechanics are used to achieve similar effects. Most notably, an actor is not modelled as a named behaviour. Rather actors are modelled as processes that run from start till completion. Erlang actors can use the primitive `receive` to specify what messages the executing actor can receive when the execution of a process reaches that expression. When evaluating a `receive` expression the actor pauses until a message is received. If a message is received, the matching code is evaluated and execution continues until a new `receive` block is evaluated. One can use recursion to ensure that an actor continuously processes incoming messages. What types of messages an actor understands throughout its lifetime is determined by the dynamic extent of the expression it is reducing.

```
loop(Contents) ->
  receive
    {put, NewContent} ->
      loop(NewContent);
    {get, From} ->
      From ! Contents,
      loop(Contents)
  end.

MyCell = spawn(loop, [nil]).
MyCell ! {get, self()}.
```

Listing 4. An Erlang process.

This is illustrated by Listing 4. The `spawn` primitive creates a new Erlang process. This will call the provided function, `loop`, in a new process and returns that process' id. The cell uses the primitive `receive` to match incoming get- and put-messages. Once the message body is processed the loop function calls itself recursively to process the next message, passing along the updated state.

Definition: Processes

Every process runs from start till completion. The sequential subset of actor systems that implement this model is typically functional. Changes to the state of an actor are aggregated in a single receive statement. The scope of this receive statement then defines the current state of that actor. Processes have a single entry point that defines a flexible interface that can change by evaluating different receive expressions over time.

The Scala Actor Library [16] is another well-known implementation of the Process model. Other examples include Kilim [30] and SALSA [36].

5.1 Operational Semantics

In this section we provide a minimal operational semantics for the Process Model. An executable implementation in PLT Redex of this operational semantics can be found online[4]. In red, we highlight the differences between this formal specification and the one for the Classic Actor Model.

Figure 11 lists the semantic entities for the Process Model. In the Process Model, each **Actor**, a, is represented by an identifier, ι_a, a mailbox, μ, and the expression it is currently evaluating, e. Contrary to the Classic Actor Model, the behaviour (i. e. interface and state) is defined by the expression the actor is currently reducing. An actor can *change* its interface by using the primitive `receive` to specify what types of messages it understands. The state of an actor is modeled by the expression itself. the Process Model is not a double-layered actor model, so we do not model objects in this formalism.

$$
\begin{array}{rcl}
k \in \textbf{Configuration} & ::= & A \\
a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}\langle \iota_a, \mu, e \rangle \\
\mu \in \textbf{Mailbox} & ::= & \overline{m} \\
m \in \textbf{Message} & ::= & \mathcal{M}\langle msg, \overline{v} \rangle \\
v \in \textbf{Value} & ::= & r \mid \texttt{null} \\
r \in \textbf{Reference} & ::= & \iota_a
\end{array}
$$

$$\iota_a \in \textbf{ActorId}$$

Fig. 11. Semantic entities of the Process Model

5.1.1 *Syntax.*

Figure 12 lists the syntax, evaluation contexts and initial configuration for the Process Model.

Syntax. Contrary to the Classic Actor Model and the Active Object Model, a Process **Program**, p, is a set of functions, M (named **Methods** here, for consistency with the other formalisms). This set remains constant throughout the lifetime of the program execution and must contain a single method named *run*. The syntax for expressions is extended with function application, $msg(\overline{e})$ and a new `receive` primitive. A new actor is no longer spawned (i. e. initialised) from a class name, but rather from a method name. Spawning a new actor simply executes the corresponding method using the provided arguments in a new process.

[4] https://gitlab.soft.vub.ac.be/jdekoste/actormodelhistorypltredex

Syntax

$p \in$ **Program** $::=$ M

$M \subseteq$ **Method** $::=$ $msg \rightarrow \lambda\overline{x_m}.e$

$e \in$ **Expression** $::=$ $\texttt{self} \mid x \mid \texttt{null} \mid e\,;\,e \mid msg(\overline{e}) \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid$
$\texttt{spawn}(msg, \overline{e}) \mid \texttt{send}(e, msg, \overline{e}) \mid \texttt{receive}(\overline{msg \rightarrow \lambda\overline{x_m}.e})$

$x \in$ **VarName**, $msg \in$ **MessageName**

Runtime Syntax

$e ::= \ldots \mid r$

Evaluation Contexts

$\mathcal{E} ::= \square \mid msg(\overline{v}, \mathcal{E}, \overline{e}) \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ e \mid \texttt{spawn}(msg, \overline{v}, \mathcal{E}, \overline{e}) \mid \texttt{send}(\mathcal{E}, msg, \overline{e}) \mid \texttt{send}(v, msg, \overline{v}, \mathcal{E}, \overline{e})$

Syntactic Sugar

$e\,;\,e' \stackrel{\text{def}}{=} \text{let}\ x = e\ \text{in}\ e' \quad x \notin \text{FV}(e')$

Initial Configuration

$K_{init} = \{\mathcal{A}\langle \iota_a, \varnothing, run()\rangle\}$

Fig. 12. Syntax of the Process Model

Evaluation Contexts. Evaluation contexts for function application have been added to ensure arguments are also deterministically reduced from left to right.

Initial Configuration. The initial configuration is changed such that the single main actor is now initialised with a single expression that simply calls the main *run* method.

