# Decoupling Isolation and Concurrency: An Actor-Centric View of Behaviour-Oriented Concurrency

Luke Cheeseman[1], Elias Castegren[1], Sophia Drossopoulou[2], Tobias Wrigstad[1], Sylvan Clebsch[3], and Matthew Parkinson[3]

[1] Uppsala University, Sweden first.last@it.uu.se
[2] Imperial College, UK s.drossopoulou@imperial.ac.uk
[3] Azure Research, USA and UK {sylvan.clebsch, mattpark}@microsoft.com

**Abstract** The actor model is an elegant concurrency model with the actor as the central concept. An actor encapsulates a thread of control and isolated state, communicating with other actors exclusively via message passing. This makes reasoning about the behaviour of a single actor simple, but, due to the coupled units of isolation and concurrency, performing atomic operations involving multiple actors becomes harder. Recent work on behaviour-oriented concurrency mitigates this by explicitly decoupling isolation and concurrency.

In this paper we explore the connections between the actor model and behaviour-oriented concurrency and the effects of (de)coupling the units of isolation and concurrency. We derive the semantics of behaviour-oriented concurrency by starting from a semantics of the actor model and gradually decoupling isolation and concurrency. We show that behaviour-oriented concurrency generalises the actor model by proving a simulation theorem: a program in the actor model has a corresponding program using behaviour-oriented concurrency.

## 1 Introduction

The actor paradigm was introduced in 1973 by Carl Hewitt et al. [16], formalised by Irene Greif [13] in 1975, and again by Gul Agha [1] in 1986 as a computational model for handling concurrency in distributed systems. It has also been used as an elegant model for single-node concurrency. The actor is the fundamental unit of computation in the paradigm; an actor encapsulates a single-threaded control loop, isolated state, and an interface through which other actors can communicate via asynchronous message passing. The goal of the paradigm is to simplify the many complexities of concurrent programming into one concept: sending messages to actors. This simplification provides structure to concurrent code, improving the programmer's ability to understand and reason about their program.

We can view the actor paradigm as marrying together an object with concurrency in object-oriented programming [2]. Typically in object-oriented programming, message-passing (or "calling a method"), is handled synchronously in

the calling thread. If we were to introduce multiple threads calling methods on an object, then we must expect, and prevent, data-races. Instead, in the actor paradigm, the actor uses its own thread to process the messages one at a time, thus freeing the programmer the concerns of synchronisation, data-races, *etc.*

This design intentionally creates a tight coupling between the *unit of isolation* and the *unit of concurrency*. In the actor context the unit of isolation is the fragment of the program state reachable only by a single actor, and the unit of concurrency is the single-thread control loop of an actor. This design entails that an actor's thread can only access that actor's state and conversely that an actor's state can only be accessed by that actor's thread. This is the key to the elegance and simplicity of the model: computation internal to an actor permits sequential reasoning.

Recent work on *behaviour-oriented concurrency* (BoC) demonstrates how tightly coupling isolation and concurrency can create tensions between atomicity and a program's inherent concurrency [9]. It introduces an alternative concurrency paradigm to address these challenges in a non-distributed setting. BoC draws inspiration from the actor model, but differentiates itself by decoupling the units of isolation and concurrency, relieving some of the tensions in program design. In this paper we demonstrate this inspiration by reimagining the actor model paradigm, decoupling isolation and concurrency, to arrive at BoC.

We make the following contributions:

- We highlight the inherent tensions of the actor model that comes from coupling concurrency and isolation (Section 2),
- We show how these tensions can be relieved by behaviour-oriented concurrency (Section 3),
- We derive the formal semantics of behaviour-oriented concurrency by starting from a formal actor semantics and modifying it until we arrive at behaviour-oriented concurrency (Section 4),
- We prove that the semantics for behaviour-oriented concurrency can simulate the actor semantics we started from (Section 5).

After the main contributions we discuss our results in Section 6 and conclude in Section 7.

