43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties

Joeri De Koster
Vrije Universiteit Brussel
Pleinlaan 2
1050 Elsene, Belgium
jdekoste@vub.ac.be

Tom Van Cutsem
Nokia Bell Labs
Copernicuslaan 50
2018 Antwerp, Belgium
tom.van_cutsem@nokia-bell-labs.com

Wolfgang De Meuter
Vrije Universiteit Brussel
Pleinlaan 2
1050 Elsene, Belgium
wdemeuter@vub.ac.be

Abstract

The Actor Model is a message passing concurrency model that was originally proposed by Hewitt et al. in 1973. It is now 43 years later and since then researchers have explored a plethora of variations on this model. This paper presents a history of the Actor Model throughout those years. The goal of this paper is not to provide an exhaustive overview of every actor system in existence but rather to give an overview of some of the exemplar languages and libraries that influenced the design and rationale of other actor systems throughout those years. This paper therefore shows that most actor systems can be roughly classified into four families, namely: *Classic Actors, Active Objects, Processes and Communicating Event-Loops*. This paper also defines the Isolated Turn Principle as a unifying principle across those four families. Additionally this paper lists some of the key properties along which actor systems can be evaluated and formulates some general insights about the design and rationale of the different actor families across those dimensions.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages; D.1.3 [*Concurrent Programming*]: Parallel programming

General Terms Languages, Theory

Keywords Actor Model, Concurrency

1. Introduction

The Actor Model is a concurrency model that was originally proposed by Hewitt et al. in 1973 [13]. Since then it has been used as the model of concurrency for both academic research languages as well as industrial strength programming languages and libraries.

Because actors are often strictly isolated software entities and because of the asynchronous nature of its communication mechanism, they avoid common concurrency issues such as low-level data races and deadlocks by design. Over the years these properties have made the Actor Model into an interesting concurrency model to be used both for exploiting fine-grained concurrency in massively parallel workstations as well as for exploiting more coarse-grained concurrency in a distributed setting. More than 40 years later, there now exist bountiful implementations of the Actor Model in various programming languages and libraries. Each of these implementations defines a slightly different flavour of the Actor Model. This paper has three main contributions.

Firstly, because of all the variations on the Actor Model, over the years researchers have employed different terms to describe the different concepts that make up the Actor Model. One of the contributions of this paper is to define a common nomenclature for each of the key concepts of the Actor Model. While a precise definition would require a formal grounding, for this work, we currently limit ourselves to an informal definition for each of the terms and concepts. A list of terms and their definitions can be found under Section 2.

Secondly, while each of these variations on the Actor Model builds on top of a common substrate, in this paper we identify and define four broad families along which each actor system can be categorised. Namely: *Classic Actors, Active Objects, Processes and Communicating Event-Loops*. We provide an overview of the history of actor systems and a definition for each of the different families along which every actor system can be categorised in Section 3.

Thirdly, categorising an actor system along one of these four families gives some indication of the properties and structure of that actor system. However, the actual properties of that system still remain largely dependent on the specific implementation of the actor system. We present a number

of properties along which an actor system can be evaluated in Section 4 and formulate some general insights about the design and rationale of the different actor families across those dimensions.

2. Terminology and Definitions

Before we further delve into the history of the Actor Model we must first start by establishing a common terminology by which we can denominate the different concepts defined by an actor system. In this section we give an overview of the different terms used for each of the key concepts found in every actor system. We also provide an informal definition for each of these concepts. In the following sections we will mix these terms depending on the language being discussed.

***message,* envelope, event, request** A message is the unit of communication between different actors. A message is a combination of an identifier that defines the type of message and a payload that contains additional information sent with that message. If one actor sends a message to another actor, that message is stored in the latter actor's inbox, independent of the recipient actor's current processing state.

***inbox,* mailbox, message queue, event queue** The inbox of an actor stores an ordered set of messages received by that actor. While the inbox defines the order in which the messages were received, that does not necessarily imply that those messages are processed by that actor in that order.