The reduction rules are again split into two layers (i. e. *actor-local reduction rules*, $\longrightarrow_a$, and *actor-global reduction rules*, $\longrightarrow_k$.

Actor-local reductions. Actors in this model are processes that run from start till completion. With respect to the operational semantics, this means that an actor will keep reducing its expression until it is fully reduced to a value. For simplicity, we do not model the removal of such actors from the configuration. Rather, when that happens, the actor sits idle and will no longer be able to process any incoming messages. If no reduction rule is applicable to further reduce a reducible expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its current expression is fully reduced to a value or when the current evaluation context contains a `receive` expression and none of the messages in its mailbox match with the interface of that expression. When that happens, the actor sits idle until it receives a new message that does match its interface.

We now explain the actor-local reduction rules ($\longrightarrow_a$) found in Figure 13:

- LET: Reducing a "let"-expression simply substitutes the value of x for v in e.
- RECEIVE: This rule describes the processing of messages in the mailbox of an actor. A new message can be processed only if two conditions are satisfied: the current evaluation context contains a `receive` expression and one of the messages in the mailbox of the actor matches that `receive` expression's interface. To match a message from the actor's mailbox the auxiliary function `match` is used. This function

$$\text{(LET)}\quad \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{let}\ x = v\ \texttt{in}\ e]\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[[v/x]e]\rangle$$

$$\text{(RECEIVE)}\quad \frac{e, \mu' = \text{match}(\overline{msg \to \lambda \overline{x_m}.e}, \mu)}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{receive}(\overline{msg \to \lambda \overline{x_m}.e})]\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu', \mathcal{E}[e]\rangle}$$

$$\text{(CALL)}\quad \frac{msg \to \lambda \overline{x_m}.e \in M}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[msg(\overline{v})]\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[[\iota_a/\texttt{self}][\overline{v}/\overline{x_m}]e]\rangle}$$

$$\text{(SELF-SEND)}\quad \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{send}(\iota_a, msg, \overline{v})]\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu \cdot \mathcal{M}\langle msg, \overline{v}\rangle, \mathcal{E}[\texttt{null}]\rangle$$

Fig. 13. Local Reduction Rules for Process Model

returns the body expression. e, of the matched method and an updated mailbox, μ where the matched message was removed. The `receive` expression reduces to the body expression of the associated method and the actor's mailbox is updated.

- CALL: When an actor calls a function, that function is looked up in the constant set of functions, M. The call expression reduces to the body expression of the corresponding method where the parameters, $\overline{x_m}$ are substituted with the arguments to the call, $\overline{v}$ and the pseudovariable, `self` is substituted with a reference to the current actor, ι_a.
- SELF-SEND: An asynchronous message sent to the same actor simply appends a new message to the end of that actor's own mailbox. The message send itself immediately reduces to `null`.

$$\text{(SPAWN)}\quad \frac{\iota_{a'}\ \text{fresh} \qquad msg \to \lambda \overline{x_m}.e \in M}{A \uplus \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{spawn}(msg, \overline{v})]\rangle\} \longrightarrow_k A \cup \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_{a'}]\rangle\} \cup \{\mathcal{A}\langle \iota_{a'}, \varnothing, [\iota_{a'}/\texttt{self}][\overline{v}/\overline{x_m}]e\rangle\}}$$

$$\text{(SEND)}\quad A \uplus \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{send}(\iota_{a'}, msg, \overline{v})]\rangle\} \uplus \{\mathcal{A}\langle \iota_{a'}, \mu', e'\rangle\} \longrightarrow_k A \cup \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{null}]\rangle\} \cup \{\mathcal{A}\langle \iota_{a'}, \mu' \cdot \mathcal{M}\langle msg, \overline{v}\rangle, e'\rangle\}$$

$$\text{(CONGRUENCE)}\quad \frac{a \longrightarrow_a a'}{A \uplus \{a\} \longrightarrow_k A \cup \{a'\}}$$

Fig. 14. Global Reduction Rules for Process Model

Actor-global reductions. We now explain the actor-global reduction rules ($\longrightarrow_k$) found in Figure 14:

- SPAWN: Reducing a spawn expression adds a new actor to the set of actors of the configuration. The newly created actor is initialised with a fresh identifier, $\iota_{a'}$, and an empty mailbox. The method named, msg is looked up in the constant set of methods, M. The `spawn` expression reduces to the body expression of the corresponding method where the parameters, $\overline{x_m}$ are substituted with the arguments to the `spawn` expression, $\overline{v}$ and the pseudovariable, `self` is substituted with a reference to the newly created actor, $\iota_{a'}$. The spawn expression itself reduces to the same reference to the newly created actor, $\iota_{a'}$.
- SEND: An asynchronous message sent to a different actor appends that message to the front of the mailbox of the recipient actor (the actor with identifier ι_a). The send expression itself reduces to `null`.