## 2 Actors and the Tensions of Coupling Concurrency and Isolation

Coupling the units of concurrency and isolation leads to tension between the ability to reason atomically about updates to state, and the degree of concurrency inherent in a program. We can illustrate this tension in designing a solution to the classic bank transfer example. Take two bank accounts from which we can withdraw money, and to which we can deposit money. We want to be able to *atomically* transfer money from one account to another account. This means it should be not be possible for another operation involving either account to take place during the transfer, nor should we be able to lose money in the system.

```
actor Account
  var balance: U64
  var frozen: Bool = false

  new open(balance': U64) =>
    balance = balance'

  be withdraw(amount: U64) =>
    if ((not frozen) and (balance >= amount)) then
      balance = (balance - amount)
    end

  be deposit(amount: U64) =>
    if (not frozen) then
      balance = (balance + amount)
    end
```

Listing 2.1: A naive implementation of accounts using actors.

Listing 2.1 shows a naive implementation (written in Pony [19]) where each bank account is modelled as an actor. Two actors can concurrently operate on their accounts, but we are unable to express atomic transfer of money between accounts in a straightforward manner. Note that both `withdraw` and `deposit` can fail, either due to accounts being frozen or due to there not being enough funds for a withdrawal. If we instead modelled a bank as an actor encapsulating all of its accounts, we get atomic transfer of money straightforwardly, but operations on unrelated accounts will be serialised.

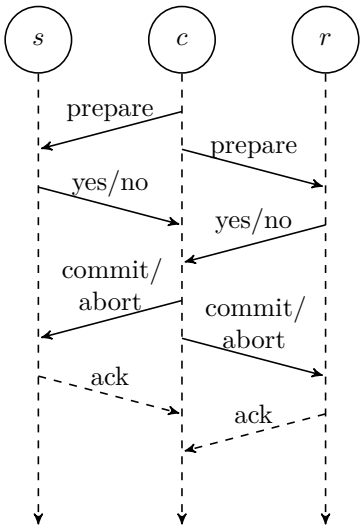



Figure 1: Two-phase commit

If we pursue the goal of highly concurrent programs, modelling each account as an actor, then we will need to build a protocol to ensure transfers can complete atomically. Two-phase commit is an often relied upon protocol to achieve atomic transfer between two actors; this involves a third actor to coordinate the transfer. Figure 1 shows two-phase commit with a coordinator $c$ synchronising accounts $s$ and $r$ (acks may be omitted when message failure cannot occur).

Listing 2.2 shows a Pony program implementing atomic transfers between accounts using two-phase commit. Code that is *not* related to synchronisation (the same as in Listing 2.1) is highlighted in blue. For a transfer scenario involving a `Coordinator` $c$, coordinating the transfer of the amount $a$, from `Account` $s$ to `Account` $r$ the two-phase commit proceeds as follows:

```
1  actor Coordinator
2    var from: Account
3    var to: Account
4    var amt: U64
5    var aborted: Bool = false
6    var from_ok: Bool = false
7    var to_ok: Bool = false
8
9    new create(from': Account,
10              to': Account,
11              amt': U64) =>
12     from = from'
13     to = to'
14     amt = amt'
15     from.prep_withdraw(amt, this)
16     to.prep_deposit(this)
17
18   be confirm_withdraw() =>
19     if aborted then
20       from.abort_transaction()
21       return
22     end
23     from_ok = true
24     if to_ok then
25       from.commit_withdraw(amt)
26       to.commit_deposit(amt)
27     end
28
29   // deposit logic analogous
30
31   be abort() =>
32     aborted = true
33     if to_ok then
34       to.abort_transaction()
35     elseif from_ok then
36       from.abort_transaction()
37     end
38
39   ...
40
41 fun transfer(s: Account, r: Account, amt: U64) =>
42   Coordinator(s, r, amt)
```

```
43 actor Account
44   var balance: U64
45   var frozen: Bool = false
46   var in_transaction: Bool = false
47
48   new open(balance': U64) =>
49     balance = balance'
50
51   be abort_transaction() =>
52     in_transaction = false
53
54   be prep_withdraw(amt: U64,
55                    c: Coordinator) =>
56     if (not in_transaction) and
57        (not frozen) and
58        (balance >= amt) then
59       in_transaction = true
60       c.confirm_withdraw()
61     else
62       c.abort()
63     end
64
65   be commit_withdraw(amt: U64) =>
66     balance = (balance - amt)
67     in_transaction = false
68
69   be prep_deposit(c: Coordinator) =>
70     if (not in_transaction) and
71        (not frozen) then
72       in_transaction = true
73       c.confirm_deposit()
74     else
75       c.abort()
76     end
77
78   be commit_deposit(amt: U64) =>
79     balance = (balance + amt)
80     in_transaction = false
```