***turn,* epoch, step** A turn is defined as the processing of a single message by an actor. In other words, a turn defines the process of an actor taking a message from its inbox and processing that message to completion.

interface At any given point in time, an actor's interface defines the list and types of messages it understands. An actor can only process incoming messages that fit this interface. For some actor systems this interface is fixed while other actor systems allow an actor to change its interface, thus allowing it to process different types of messages at different points in time.

state At any given point in time, we define an actor's state as all the state that is synchronously accessible by that actor (i. e. state that can be read or written without blocking its thread of control). Depending on the implementation, that state can be mutable or immutable, and isolated or shared between actors.

behaviour A behaviour is a common term to denote the combination of an actor's interface and its state. Some actor systems enable an actor to modify its entire behaviour in one single operation.

***actor,* active object, activity, vat, grain** An actor can be defined as a four-tuple: an execution context, an inbox, an interface and its state. An actor perpetually takes messages from its inbox and processes them in a new execution context with respect to that actor's interface and state. This continues until the inbox is empty after which the actor goes to an idle state until a new message arrives in its inbox.

actor system An actor system is a language or library that implements the Actor Model as an abstraction for concurrency. Every actor system enables the creation and concurrent execution of actors.

3. History of Actor Systems

The Actor Model was originally proposed by Hewitt et al. in 1973 [13] in the context of artificial intelligence research at MIT. 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. In October of 1975 Hewitt and Smith [12] wrote a primer on a language called PLASMA, the first language implementation of the Actor Model. 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 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)`. The Actor Model only became more widely regarded as a general-purpose concurrency model when it was recast in terms of three simple primitives by Gul Agha [1, 2] in 1986. 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.

This section gives an overview of a small but representative selection of the ancestry of actor systems from 1973 until today. We classify the different actor languages into four major paradigms: Classic Actors, Active Objects, Processes and Communicating Event-Loop actors.

3.1 The Classic Actor Model

Agha [1, 2] recasts the Actor Model as a concurrent object-oriented programming model. The main focus 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.

The example in Listing 2 is written in the Rosette actor language [19] which was based on this model.

```
(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.

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 inbox 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 methods that are understood by the actor. A method is specified by a keyword followed by a table of arguments. In this case the `put` method expects a value for the new content. Afterwards follows the body that specifies how each method 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 a lot of 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.

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 state of an actor are aggregated in a single become statement. Actors have a flexible interface that can similarly be changed by switching the behaviour of that actor.

However, there are many other library implementations of this model for different languages.

The sequential subset of an actor model is the subset of instructions 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.

3.1.1 The Isolated Turn Principle

The semantics of the Classic Actor Model enable 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 message from its inbox. This means that 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 over 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 object 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 [17] 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.

The Isolated Turn Principle guarantees that the Actor Model is free of low-level data races and deadlocks. However, these properties only apply for the processing of a single message, once you broaden that boundary to 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 Active Objects

Around the same time that Agha reformulated Hewitt's actors in terms of OOP, Yonezawa [23] worked on a object-oriented concurrent programming language called ABCL/1. In this language, each object has its own thread of control and may have its own local persistent memory. In this model state changes are not specified in terms of behaviour updates (become) but rather by individual assignment statements. To maintain actor isolation, the state of each active object is only accessible 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[1].

There are three types of messages in ABCL/1: *past*, *now* and *future*. *Past type* messages are sent to the receiver and immediately return. The sender does not wait for the receiver to process the message before continuing its current computation. This message type corresponds to the standard way of message passing in Classic Actors. *Now type* messages are very similar to procedure call and return. When an object `O` sends a *now type* message to another object `T`, `O` will wait for `T` to process that message and send back a result before continuing with its current computation. *Future type* messages are used when the sender of a message does not need the result of the message immediately. In other actor models, the sender of a request has to finish its computation before being able to receive the response from the receiver. If sending this request and processing the result is part of the sender's task, this often leads to an unnatural breakdown of that task in different execution steps. ABCL/1's futures was the first attempt to solve that issue.

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

Cell <= [:get]
```

Listing 3. An active object in ABCL/1.