- CONGRUENCE: this rule connects the actor-local reduction rules to the global configuration reduction rules.

$$\begin{array}{rcll} \text{match}(M, \mathcal{M}\langle msg, \overline{v}\rangle \cdot \mu) & \stackrel{def}{=} & [\overline{v}/\overline{x_m}]e, \mu & \text{if } msg \rightarrow \lambda\overline{x_m}.e \in M \\ \text{match}(M, m \cdot \mu) & \stackrel{def}{=} & e, m \cdot \mu' & \text{if } e, \mu' = \text{match}(M, \mu) \end{array}$$

Fig. 15. Auxiliary Functions

Auxiliary functions. The auxiliary function `match` is identical to the one for the Classic Actor Model. It finds the first message (going from left to right) in the mailbox, μ, that matches the selector msg. It has two return values: the body expression of the associated lambda where all parameters, $\overline{x_m}$ have been substituted by the arguments of the message, $\overline{v}$, and and updated mailbox where the selected message has been removed.

5.2 Proving the Isolated Turn Principle for the Process Model

In this section we provide a formal proof that the Isolated Turn principle holds for the Process Model.

Definition 5.1 (Turn). Let a turn, $\xrightarrow{\iota_a}{}^*_k$, be any sequence of reduction rules such that $A \uplus \{\mathcal{A}\langle \iota_a, \mu, e\rangle\} \xrightarrow{\iota_a}{}^*_k A \uplus \{\mathcal{A}\langle \iota_a, \mu', v\rangle\}$ and the (RECEIVE) reduction rule for the actor with id, ι_a, $\notin \xrightarrow{\iota_a}{}^*_k$.

THEOREM 5.2 (THEOREM OF SAFETY). *If $A \uplus \{\mathcal{A}\langle \iota_a, \mu, e\rangle\} \xrightarrow{\iota_a}{}^*_k A' \uplus \{\mathcal{A}\langle \iota_a, \mu', v\rangle\}$ then all possible values for v are α-equivalent.*

PROOF OF SAFETY. The evaluation contexts defined in Figure 12 determine a strict total order in which compound expressions can be reduced. Therefore, This proof follows by induction over the the reduction of expressions. Cases (SELF-SEND) and (SEND) are trivial as they both reduce to `null`. Case (CALL) follows by induction. If $msg(\overline{v})$ is an α-equivalent expression, then selected method will be identical because M is constant and therefore the substitution $[\iota_a/\texttt{self}][\overline{v}/\overline{x_m}]e$ is also α-equivalent. Similarly, case (SPAWN) follows by induction. If $\texttt{spawn}(msg, \overline{v})$ is an α-equivalent expression, then the substitution $[\iota_{a'}/\texttt{self}][\overline{v}/\overline{x_m}]e$ is also α-equivalent because $\iota_{a'}$ is fresh and e is selected from a constant set of methods, M. Case (LET) follows by induction, if all possible v are α-equivalent then the substitution $[v/x]e$ will also be α-equivalent. Case (SPAWN) reduces to a fresh α-equivalent identifier $\iota_{a'}$. Case (CONGRUENCE) follows by induction over the actor-local reduction rules. Case (RECEIVE) does not apply as (RECEIVE) $\notin \xrightarrow{\iota_a}{}^*_k$ using definition 5.1. □

Definition 5.3 (Error). If no reduction rule is applicable to further reduce a reducible expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its current expression is fully reduced to a value or when the current evaluation context contains a `receive` expression and no messages in its mailbox matches with that receive's interface.

THEOREM 5.4 (THEOREM OF LIVENESS). $\forall K = A \uplus \{\mathcal{A}\langle \iota_a, \mu, e\rangle\}, e \notin \textbf{Value}, \exists K' : K \xrightarrow{\iota_a}_k K'$

PROOF OF LIVENESS. This proof follows by induction over the reduction of expressions. Cases (LET), (CALL), (SELF-SEND), (SEND) and (SPAWN) all operate on reducible expressions and can be applied in any error-free program following definition 5.3. Case (CONGRUENCE) follows by induction over the actor-local reduction rules. Case (RECEIVE) does not apply as (RECEIVE) $\notin \xrightarrow{\iota_a}_k$ using definition 5.1. □

In theorem 5.2 and theorem 5.4 we have shown that the Process Model satisfies both safety and liveness for the execution of a single turn and therefore the Isolated Turn Principle holds.

6 COMMUNICATING EVENT LOOPS

Vulcan [22] is a concurrent object-oriented logic programming language that was also inspired by Hewitt's actor model of computation. It was developed in 1986 as a preprocessor for Concurrent Prolog by Kahn et al. at Xerox PARC. Dean Tribble, an intern at Xerox PARC during the eighties, developed the distributed programming language Joule in 1994. Joule can be seen as the most direct and important ancestor of the E programming language [25] developed by Mark Miller. E was the first language to introduce *the Communicating Event-loop Actor Model*. Similar to the Active Object Model, the Communicating Event-loop Actor Model is a double-layered model where actors (called *vats* in E) and objects co-exist. However, it has an important difference when compared to the Active Object Model in that it does not make a distinction between passive and active objects. In this model, actors do not have a single active object that serves as the entry point to that actor. Actors are not even first-class entities in this model. Rather, actors can obtain direct references to objects in the heap of another actor. Within an actor, references to objects owned by that same actor are called *near references*. References to objects owned by other actors are called *far references*. The type of reference determines the access capabilities of that actor's thread of execution on the referenced object. While actors can obtain direct references to objects owned by a different actor (far reference), they are not allowed to make immediate calls on those references. Generally, actors are introduced to one another by exchanging addresses. In the Communicating Event Loop model such an address is always in the form of an far reference to a specific object. The referenced object then defines how another actor can interface with that actor. The main difference between Communicating Event Loops (CEL) and other actor models seen so far was that other actor models usually only provide a single entry point or address to an actor (in other words, at any point in time, an actor can have only 1 interface). A CEL actor can define multiple objects that all share the same mailbox and thread of control and hand out different references to those objects, thus essentially allowing one to model an actor that has multiple interfaces at the same time. This helps support a PoLP (principle of least privilige) style of programming [27], by facilitating the creation of many small, object-level interfaces, rather than a single large actor-level interface. The example in Listing 5 illustrates how to create an object in E and send it an eventual message `get`.