Listing 2.2: Two-phase commit for actors. Logic related to the problem at hand (rather than synchronisation) highlighted in blue.

1. $c$ receives a request to transfer $a$ funds from $s$ to $r$ (constructor on Line 9).
2. $c$ tells $s$ and $r$ to reserve/accept $a$ funds (Line 15). If they confirm, $s$ and $r$ will ignore further messages (Lines 59 and 72) until released by $c$.
3. $s$ and $r$ either confirms the transfer to $c$ (Lines 60 and 73) or reports back that they cannot fulfill the request.
4. If both $s$ and $r$ confirms, $c$ sends a commit message to $s$ and $r$ causing them to perform their operations (Line 25). If one confirms and the other aborts, $c$ sends the confirming actor a message to abort (Lines 20 and 31).
5. After receiving confirm messages $s$ and $r$ can both continue to process messages (Lines 67 and 80).

Compare this with the bank-as-actor implementation:

1. The bank $c$ receives a request to transfer $a$ funds from $s$ to $r$. The bank verifies that both accounts exist and can withdraw and deposit $a$ funds and performs the corresponding withdrawal and deposit actions.

The coupling of the unit of isolation and concurrency brings a complexity to both the programmer and the program when building these atomic operations.

In conclusion: when problem decomposition perfectly fits the actor model, actor solutions are simple and elegant. However, when the fit is imperfect, the coupling of unit of concurrency to unit of isolation that permits sequential reasoning causes problems: fine-grained actors require bespoke complex protocol implementations to reason about state spread across several actors; coarse-grained actors unnecessarily serialise unrelated computation.

## 3 Behaviour-Oriented Concurrency

Behaviour-oriented concurrency has the same goal as actors: to simplify the complexities of concurrent programming. However, instead of abstracting concurrency to sending messages to actors, BoC abstracts concurrency into spawning tasks that operate over explicitly declared state. BoC, like actors, brings structure to concurrency.

To achieve this goal, BoC reimagines the actor paradigm. We take a step back from actors and coupling the units of isolation and concurrency into an actor, and intentionally decouple them into two distinct concepts. The unit of isolation is the concurrent owner, or *cown* (pronounced 'cone'), a single entry point into an isolated fragment of the program state. The unit of concurrency is the *behaviour*, an asynchronous unit of work that explicitly lists its required cowns. Behaviours execute asynchronously with exclusive access to their required cowns [9].

In this section we will use the C++ BoC runtime API to discuss the concepts of BoC [25]. A cown restricts access to an isolated part of shared state. In Listing 3.1, we define an `Account` struct, Lines 1 to 4, and construct an `Account` protected by a newly created cown, Line 8. The account is inaccessible through the cown until the cown is acquired by a behaviour. This means that the attempted access to the balance on Line 11 is a program error.

```
1  struct Account {
2    uint64_t balance = 0;
3    bool frozen = false;
4  };
5
6  int main() {
7    cown<Account> acc =
8      make_cown<Account>();
9
10   // invalid direct access
11   acc.balance += 10;
12 }
```

Listing 3.1: Creating cowns to protect data

The only way to gain access to the contents of the account, and indeed any cown, is to *spawn* a behaviour that requires it. Spawning a behaviour is achieved using the **when** construct which takes the cowns to acquire, between regular braces `(...)`, and a lambda function

to apply when the cowns are available. The behaviour will run in the future, asynchronously, when the cowns are not in use by any other behaviour. Behaviours cannot acquire more cowns or release access to cowns throughout their execution and cowns are implicitly released at the end of a behaviour's execution.