ASP In ABCL/1 every object is an active object. This makes it very suitable for exploiting fine-grained concurrency. Asynchronous Sequential Processes (ASP)[8] is a programming model similar to ABCL/1 that enables a more coarse-grained use of active objects. Contrary to ABCL/1, not every object in this model is an active object. Rather, actors in this model are represented by an *activity*. Each activity has a single root object called the *active* object. Every other object that is encapsulated by that activity is called a *passive* object. Different activities do not share memory, the active objects' whole object graph is deep-copied into the activity. The copied objects are called passive objects. Any method call on an active object will result in an asynchronous request being sent to the activity and returns a future. The request is stored in a request-queue and is called *pending*. Later this request will be *served* and when it is finished the request is *calculated* and the future is replaced with a (deep) copy of the return value. Similarly to ABCL/1's futures, execution will block if any attempt is made to perform a strict operation (e. g., a method call) on such a future. Execution resumes when the corresponding request is calculated. Isolation of the different activities is guaranteed by passing passive objects by copy between the different activities. All references to passive objects are always local to an activity and any method call on a passive object is synchronously executed. An implementation of this model can be found in ProActive [6], which is a framework for Java.

SALSA SALSA [21] is another actor language that implements the Active Objects model on top of Java. The implementation translates SALSA code into Java code that can

[1] Barring the use of ABCL's *express messages*, which can potentially interrupt the processing of a message and thus violate this principle [23].

Active Objects

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.

be compiled together with the SALSA actor library to Java byte-code and run on any JVM. SALSA was proposed as an actor-based language for mobile and internet computing and has support for mobile actors which enables distributed systems reconfiguration. A few of the other main contributions include actor garbage collection and three new language mechanisms to help coordinate asynchronous communication between different actors. When an actor sends an asynchronous message to another actor, that actor may include an implicit *customer* to which the result should be sent after the message has been processed. This can be done by using one of three kinds of continuations, namely token-passing continuations, join continuations and first-class continuations. These continuations enable high-level synchronization patterns to be specified, without the drawback of futures[2] whereby actors can potentially block waiting for the future to be resolved. These token-passing continuations can be implemented or specified as "partial" messages [22]. These partial messages are stored in a separate mailbox which represent continuations that will be processed after tokens get resolved.

Orleans A more recent industry-strength addition to the active objects family of actor languages is the Orleans .NET framework [15]. Orleans is a framework aimed at building distributed high-scale computing applications. It was created by Microsoft Research and designed for use in the cloud. The Orleans runtime schedules execution of a large number of actors across a custom thread pool with a thread per physical processor core. This makes Orleans highly suitable for exploiting fine-grained concurrency. In Orleans, actors are called *grains*. The implementation of a grain specifies a class for which the methods are only asynchronously available through a proxy object. Calling such an asynchronous message returns a task and Orleans inherits the `await` keyword from C# in order to join with asynchronously executing tasks.