When an object in one actor sends an eventual message to an object in another actor the message is enqueued in the mailbox of the owner of the receiver object and immediately returns a *promise*. That promise will be resolved with the return value of the message once that message is processed. It is not allowed for an actor to use a promise as a near reference. If an actor wants to make an immediate call on the value represented by a promise, like printing it on the screen, that actor must set up an action to occur when the promise resolves. This is done by using the `when` primitive. Promises in E are based on Argus's promises [24]. With the main difference being that accessing a promise in Argus is a blocking operation while E adopts a purely asynchronous model (i. e. executing the `when` primitive is also an asynchronous operation). When the promise for the value of the `get` message becomes resolved, the body of the `when` primitive is executed. During that execution the promise is resolved and can be used as a local object.

The Communicating Event Loop model was later adopted by AmbientTalk [35], a distributed object-oriented programming language that was designed for developing applications on mobile ad hoc networks. AmbientTalk was designed as an *ambient-oriented programming* (AmOP) language [13]. It adds to the Actor Model a number

```
def cell {
  var contents := null
  to put(newContents) {
    contents := newContents
  }
  to get() {
    return contents
  }
}

var promise := cell<-get()
when (promise) -> {
  println(promise)
}
```

Listing 5. An actor in E.

Definition: Communicating Event Loops

An actor is a combination of an object heap, a mailbox and a thread of control. Every reference that is passed between different actors is exported as a far reference with a fixed interface and can serve as an entry point for that actor. The sequential subset of actor systems that implement this model is typically imperative. Changes to the state of an actor can be done imperatively and isolation is guaranteed by sending composite values between actors by far reference.

of new primitives to handle disconnecting and reconnecting nodes in a network where connections are volatile. The core concurrency model however remains faithful to the original Communicating Event Loops of E.

6.1 Operational Semantics

In this section we provide a minimal operational semantics for Communicating Event Loops. An executable implementation in PLT Redex of this operational semantics can be found online[5]. Because the operational semantics for Communicating Event Loops shares a lot of similarities with the one for the Active Object Model we will only highlight (in red) the differences between both formalisms.

$$
\begin{array}{rcl}
k \in \textbf{Configuration} & ::= & A \\
a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}\langle \iota_a, \mu, e, O \rangle \\
\mu \in \textbf{Mailbox} & ::= & \overline{m} \\
m \in \textbf{Message} & ::= & \mathcal{M}\langle \iota_o, msg, \overline{v} \rangle \\
o \in O \subseteq \textbf{Object} & ::= & \mathcal{O}\langle \iota_o, cls, \overline{v} \rangle \\
v \in \textbf{Value} & ::= & r \mid \texttt{null} \\
r \in \textbf{Reference} & ::= & \iota_a.\iota_o
\end{array}
$$

$$\iota_a \in \textbf{ActorId}, \iota_o \in \textbf{ObjectId}$$

Fig. 16. Semantic entities of the Communicating Event Loops Model

[5] https://gitlab.soft.vub.ac.be/jdekoste/actormodelhistorypltredex

Figure 16 lists the semantic entities for Communicating Event Loops. In Communicating Event Loops, each **Actor**, a, is represented by an identifier, ι_a, a mailbox, μ, the expression it is currently evaluating, e, and a heap of objects, O. Contrary to the Active Object Model, there is no single active object that serves as the entry point to an actor. Rather, actors can share far references to objects in their own heap with other actors. Each of these far references can be used by other actors to send asynchronous messages. Therefore, any such far reference can serve as an entry point to an actor. However, the interface of the referenced objects cannot change throughout the lifetime of the actor. That means, while a Communicating Event-loop actor can have many interfaces, they remain fixed. Because messages are sent to an object within the heap of an actor, a **Message** has an additional identifier, ι_o, that identifies the target object of the message that was received. Far references are modeled as a double identifier, $\iota_a.\iota_o$, that identifies the referenced object and the actor that owns that object.

6.1.1 Syntax.

Figure 17 lists the syntax, evaluation contexts and initial configuration for Communicating Event-loop Actor programs.

Syntax. Identical to the Classic Actor Model and the Active Object Model, a **Program**, p, is a set of class definitions, C, of which one class is the *Main* class that has to implement the *run* method. The definition of **Classes** and **Methods** has been left unchanged. With respect to the Active Object Model, the set of expressions that are part of the syntax also remains identical.