In Listing 3.2 we define a function `transfer` which takes two `Accounts` wrapped in `cowns` and an amount to transfer between the accounts. We first spawn a behaviour, Lines 4 to 6, which requires the source account `s`. `[=]` instructs C++ to capture `amount` by value. The arguments of the lambda must match up with the arguments of the **`when`**, *i.e.,* each `cown<T>` must be matched with an `acquired<T>`. When the cown is available the behaviour may run and will reduce the balance of the account. Similarly, we spawn a second behaviour, Lines 8 to 10, which increases the balance of the destination account `r`. Note that behaviours run asynchronously: the `transfer` function will complete directly, regardless of when the behaviours run.

```
1  void transfer(cown<Account> s,
2    cown<Account> r, uint64_t amount) {
3
4    when(s)<<[=](acquired<Account> s){
5      s.balance -= amount
6    };
7
8    when(r)<<[=](acquired<Account> r){
9      r.balance += amount
10   };
11 }
```

Listing 3.2: Spawning behaviours

A defining feature of BoC, with respect to actors, is that behaviours can require and use multiple cowns. Such behaviours will only run when none of the required cowns are in use by other behaviours. When such a behaviour does run, it will have access to the contents of *all* cowns, allowing us to construct more complicated behaviours that depend on and affect the state of multiple cowns. Listing 3.3 demonstrates this key feature of BoC in the context of the bank transfer. We define a behaviour that requires both cowns `s` and `r`, the behaviour checks if both accounts are able to perform the transaction and if so completes it. This transfer is an atomic operation by construction of the BoC paradigm.

BoC allows us to create ad-hoc atomic operations that operate over the state of multiple cowns. The type of the data which the cown protects does not have to provide an interface which guarantees a certain degree of atomicity (as per behaviours in actors). Moreover, if one of the cowns is currently in use

```
1  void transfer(cown<Account> s, cown<Account> r, uint64_t amount) {
2    when(s, r)<<[=](acquired<Account> s, acquired<Account> r) {
3      if (s.balance >= amount && !s.frozen && !r.frozen) {
4        s.balance -= amount
5        r.balance += amount
6      }
7    };
8  }
```

Listing 3.3: Behaviours with multiple cowns

elsewhere the running of the behaviour is simply delayed without having to resort to protocols like two-phase commit with retries.

### 3.1 Deadlock freedom and Causality

It is important to note that spawning is a synchronous action performed by `transfer`, but starting and running the behaviours are asynchronous actions performed by the runtime. Thus, assuming the two accounts do not alias, the increase and decrease behaviours in Listing 3.2 can execute in parallel, or in either order. To further stress the importance of this point, in Listing 3.4 the nested behaviour which increases the balance of `r`, Lines 2 to 4, does *not* have access to the account `s`. The outer behaviour spawns the inner behaviour synchronously and continues executing the rest of its behaviour. This means that the operation which reduces the balance of `s`, Line 5, and the operation which increases the balance of `r`, Line 3, may be executing in parallel. Moreover, by construction, BoC is *deadlock-free*. In Listing 3.4 we construct two pairs of behaviours, where the cowns required for the inner and outer behaviours are swapped. If **when** is (erroneously) viewed as a synchronous lock-guard, then one could reasonably expect a potential deadlock, however a deadlock is not possible here.

```
1  when(s)<<[=](acquired<Account> s){
2    when(r)<<[=](acquired<Account> r){
3      r.balance += amount
4    };
5    s.balance -= amount
6  }
7
8  when(r)<<[=](acquired<Account> r){
9    when(s)<<[=](acquired<Account> s){
10     s.balance += amount
11   };
12   r.balance -= amount
13 };
```