3.3 Processes

Erlang [5] was the first industry-strength language to adopt the actor model as its model of concurrency. It was developed at the Ericsson and Ellemtel Computer Science Laboratories as a declarative language for programming large industrial telecommunications switching systems. While Erlang's programming style is very close to that of Classic Actors, it uses different mechanics to achieve similar effects. An actor is not modelled as a named behaviour. Rather actors are modelled as processes that run from start to completion. Erlang actors can use the primitive `receive` to specify what messages an actor can receive at that moment in time. 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. This is illustrated by Listing 4.

```
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.

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.

Scala Actor Library The Scala Actor Library [11] offers a full-fledged implementation of Erlang-style actors on top of Scala. Scala Actors can use two different primitives to receive a message. On the one hand, `receive` suspends the current actor together with its full stack frame until a message is received. Once the message is received the actor can continue processing that message and the context in which the `receive` block was executed is not lost. On the other hand, `react` suspends the actor with just a continuation closure. This closure only contains information on how to proceed with processing an incoming message. The context in which the `react` was called is lost. This type of message processing has the benefit of being more lightweight because it decouples the actor from its underlying thread of control, allowing a single thread to execute many actors, allowing the actor system to scale to a much larger number of actors

Kilim Kilim [18] is an actor framework for Java. The Kilim weaver post-processes Java byte-code to add a lightweight implementation of processes and provide strong isolation guarantees. Each actor class needs to specify an `execute` method as entry point for the actor. We categorise Kilim as belonging to the Processes family of actor systems in our

[2] This is only a drawback for blocking futures as can be found in ABCL and ASP. The E programming language has support for non-blocking promises which do not exhibit this drawback.

Processes

Every process runs from start to 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.

taxonomy because a Kilim actor is modelled as a process that runs from start to completion. Getting a message from an actor's inbox is a blocking operation that in analogous to executing a receive statement. The Kilim weaver makes sure that context switching is possible during the execution of any method that is annotated with the `@pausable` annotation, for example, when an actor is waiting for a new message to arrive in its inbox. Contrary to Erlang, the sequential subset of Kilim is not functional and supports any Java statement. To ensure isolation of the different actors, objects that are transmitted over a mailbox have to implement the `Message` interface and are passed by copy. However, there exist extensions to Kilim's type system to enable zero-copy message passing [10]. As long as the type system can guarantee that any "shared" object is only accessible from within a single actor then race conditions can be avoided. The use of linear type systems to introduce shared-state in an actor model in a safe and efficient ways is not only limited to Kilim. Pony [9] also employs linear types to avoid deep copying messages between actors.

3.4 Communicating Event-Loops

The E programming language [16] was the first language to introduce *the Communicating Event-Loop Actor Model*. This model takes a very similar approach to Asynchronous Sequential Processes with the exception that it does not make a distinction between passive and active objects. In this model, each actor is represented as a *vat*. A vat has a single thread of control (the event-loop), a heap of objects, a stack, and an event queue. Each object in a vat's object heap is *owned* by that vat and those objects are owned by exactly one vat. Within a vat, references to objects owned by that same vat are called *near references*. References to objects owned by other vats are called *eventual references*.

The type of reference determines the access capabilities of that vat's thread of execution on the referenced object. 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 eventual 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

Communicating Event-Loops

A vat is a combination of an object heap, an event queue and an event loop. Every reference that is passed between different vats is exported as an eventual 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 vats by eventual reference.

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 can define multiple objects that all share the same message queue and event-loop and hand out different references to those objects, thus essentially allowing one to model an actor that has multiple interfaces at the same time. The example in Listing 5 illustrates how to create an object in E and send it an eventual message `get`.

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

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

Listing 5. A vat in E.

When an object in one vat sends an eventual message to an object in another vat the message is enqueued in the event queue 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 a vat to use a promise as a near reference. If a vat wants to make an immediate call on the value represented by a promise, like printing it on the screen, that vat 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 [14]. 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.