Syntax

$p \in$ **Program**	$::=$	C
$C \subseteq$ **Class**	$::=$	$\texttt{class}\ cls\ \{\overline{x_f}; M\}$
$M \subseteq$ **Method**	$::=$	$msg \rightarrow \lambda\overline{x_m}.e$
$e \in$ **Expression**	$::=$	$\texttt{self} \mid x \mid \texttt{null} \mid e\ ;\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{spawn}(cls, \overline{e}) \mid \texttt{send}(e, msg, \overline{e}) \mid$ $\texttt{new}(cls, \overline{e}) \mid e.x \mid e.x = e \mid e.msg(\overline{e})$

$x \in$ **VarName**, $cls \in$ **ClassName**, $msg \in$ **MessageName**

Runtime Syntax

$e ::= \ldots \mid r$

Evaluation Contexts

$\mathcal{E} ::= \square \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ e \mid \texttt{spawn}(cls, \overline{v}, \mathcal{E}, \overline{e}) \mid \texttt{send}(\mathcal{E}, msg, \overline{e}) \mid \texttt{send}(v, msg, \overline{v}, \mathcal{E}, \overline{e}) \mid$ $\texttt{new}(cls, \overline{v}, \mathcal{E}, \overline{e}) \mid \mathcal{E}.x \mid \mathcal{E}.x = e \mid v.x = \mathcal{E} \mid \mathcal{E}.msg(\overline{e}) \mid v.msg(\overline{v}, \mathcal{E}, \overline{e})$

Syntactic Sugar

$e\ ;\ e' \stackrel{\text{def}}{=} \text{let}\ x = e\ \text{in}\ e' \quad x \notin \text{FV}(e')$

Initial Configuration

$K_{init} = \{\mathcal{A}\langle \iota_a, \varnothing, \iota_a.\iota_o.run(), \{\mathcal{O}\langle \iota_o, Main, \varnothing\rangle\}\rangle\}$

Fig. 17. Syntax of the Communicating Event Loops Model

Initial Configuration. The initial configuration is changed such that the single *Main* actor is now initialised with a singleton set containing an object that is an instance of the *Main* class. The actor's expression is initialised with $\iota_a.\iota_o.run()$, which will synchronously invoke the *run* method on that object.

Actor-local reductions. Actors operate by perpetually taking the first message from their mailbox and processing it. Any received message is simply forwarded to the recipient object by synchronously invoking the corresponding method on that object. When the invocation expression is fully reduced to a value, the next message can be processed. Contrary to the Classic Actor Model and the Process Model, messages are always processed in order. If the first message in the mailbox of an actor does not match the interface of the recipient object, or if no actor-local reduction rule is applicable to further reduce a reducible expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its current expression is fully reduced to a value and and its mailbox is empty. A value cannot be further reduced and the actor sits idle until it receives a new message.

$$\textsc{(let)}\quad \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\texttt{let}\ x = v\ \texttt{in}\ e], O\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[[v/x]e], O\rangle$$

$$\textsc{(new)}\quad \frac{\iota_o \text{ fresh} \qquad o = \mathcal{O}\langle \iota_o, cls, \overline{v}\rangle}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathsf{new}(cls, \overline{v})], O\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_a.\iota_o], O \cup o\rangle}$$

$$\textsc{(invoke)}\quad \frac{\mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \in O \qquad \texttt{class}\ cls\ \{\overline{x_f}; M\} \in C \qquad msg \to \lambda\overline{x_m}.e \in M}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_a.\iota_o.msg(\overline{v})], O\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[[\iota_a.\iota_o/\texttt{self}][\overline{v_f}/\overline{x_f}][\overline{v}/\overline{x_m}]e], O\rangle}$$

$$\textsc{(field-access)}\quad \frac{o = \mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \qquad o \in O \qquad v = \text{lookup}(o, x_f)}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_a.\iota_o.x_f], O\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[v], O\rangle}$$

$$\textsc{(field-update)}\quad \frac{o = \mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle \qquad o' = \text{update-field}(o, x_f, v)}{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_a.\iota_o.x_f = v], O \cup o\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[v], O \cup o'\rangle}$$

$$\textsc{(self-send)}\quad \mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathsf{send}(\iota_a.\iota_o, msg, \overline{v})], O\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu \cdot \mathcal{M}\langle \iota_o, msg, \overline{v}\rangle, \mathcal{E}[\mathtt{null}], O\rangle$$

$$\textsc{(receive)}\quad \mathcal{A}\langle \iota_a, \mathcal{M}\langle \iota_o, msg, \overline{v}\rangle \cdot \mu, v, O\rangle \longrightarrow_a \mathcal{A}\langle \iota_a, \mu, \iota_a.\iota_o.msg(\overline{v}), O\rangle$$

Fig. 18. Local Reduction Rules for Communicating Event Loops

We now explain the actor-local reduction rules ($\longrightarrow_a$) found in Figure 18:

- LET: Reducing a "let"-expression simply substitutes the value of x for v in e.
- NEW: Newly created objects are always owned by the actor that creates them, i. e. the newly created object is added to the heap of the actor for which the new expression was reduced. The new expression reduces to a reference to the newly created object. That reference is a double identifier that references the object, ι_o, and the actor that created (or owns) the object, ι_a.
- INVOKE: Actors can only synchronously invoke methods on their own objects, i. e. the object has to be part of the heap of that actor and the actor part of the reference, ι_a, needs to be the same as the actor

identifier. In the result expression, the pseudovariable `self` is substituted with the reference to the object on which the method was invoked, $\iota_a.\iota_o$.