Listing 3.4: Deadlock-free behaviours

Finally, BoC guarantees causal ordering between behaviours, which we refer to as a *happens before* order. A behaviour $b$ happens before another behaviour $b'$ iff $b$ and $b'$ require overlapping sets of cowns, and $b$ is spawned before $b'$ in program order. In Listing 3.5, `b1` and `b2` do not require the same cowns and thus can execute in parallel, whereas both `b1` and `b2` overlap with the cowns required by `b3` and thus `b3` will only execute after both `b1` and `b2`. By similar reasoning, `b4` can only execute after `b3` (and thus also after `b1`). The causal ordering offered by BoC lets programmers reason about orderings also of nested behaviours. There are interesting design patterns that have been explored in other literature on BoC [8, 9]. We do not explore causal ordering further in this paper.

```
1  when(s)<<[=](acquired<Account> s){ /* b1 */ };
2  when(r)<<[=](acquired<Account> r){ /* b2 */ };
3  when(s, r)<<[=](acquired<Account> s, acquired<Account> r){/* b3 */};
4  when(s)<<[=](acquired<Account> s){ /* b4 */ };
```

Listing 3.5: Partially-ordered behaviours

### 3.2 Behaviour-Oriented Concurrency is Concurrent Procedural Programming

Just as actor programs can be viewed as marrying objects with concurrency in object-oriented programming, BoC can be viewed as marrying procedures with concurrency. A BoC program is constructed as a collection of behaviours which define the cowns they require and the operations that they will perform on these cowns; the programmer only needs to define these operations and does not have to consider the complexities of synchronisation, deadlocks and so on.

Finally, BoC can also be used to simulate actor programs. Actor programs can be translated into BoC programs by using one cown per actor and creating behaviours which operate over this isolated actor state. The program sketched in Listings 3.1 and 3.2 behaves exactly like the actor program in Listing 2.1. In the next two sections we show that this simulation relation also holds formally.

## 4 From Actors to Behaviour-Oriented Concurrency

In this section we explain behaviour-oriented concurrency formally in terms of actors. We do so by starting from a high-level actor semantics, gradually changing it until we have a formal semantics of BoC that is equivalent to previous work [9]. In Section 5 we prove that for each change the updated semantics simulates the previous semantics, with the corollary that BoC can simulate the actor model.

The changes made to the actor semantics do not reflect how BoC was actually developed, but each change represents an important insight into its design. Our goal is to highlight the similarities and differences between the actor model and BoC and show how BoC generalises the actor model.

The rest of this section is structured as follows. Section 4.1 presents the semantics of traditional actors. Section 4.2 moves all messages into a single global data structure, making scheduling a centralised concern instead of a concern spread across all actors. Section 4.3 makes actors ephemeral, releasing the resources of an actor whenever it is idle. Section 4.4 introduces the ability for messages to have multiple receivers, producing a semantics that is equivalent to BoC. Finally, Section 4.5 discusses how to formalise isolated actor states.

### 4.1 Traditional Actors, Formally

The focus in our formal development is on actors and BoC as *concurrency models* that go on top of some otherwise sequential language. We are not interested in the details of the underlying language, so we keep it abstract. The underlying language is the same for all the actor semantics and will only change slightly for BoC (cf. Section 4.4).

We use $E$ as the state of a sequential process, which could be as simple as an expression or more complex such as a stack with local variables and other state. We use $h$ as a global heap that the underlying language operates over. We assume the existence of a relation $\iota \vdash E, h \hookrightarrow E', h'$ representing a small-step evaluation of some sequential process $E$ (to $E'$), operating on the heap $h$ (resulting in $h'$).

The relation also includes the identifier $\iota$ of the running actor, for example to allow the semantics of the underlying language to restrict which data in the heap $h$ can be accessed by the process, as exemplified in Section 4.5.

In order to interact with the concurrency model, the underlying language can also take two kinds of effect-producing steps. The relation $\iota \vdash E, h \hookrightarrow_{\text{create } \iota'} E', h'$ represents the effect of creating an actor with identifier $\iota'$. The step $\iota \vdash E, h \hookrightarrow_{\text{send } \iota' \ E''} E', h'$ represents sending a message containing $E''$ to an actor with identifier $\iota'$.