AmbientTalk The communicating event-loop model was later adopted by AmbientTalk [20], a distributed object-oriented programming language which has been designed for developing applications on mobile ad hoc networks. AmbientTalk was designed as an *ambient-oriented programming* (AmOP) language.It adds to the Actor Model a number 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.

4. Actor System 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 an overview of all the features and properties we use to classify the different actor systems discussed in Section 3. 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.

4.1 Message Processing

An important part of any actor system is the way in which messages are processed. This is what we referred to earlier as the sequential subset of the language. An important side-note here is that, any property that holds for the sequential subset of the language, typically only holds for the processing of a single message. For example, any actor system that upholds the Isolated Turn Principle guarantees that each message is processed sequentially and fully isolated. However, once you broaden that boundary to the processing of several messages, most of these properties no longer hold. In this section we only consider properties that hold during the processing of a single message.

Paradigm The sequential subset of an actor language can either be functional or imperative. If it is functional then, typically, the only way to modify state is to change the behaviour of the actor. If it is imperative then that means that extra care needs to be taken to guarantee isolation of the different actors. If the Isolated Turn Principle is guaranteed, then the choice of paradigm does not impact the concurrency properties of the resulting model.

Continuous The sequential subset of a language can allow blocking statements or can ensure a continuous processing of each message. In the latter case actors are guaranteed to process a message from start to completion without having to worry about deadlocks. Again, this only applies to the processing of a single message. Other forms of lost progress can still occur between the processing of different messages.

	Message Processing		
	Paradigm	Continuous	Consecutive
Classic Actor Model			
Agha (ACT, SAL, Rosette)	Functional	Continuous	Consecutive
Akka	Imperative	Blocking	Consecutive
Processes			
Erlang	Functional	Continuous	Consecutive
Scala Actor Library	Imperative	Blocking	Consecutive
Kilim	Imperative	Blocking	Consecutive
Active Objects			
ABCL/1	Imperative	Blocking	Interleaved
ASP (ProActive)	Imperative	Blocking	Consecutive
SALSA	Imperative	Continuous	Consecutive
Orleans	Imperative	Continuous	Consecutive
Communicating Event-Loops			
E	Imperative	Continuous	Consecutive
AmbientTalk	Imperative	Continuous	Consecutive

Table 1. Message Processing Properties

Consecutive We consider a message to be processed in consecutive order if it is processed from start to completion without being interleaved with the processing of other messages of the same actor. This is usually guaranteed unless there is some way to interrupt the processing of a single message (e. g. express messages in ABCL/1).

A summary of the different actor languages discussed in this paper and their properties with respect to message processing can be found in Table 1. In conclusion, we list a number of observations made from this table. Firstly, the Isolated Turn principle is a unifying principle across all actor families. On the one hand, we have actor languages which mostly support this principle. The major benefit of this is that the developer gets strong safety guarantees: low-level data races are ruled out by design. On the other hand, we find actor libraries, which are often built on top of a shared-memory concurrency model and typically do not enforce actor isolation, i. e., they cannot guarantee that actors do not share mutable state, therefore violating the Isolated Turn Principle. Secondly, the sequential subset of an actor system can be functional or imperative, and as long as the Isolated Turn principle is upheld, the choice does not impact the concurrency properties of that system. Finally, having continuous message processing guarantees that the the sequential subset of the actor system is free of low-level deadlocks. However, this does not guarantee global progress of the system as deadlocks can still occur when two or more actors are waiting for a message to arrive. What it does guarantee is that an actor cannot be blocked while processing a message. Finally, if we do not consider express messages from ABCL, every actor system processes messages in consecutive order.

4.2 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. In this section we discuss some properties of actor systems according to

how they take messages out of their inbox before processing them.

Interface The interface (i. e. behaviour) of an actor can be specified in various ways. Some actor systems specify a behaviour as a list of messages and processing instructions, to be called implicitly when a matching message is available. Others use special primitives such as `receive` to let the actor proactively take a message from their inbox. Others use an object-oriented style where the interface of the actor corresponds to the interface of an object and a message send corresponds to a method invocation.