- FIELD-ACCESS: Field access works identical to the Active Object Model, except that field access only works on references to objects that are owned by the actor dereferencing them (i. e. the actor part of the reference, ι_a, needs to be the same as the actor identifier).
- FIELD-UPDATE: Similar to field access.
- SELF-SEND: An asynchronous message sent to the same actor simply appends a new message to the end of that actor's own mailbox. The object part of the reference, ι_o, is copied into the message to later identify the recipient object. The message send itself immediately reduces to `null`.
- RECEIVE: A message can only be processed if the current expression of the actor cannot be reduced any further (the expression is a value v). Processing a message simply forwards that message to the recipient object. This is achieved by replacing the value with a synchronous invocation expression that invokes the corresponding method on the object that was the target of the message.

$$\text{(SPAWN)}\quad \frac{\iota_{a'}, \iota_o \text{ fresh}}{\begin{array}{l} A \uplus \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathsf{spawn}(cls, \overline{v})], O\rangle\} \\ \longrightarrow_k A \cup \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\iota_{a'}.\iota_o], O\rangle\} \cup \{\mathcal{A}\langle \iota_{a'}, \varnothing, \mathtt{null}, \{\mathcal{O}\langle \iota_o, cls, \overline{v}\rangle\}\rangle\} \end{array}}$$

$$\text{(SEND)}\quad \begin{array}{l} A \uplus \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathsf{send}(\iota_{a'}.\iota_o, msg, \overline{v})], O\rangle\} \uplus \{\mathcal{A}\langle \iota_{a'}, \mu', e', O'\rangle\} \\ \longrightarrow_k A \cup \{\mathcal{A}\langle \iota_a, \mu, \mathcal{E}[\mathtt{null}], O\rangle\} \cup \{\mathcal{A}\langle \iota_{a'}, \mu' \cdot \mathcal{M}\langle \iota_o, msg, \overline{v}\rangle, e', O'\rangle\} \end{array}$$

$$\text{(CONGRUENCE)}\quad \frac{a \longrightarrow_a a'}{A \uplus \{a\} \longrightarrow_k A \cup \{a'\}}$$

Fig. 19. Global Reduction Rules for Communicating Event Loops

Actor-global reductions. We now explain the actor-global reduction rules ($\longrightarrow_k$) found in Figure 19:

- SPAWN: Reducing a spawn expression adds a new actor to the set of actors of the configuration. The newly created actor is initialised with an empty mailbox, `null` as its expression (the actor will be in an idle state), and an object heap with a single object. The spawn expression itself reduces to a reference to that object $\iota_{a'}.\iota_o$.
- SEND: An asynchronous message sent to a different actor appends that message to the front of the mailbox of the recipient actor (the actor with identifier $\iota_{a'}$). To preserve actor isolation, event-loop actors adopt a strict no-shared-state policy. Contrary to the Active Object Model, where passive objects are shared by copy, references to objects in an event-loop actor can be freely shared between actors because they are always *tagged* with a reference to the actor that owns the object. Only the actor that owns an object is allowed to synchronously reference, mutate, or invoke methods on that object.
- CONGRUENCE: this rule connects the actor-local reduction rules to the global configuration reduction rules.

Auxiliary functions. The auxiliary functions `lookup` and `update-field` are identical to the ones for the Active Object Model. They simply lookup a field or modify a field of a given passive object respectively.

$$
\begin{array}{rcll}
\text{lookup}(\mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle, x_f) & \stackrel{def}{=} & \text{lookup}(\overline{x_f}, \overline{v_f}, x_f) & \text{if } \texttt{class}\ cls\ \{\overline{x_f}; M\} \in C \\
\text{lookup}(x_f \cdot \overline{x_f}, v_f \cdot \overline{v_f}, x_f) & \stackrel{def}{=} & v & \\
\text{lookup}(x'_f \cdot \overline{x_f}, v_f \cdot \overline{v_f}, x_f) & \stackrel{def}{=} & \text{lookup}(\overline{x_f}, \overline{v_f}, x_f) & \\
 & & & \\
\text{update-field}(\mathcal{O}\langle \iota_o, cls, \overline{v_f}\rangle, x_f, v) & \stackrel{def}{=} & \mathcal{O}\langle \iota_o, cls, \text{update-field}(\overline{x_f}, \overline{v_f}, x_f, v)\rangle & \text{if } \texttt{class}\ cls\ \{\overline{x_f}; M\} \in C \\
\text{update-field}(x_f \cdot \overline{x_f}, v_f \cdot \overline{v_f}, x_f, v) & \stackrel{def}{=} & v \cdot \overline{v_f} & \\
\text{update-field}(x'_f \cdot \overline{x_f}, v_f \cdot \overline{v_f}, x_f, v) & \stackrel{def}{=} & v_f \cdot \text{update-field}(\overline{x_f}, \overline{v_f}, x_f, v) &
\end{array}
$$

Fig. 20. Auxiliary Functions

6.2 Proving the Isolated Turn Principle for the Communicating Event Loop Model

In this section we provide a formal proof that the Isolated Turn principle holds for Communicating Event Loop Actors.

Definition 6.1 (Turn). Let a turn, $\xrightarrow{\iota_a}{}^*_k$, be any sequence of reduction rules such that $A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, O\rangle\} \xrightarrow{\iota_a}{}^*_k A \uplus \{\mathcal{A}\langle \iota_a, \mu', v, O'\rangle\}$ and the (RECEIVE) reduction rule for the actor with id, ι_a, $\notin \xrightarrow{\iota_a}{}^*_k$.