Figure 2 shows the syntax and semantics of the actor model. An actor $a$ consists of its identifier $\iota$, the state of its currently executing behaviour $e$, and its message queue $q$. The $e$ can be a sequential process $E$, or $\epsilon$ if the actor is idle. Each message in the message queue $q$ is represented by another $E$. A configuration of the actor model consists of its set of actors $A$ and the global heap $h$. Each rule selects a single actor from $A$, using $\uplus$ to mean that there is no other actor with the same identifier in $A$:

**Definition 1.** *Disjoint union by address.*

$$A \uplus (\iota, e, q) \triangleq \begin{cases} A \cup (\iota, e, q) & \textit{if } \iota \notin \{\iota' \mid (\iota', e', q') \in A\} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

There are five rules that can step an actor. In rule Step, an actor $\iota$ executes a single step without producing any effect (other than the updated heap). In rule End, an actor turns idle when its current behaviour is *finished*, a predicate that holds when no further sequential steps are possible. In rule Recv, an idle actor proceeds by processing the next message in its queue. In rule Create, an actor produces the create effect, which results in a new idle actor with an empty queue being added to the set of actors (note that the definition of $\uplus$ ensures that the identifier $\iota'$ is not in $A$). Finally, in rule Send, an actor produces the send effect, which results in the specified message being added to the queue of the receiver (the message is appended through a slight abuse of notation). The second premise of the rule is there to handle self sends: if $\iota = \iota'$ then $A = A'$ and the message is added to the message queue of $\iota$, otherwise $\iota$ is one of the actors in $A'$.

We argue that this semantics captures the essence of actors as a concurrency model, with just enough detail about the underlying language to allow interaction (through the create and send effects). The isolated state of an actor is represented by its identifier $\iota$, and this state is coupled with the actor as a unit of concurrency: an actor with identifier $\iota$ will only ever take steps $\iota \vdash E, h \hookrightarrow E', h'$, and sending a message to this actor is the only way to make such a step happen. Some features of actor systems, such as selective receives and messages being delivered out-of-order, are not captured explicitly by our semantics but can be modelled in the underlying language by adding information in the local state $E$ or global state $h$.

### 4.2 Processes with a Shared Message Queue

Our main motivation for BoC is decoupling the units of isolation and concurrency. In order to synchronise multiple concurrent entities, we need to have a centralised

$$\begin{array}{ll} a ::= (\iota, e, q) & q ::= E :: q \mid \epsilon \\ e ::= E \mid \epsilon & A \in \mathcal{P}(a) \end{array}$$

$$\frac{\iota \vdash E, h \hookrightarrow E', h'}{A \uplus (\iota, E, q), h \rightsquigarrow_{\mathrm{a}} A \uplus (\iota, E', q), h'} \textsc{Step} \qquad \frac{\mathit{finished}(E)}{A \uplus (\iota, E, q), h \rightsquigarrow_{\mathrm{a}} A \uplus (\iota, \epsilon, q), h} \textsc{End}$$

$$\frac{}{A \uplus (\iota, \epsilon, E :: q), h \rightsquigarrow_{\mathrm{a}} A \uplus (\iota, E, q), h} \textsc{Recv}$$

$$\frac{\iota \vdash E, h \hookrightarrow_{\mathrm{create}\ \iota'} E', h'}{A \uplus (\iota, E, q), h \rightsquigarrow_{\mathrm{a}} A \uplus (\iota, E', q) \uplus (\iota', \epsilon, \epsilon), h'} \textsc{Create}$$

$$\frac{\iota \vdash E, h \hookrightarrow_{\mathrm{send}\ \iota'\ E''} E', h' \qquad A \uplus (\iota, E', q) = A' \uplus (\iota', e, q')}{A \uplus (\iota, E, q), h \rightsquigarrow_{\mathrm{a}} A' \uplus (\iota', e, q' :: E''), h'} \textsc{Send}$$

Figure 2: Syntax and high-level semantics of traditional actors

view of all pending work in the system. This is hard when each actor has their own message queue. The first change we make to our actor semantics is therefore to merge all queues into a single global one. Since we will be deviating from the previous definition of an actor, we will be using the term *process* in place of actor going forward. Note that we are using processes to implement an actor system in Sections 4.2 and 4.3.