Flexibility The interface to an actor can be fixed or flexible. When the interface of an actor is fixed that means that actor will always understand the same set of messages throughout its lifetime. When the interface is flexible, the set of messages an actor understands can vary over time. This is typically done by changing the behaviour of the actor. However, this does not imply that changing the behaviour of an actor has to somehow change its interface. A behaviour is stateful and an actor can change its behaviour to update its internal state without changing what type of messages it understands. Similarly, having a fixed interface does not imply that actors always respond to a message in the same way. With an imperative sequential subset it is possible to change how an actor responds to a message depending on earlier state updates.

Number of interfaces Traditionally, every actor has a single addressable entry point, namely the interface (i. e., the behaviour) of that actor. However, in the case of the communicating event-loop model, each actor can hand out many references to multiple of its own objects, creating multiple addressable entry points each with a potentially different 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. Finally, actors in the E programming language processes messages in *E-ORDER* where the order of messages is preserved when some messages are first sent trough a forwarding actor.

A summary of the different actor languages discussed in this paper and their properties with respect to message reception can be found in Table 2. In conclusion, we list a number of observations made from this table. Firstly, Classic Actors and Processes provide the best support for flexible interfaces and out of order message processing and thus facilitate what is known as "conditional synchronisation" [7] (e.g. implementing a blocking bounded buffer, or other more complex forms of synchronisation). Secondly, actor systems developed within an object-oriented paradigm tend to support fixed actor interfaces in combination with imperative behaviour, while actor systems developed within a functional paradigm tend to support flexible actor interfaces in combination with a purely functional behaviour. Finally, Communicating Event Loops is the only actor family that allows multiple addressable entry points to a single actor. 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.

	Message Reception			
	Interface	Flexibility	# Interfaces	Order
Classic Actor Model				
Agha (ACT, SAL, Rosette)	Behaviour	Flexible	1	OoOE
Akka	Behaviour	Flexible	1	OoOE
Processes				
Erlang	Receive	Flexible	1	OoOE
Scala Actor Library	Receive	Flexible	1	OoOE
Kilim	Mailbox	Flexible	1	FIFO
Active Objects				
ABCL/1	Methods	Fixed	1	FIFO
ASP (ProActive)	Methods	Fixed	1	FIFO
SALSA	Methods	Fixed	1	FIFO*
Orleans	Methods	Fixed	1	FIFO
Communicating Event-Loops				
E	Methods	Fixed	*	E-ORDER
AmbientTalk	Methods	Fixed	*	FIFO

Table 2. Message Reception Properties

4.3 State Changes

Regardless of whether their sequential subset is functional or not, all implementations of the Actor Model have some form of mutable state (e. g. the behaviour/inbox of an actor).

Granularity 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. If the sequential subset of the actor system is imperative then state changes can be made on an individual basis.

Isolation Isolation is guaranteed when no two actors can read-write or write-write to the same memory location. In actor systems where the sequential subset is functional this is guaranteed because in those languages the only mutable state is the behaviour of an actor and actors are only able to modify their own behaviour. In actor systems where the sequential subset is imperative some extra care needs to be taken when sharing mutable state. For example, by (deep) copying any data structure when it is passed between actors. Actor systems designed as libraries on top of execution environments that support shared-memory multithreading typically cannot guarantee isolation.

A summary of the different actor languages discussed in this paper and their properties with respect to state changes

	State Changes	
	Granularity	Isolation
Classic Actor Model		
Agha (ACT, SAL, Rosette)	Aggregated	Isolated
Akka	Individual	Shared-Memory
Processes		
Erlang	Aggregated	Isolated
Scala Actor Library	Individual	Shared-Memory
Kilim	Individual	Isolated
Active Objects		
ABCL/1	Individual	Isolated
ASP (ProActive)	Individual	Isolated
SALSA	Individual	Isolated
SALSA	Individual	Shared-Memory
Communicating Event-Loops		
E	Individual	Isolated
AmbientTalk	Individual	Isolated

Table 3. State Changes Properties

	Actors per Execution Environment
	Granularity
Classic Actor Model	
Agha (ACT, SAL, Rosette)	Fine-grained
Akka	Fine-grained
Processes	
Erlang	Fine-grained
Scala Actor Library	Fine-grained
Kilim	Fine-grained
Active Objects	
ABCL/1	Coarse-grained
ASP (ProActive)	Coarse-grained
SALSA	Coarse-grained
Orleans	Fine-grained
Communicating Event-Loops	
E	Coarse-grained
AmbientTalk	Coarse-grained

Table 4. Actors Per Execution Environment Properties

can be found in Table 3. In conclusion, we list a number of observations made from this table. Firstly, the paradigm of the sequential subset (functional or imperative) seems to directly determine whether state changes at the level of an actor are aggregated (for functional languages) or on a per-variable basis (for imperative languages). Secondly, actor systems generally ensure state changes are isolated per turn. The general exception are actor systems designed as libraries on top of execution environments that support shared-memory multithreading.