THEOREM 6.2 (THEOREM OF SAFETY). *If $A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, O\rangle\} \xrightarrow{\iota_a}{}^*_k A' \uplus \{\mathcal{A}\langle \iota_a, \mu', v, O'\rangle\}$ then all possible values for v and O' are α-equivalent.*

PROOF OF SAFETY. The evaluation contexts defined in Figure 17 determine a strict total order in which compound expressions can be reduced. Therefore, This proof follows by induction over the the reduction of expressions. Cases (SELF-SEND) and (SEND) are trivial as they both reduce to `null` and do not modify the actor's heap, O. Case (NEW) reduces to a fresh α-equivalent reference. An α-equivalent object o' is also added to the heap of the actor. Cases (INVOKE), (FIELD-ACCESS), and (FIELD-UPDATE) follow by induction. If $\iota_a.\iota_o$ is α-equivalent and the referenced object in the heap is also α-equivalent, then resulting expressions and updated heaps will also be α-equivalent. Case (LET) follows by induction, if all possible v are α-equivalent then the substitution $[v/x]e$ will also be α-equivalent. Case (SPAWN) reduces to a fresh α-equivalent identifier $\iota_{a'}.\iota_o$. Case (CONGRUENCE) follows by induction over the actor-local reduction rules. Case (RECEIVE) does not apply as (RECEIVE) $\notin \xrightarrow{\iota_a}{}^*_k$ using definition 6.1. □

Definition 6.3 (Error). If no reduction rule is applicable to further reduce a reducible expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its current expression is fully reduced to a value and its mailbox is empty.

THEOREM 6.4 (THEOREM OF LIVENESS). $\forall K = A \uplus \{\mathcal{A}\langle \iota_a, \mu, e, O\rangle\}, e \notin \textbf{Value}, \exists K' : K \xrightarrow{\iota_a}_k K'$

PROOF OF LIVENESS. This proof follows by induction over the reduction of expressions. Cases (LET), (NEW), (INVOKE), (FIELD-ACCESS), (FIELD-UPDATE), (SELF-SEND), (SEND) and (SPAWN) all operate on reducible expressions and can be applied in any error-free program following definition 6.3. Case (CONGRUENCE) follows by induction over the actor-local reduction rules. Case (RECEIVE) does not apply as (RECEIVE) $\notin \xrightarrow{\iota_a}_k$ using definition 6.1. □

In theorem 6.2 and theorem 6.4 we have shown that the Communicating Event Loop Model satisfies both safety and liveness for the execution of a single turn and therefore the Isolated Turn Principle holds.

7 ACTOR MODEL PROPERTIES

Each of the four families discussed gives some indication of the properties of the actor system. However, these properties still remain largely dependent on the specific implementation of the actor system. In this section we give a complete overview. There are four main classes of features and properties. First we look at how each system **processes individual messages**. Secondly, we look at **how messages are received** by the actor. Thirdly, we look at what mutable state is available in the actor system and how the actor system handles **state changes**. Lastly, we classify the different actor systems according to the **granularity in which actors are meant to be used** within a single execution environment.

7.1 Message Reception

Incoming messages are always stored in the inbox of an actor. How those messages are taken out of that inbox can differ between the different actor systems.

Interface. At any given point during the execution of a program, the interface of an actor is the set of messages that actor understands. This interface can either be flexible or fixed throughout the lifetime of that actor. An actor can have a single entry point or, in the case of the Communicating Event Loop Model, have multiple entry points, each with their own subset of the actor's interface. This helps support a POLA (principle of least authority) style of programming, by facilitating the creation of many small, object-level interfaces, rather than a single large actor-level interface.

Order. In the case of a fixed interface, it makes sense to process messages in the same order they arrived in the inbox of the actor. However, when the interface is flexible it can be opportune to process messages out of order (similar to Out of Order Execution, *OoOE*) depending on what messages are supported by the behaviour that is in place at the start of each turn. Support for flexible interfaces and out of order message processing and thus facilitate what is known as "conditional synchronisation" [9] (e.g. implementing a blocking bounded buffer, or other more complex forms of synchronisation). The downside of a flexible interface is that it can lead to communication deadlocks [34] where an actor is waiting indefinitely without processing any other messages in its inbox.

State. The state of an actor may consist of multiple individually addressable variables, each holding simple atomic values (e.g. numbers), composite values (e.g. a list of numbers) or references to other actors. State changes can be aggregated or on an individual, i. e. per variable, basis. If the sequential subset of the actor system is functional then state changes are typically aggregated by replacing an actor's behaviour, any composite value is also immutable and can be transmitted between actors by reference. If the sequential subset of the actor system is imperative then state changes can be made on an individual basis. Any composite value needs to be copied or proxied before being transmitted over actor boundaries.

A summary of the different actor models discussed in this paper and their properties can be found in Table 1.