Figure 3 shows the syntax and semantics after this modification. Processes, written as $A^M$ to distinguish them from actors in the previous semantics, now only contain an identifier and a currently running expression. The configuration is extended to contain a list $M$ of messages in flight, each message containing the identifier of the receiving process and a sequential process $E$ as before.

The rules Step, End, and Create are the same as before as they do not concern sending or receiving messages. In rule Recv an idle process starts processing *its own* next message in the global queue (we slightly abuse notation to concatenate $M_1$ and $M_2$). In rule Send a process produces the send effect, which results in a message addressed to specified process being appended to the global message queue.

### 4.3 Ephemeral Processes

Looking at the processes in Figure 3, there is no persistent state in a process after it has finished its running behaviour[4]. This is in line with our end goal of decoupling the units of isolation and concurrency: once a behaviour over some isolated state is done we want to allow other behaviours to run over that same

[4] A process may have persistent state in $h$, but this is not stored in the process itself.

$$
\begin{array}{llll}
a^M & ::= (\iota, e) & M & ::= (\iota, E) :: M \mid \epsilon \\
e & ::= E \mid \epsilon & A^M & \in \mathcal{P}(a^M)
\end{array}
$$

$$
\frac{\iota \vdash E, h \hookrightarrow E', h'}{A \uplus (\iota, E), M, h \rightsquigarrow_{\mathrm{m}} A \uplus (\iota, E'), M, h'} \textsc{Step} \qquad \frac{\mathit{finished}(E)}{A \uplus (\iota, E), M, h \rightsquigarrow_{\mathrm{m}} A \uplus (\iota, \epsilon), M, h} \textsc{End}
$$

$$
\frac{(\iota, \_) \notin M_1}{A \uplus (\iota, \epsilon), M_1 :: (\iota, E) :: M_2, h \rightsquigarrow_{\mathrm{m}} A \uplus (\iota, E), M_1 :: M_2, h} \textsc{Recv}
$$

$$
\frac{\iota \vdash E, h \hookrightarrow_{\mathrm{create}\ \iota'} E', h'}{A \uplus (\iota, E), M, h \rightsquigarrow_{\mathrm{m}} A \uplus (\iota, E') \uplus (\iota', \epsilon), M, h'} \textsc{Create}
$$

$$
\frac{\iota \vdash E, h \hookrightarrow_{\mathsf{send}\ \iota'\ E''} E', h'}{A \uplus (\iota, E), M, h \rightsquigarrow_{\mathrm{m}} A \uplus (\iota, E'), M :: (\iota', E''), h'} \textsc{Send}
$$

Figure 3: The syntax and semantics of actors with a centralised message queue. The actor set $A^M$ is written as $A$ to reduce clutter.

and possibly other pieces of isolated state. Thus we want to "release" isolated state when it is not being accessed. We model this in our semantics by making processes ephemeral. This means that processes are removed as soon as they finish a behaviour and are started up again whenever a new message needs processing.

Figure 4 shows the semantics of ephemeral processes (the syntax is the same as in the previous section). Rules Step and Send are identical to the previous semantics. In rule End a process is removed when its behaviour is *finished* (instead of making it idle as previously). In rule Start, which corresponds the previous rule Recv, a message is selected with a recipient that is not running in $A$ and that does not have another message earlier in the queue. A process is started up to run that message. Finally, we ponder what it means to create a process when processes are ephemeral. Since we start processes when needed and remove them as soon as they are done, creating a process is no longer meaningful. In rule Create, the create effect is simply ignored.

### 4.4 Behaviour-Oriented Concurrency, Formally

We are now ready to define the semantics of behaviour-oriented concurrency. The main difference to the previous semantics is that messages can now have multiple receivers. This means that we need to start by changing our underlying language so that we can spawn behaviours with more than one recipient. While we are at it, we will also remove the create effect since the last semantics made it a no-op.