4.4 Actors Per Execution Environment

The original intention for the Actor Model was to provide a programming model for expressing concurrent programs over different nodes in a distributed network. The message passing model and isolation of the different actors is a good fit for such systems. As such, most actor systems include support for distribution. However, where they do differentiate is how actors were meant to be used on a single node. This ranges from Erlang, which is known for its lightweight implementation of actors and supposed to run many actors in a single execution environment, to AmbientTalk, which is an actor language designed for mobile applications where the execution environment in each phone would typically host only a handful of actors.

A summary of the different actor languages discussed in this paper and their properties with respect to the granularity of their concurrency model can be found in Table 4. We found that Classic Actors and Processes generally lead to a style of programming where each individual actor is at the level of abstraction of what an OO programmer would think of as an "object", or a functional programmer would think of as an "abstract data type" (ADT), whereas for Active Objects and CELs, each actor is more at the level of abstraction of what an OO programmer would think of as a "component", or a functional programmer as a "module".

It is our conjecture that Classic Actors and Processes lead to a style of programming where the state of an actor tends to be small in terms of the number of values involved (not necessarily in terms of the size of the values involved), whereas Active Objects and CELs lead to a style of programming where the state of an actor tends to consist of many small objects, like in a regular heap of an object-oriented program.

5. Conclusion

Over the years different actor systems have used different terminology to name certain concepts. This paper provides an informal definition of a nomenclature of a common substrate by which we can name different concepts found in every actor system. Additionally, this paper provides a brief history of some of the key programming languages and libraries that implement the actor model and have influenced and will continue to influence the design and rationale of other actor systems today. While discussing these different actor systems, we identify and define four broad categories along which any actor system can be categorised. Namely: *Classic Actors, Active Objects, Processes and Communicating Event-Loops*.

We also define the Isolated Turn principle as a unifying principle across all actor families. The Isolated Turn Principle guarantees that the Actor Model is free of low-level data races and deadlocks and guarantees that, during a single turn, an actor has a consistent view over its state and its environment. Contrary to actor libraries that are built on top of a shared memory concurrency model and typically cannot enforce actor isolation, we identified that most actor languages support this principle.

The properties of an actor system remain largely dependent on the specific implementation of that actor system. Therefore this paper defines a number of important properties along which every actor system can be evaluated. However, there are still some general conclusions to be drawn from our evaluation of the different actor systems discussed in this paper. Firstly, the sequential subset of an actor system can be functional or imperative, and as long as the Isolated Turn principle is upheld, the choice does not impact the concurrency properties of that system. Secondly, actor systems developed within an object-oriented paradigm for their sequential subset tend to support fixed actor interfaces in combination with imperative behaviour, while actor systems developed within a functional paradigm tend to support flexible actor interfaces in combination with a purely functional behaviour. The flexible interface of the latter fa-

cilitates conditional synchronisation. Thirdly, 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). Finally, it is our conjecture that Classic Actors and Processes are mostly used in a fine-grained concurrency setting and lead to a style of programming where the state of an actor tends to be small in terms of the number of values involved (not necessarily in terms of the size of the values involved), whereas Active Objects and CELs are mostly used in a a coarse-grained concurrency setting and lead to a style of programming where the state of an actor tends to consist of many small objects, like in a regular heap of an object-oriented program.

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*. 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] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, 2006.

[7] 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.

[8] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, Apr. 2009.

[9] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA, 2015. ACM.

[10] O. Gruber and F. Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 281–301, Berlin, Heidelberg, 2013. Springer-Verlag.

[11] 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.

[12] C. Hewitt and B. Smith. A plasma primer (draft), 1975.

[13] 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, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[14] 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.

[15] Microsoft. Orleans, 2015-16. URL `http://dotnet.github.io/orleans`.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.