	Interface	Entry Point	Order	State Changes	Message Passing
Classic Actor Model	Flexible	Single	OoOE	Aggregated	By Reference
Processes	Flexible	Single	OoOE	Aggregated	By Reference
Active Objects	Fixed	Single	FIFO	Individual	Copied
Communicating Event Loops	Fixed	Multiple	FIFO	Individual	Proxied

Table 1. Message Reception Properties

8 CONCLUSION

At its core, the actor model of computation is a very simple programming model with one central concept: the actor. Actors are simply isolated processes that communicate asynchronously via message passing. The main benefit of this strict isolation and asynchronous communication is that the actor model is free of low-level data races and deadlocks. In this paper we define this as a key unifying principle among all actor systems and name it the isolated turn principle. However, when looking at the broader picture, half a century of research has lead to a plethora of different variations on the actor model, each with their own widely different properties.

This paper focusses primarily on the overlying programming model and identifies and defines four broad families along which any actor system can be categorised. Namely: *Classic Actors, Active Objects, Processes and Communicating Event Loops*. This paper provides a brief history of some of the key programming languages and libraries that implement each of these four broad families of actor models. These programming languages and libraries have influenced and will continue to influence the design and rationale of other actor systems today.

This paper provides a precise formal definition for a core subset of the four families of actor models through an operational semantics. An executable version of these operational semantics implemented in PLT Redex is available. This paper proves that the isolated turn principle holds for each of these four variations. The Isolated Turn Principle is a core principle that unifies all actor systems. However, there are many other properties along which an actor system can be classified. While those properties remain largely dependent on the specific implementation of an actor system, there are still some general conclusions to be drawn.

- The sequential subset of Classic Actors and Processes is typically functional, while the Active Object Model and Communicating Event Loops are typically imperative. However, as long as the Isolated Turn principle is upheld, the choice does not really impact the concurrency properties of that system.
- Classic Actors and Processes have a flexible interface which facilitates conditional synchronisation, while the Active Object Model and Communicating Event Loops have a fixed interface. Communicating Event Loops is the only actor model where actors can create many small interfaces which supports a PoLP (principle of least privilige) style of programming.
- Actor models where actors have a flexible interface typically have support out-of-order execution of messages while actor models where actors have a fixed interface typically process messages in FIFO order.
- The paradigm of the sequential subset directly determines whether state changes at the level of an actor are aggregated (for functional languages) or on a per-variable basis (for imperative languages).

REFERENCES

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
[2] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, Sept. 1990.
[3] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, Jan. 1997.

[4] J. Allen. *Effective Akka: Patterns and Best Practices*. " O'Reilly Media, Inc.", 2013.
[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.
[6] F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5), oct 2017.
[7] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. Tapia Tarifa, T. Wrigstad, and A. M. Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In M. Bernardo and E. B. Johnsen, editors, *Formal Methods for Multicore Programming (SFM 2015)*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.
[8] J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the smalltalk-80 environment. pages 109–129. University Press, 1989.
[9] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, Sept. 1998.
[10] S. Bykov. The curse of the a-word. https://temporal.io/blog/sergey-the-curse-of-the-a-word, 2021. Accessed: 2023-01-01.
[11] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. *SIGPLAN Not.*, 39(1):123–134, jan 2004.
[12] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.
[13] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag.
[14] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992.
[15] I. Greif and C. Hewitt. Actor semantics of planner-73. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '75, page 67–77, New York, NY, USA, 1975. Association for Computing Machinery.
[16] P. Haller and M. Odersky. Actors that unify threads and events. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.
[17] C. Hewitt. Planner: A language for manipulating models and proving theorems in a robot. 1970.
[18] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
[19] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.
[20] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. Abs: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, Austria, November 29-December 1, 2010. Revised Papers 9*, pages 142–164. Springer, 2012.
[21] D. Kafura. Act++: Building a concurrent C++ with actors. *J. Object Oriented Program.*, 3(1):25–37, Apr. 1990.
[22] K. Kahn, E. D. Tribble, M. Miller, and D. G. Bobrow. *Vulcan: Logical Concurrent Objects*, page 75–112. MIT Press, Cambridge, MA, USA, 1987.
[23] W. Kim. Thal: An actor system for efficient and scalable concurrent computing, 1997.
[24] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.
[25] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*, TGC'05, pages 195–229, Berlin, Heidelberg, 2005. Springer-Verlag.
[26] M. Research. Orleans, 2015-16.
[27] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
[28] C. Scholliers, E. Tanter, and W. De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program.*, 80:52–64, Feb. 2014.
[29] M. Sirjani, A. Movaghar, A. Shali, and F. S. De Boer. Modeling and verification of reactive systems using rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.
[30] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.
[31] D. Sturman and G. Agha. A protocol description language for customizing failure semantics. In *Reliable Distributed Systems, 1994. Proceedings., 13th Symposium on*, pages 148–157, Oct 1994.
[32] G. J. Sussman and G. L. Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.
[33] C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha. Rosette: An object-oriented concurrent systems architecture. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-based Concurrent Programming*, OOPSLA/ECOOP '88, pages 91–93, New York, NY, USA, 1988. ACM.

[34] C. Torres Lopez, S. Marr, E. Gonzalez Boix, and H. Mössenböck. *A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs*, pages 155–185. Springer International Publishing, Cham, 2018.

[35] T. Van Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, SCCC '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[36] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, Dec. 2001.

[37] C. A. Varela. *Programming Distributed Computing Systems: A Foundational Approach*. The MIT Press, 2013.

[38] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming ABCL/1. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 258–268, New York, NY, USA, 1986. ACM.