We use $E^b$ as the state of a sequential process of our new underlying language and assume that it operates over some global heap $h^b$. We assume the existence

$$\frac{\iota \vdash E, h \hookrightarrow E', h'}{A \uplus (\iota, E), M, h \rightsquigarrow_{\mathrm{e}} A \uplus (\iota, E'), M, h'}\ \textsc{Step} \qquad \frac{\mathit{finished}(E)}{A \uplus (\iota, E), M, h \rightsquigarrow_{\mathrm{e}} A, M, h}\ \textsc{End}$$

$$\frac{(\iota, \_) \notin A \cup M_1}{A, M_1 :: (\iota, E) :: M_2, h \rightsquigarrow_{\mathrm{e}} A \uplus (\iota, E), M_1 :: M_2, h}\ \textsc{Start}$$

$$\frac{\iota \vdash E, h \hookrightarrow_{\mathsf{create}\ \iota'} E', h'}{A \uplus (\iota, E), M, h \rightsquigarrow_{\mathrm{e}} A \uplus (\iota, E'), M, h'}\ \textsc{Create}$$

$$\frac{\iota \vdash E, h \hookrightarrow_{\mathsf{send}\ \iota'\ E''} E', h'}{A \uplus (\iota, E), M, h \rightsquigarrow_{\mathrm{e}} A \uplus (\iota, E'), M :: (\iota', E''), h'}\ \textsc{Send}$$

Figure 4: Semantics of ephemeral processes

of two stepping relations, one that touches the heap $\bar{\iota} \vdash E^b, h^b \hookrightarrow E_2^b, h_2^b$ (we use $\bar{\iota}$ for a set of zero or more identifiers) and one that produces an effect spawning a behaviour with multiple receivers $\bar{\iota} \vdash E^b, h^b \hookrightarrow_{\mathrm{spawn}\ \bar{\iota}'\ E_3^b} E_2^b, h_2^b$. We use the same kind of identifiers $\iota$ as for the actor semantics to highlight their connection, even though there is no longer a concept of an actor. Here we think of $\iota$ as the identifier of some isolated resource.

Figure 5 shows the syntax and semantics of behaviour-oriented concurrency. Behaviours $b$ consist of a set of resource identifiers $\bar{\iota}$ and a currently running sequential process $E^b$. There is a global list of pending behaviours $P$ as well as a set of currently running behaviours $R$. We use a similar version of the disjoint union from before, but extended to handle multiple identifiers:

**Definition 2.** *Disjoint union by held resources.*

$$R \uplus (\bar{\iota}, E) \triangleq \begin{cases} R \cup (\bar{\iota}, E) & \textit{if } \bar{\iota} \cap \{\iota \mid \iota \in \bar{\iota} \ \wedge\ (\bar{\iota}, E) \in R\} = \emptyset \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

The rules are very similar to the ephemeral processes except that each running behaviour is associated with multiple identifiers $\bar{\iota}$. In the rule START a pending behaviour starts running, assuming none of its required resources are either currently running (in $R$) or have been scheduled earlier (in $P_1$). Note that this is analogous to a message receive in the ephemeral process model, but with multiple receivers. In the rule SPAWN a new behaviour is spawned and added to the list of pending behaviours. Note again that this is analogous to a message send in the ephemeral process model, but with multiple receivers.

Because each behaviour can now synchronise over any number of resources, this concurrency model can express concurrent dependencies that are not easily expressible in the actor model. However, if we limit ourselves to always requiring

$$
\begin{array}{rcl}
b & ::= & (\bar{\iota}, E^b) \\
P & ::= & b :: P \mid \epsilon \\
R & \in & \mathcal{P}(b)
\end{array}
$$

$$\frac{\bar{\iota} \vdash E, h \hookrightarrow E', h'}{R \uplus (\bar{\iota}, E), P, h \rightsquigarrow_{\mathrm{b}} R \uplus (\bar{\iota}, E'), P, h'} \textsc{Step} \qquad \frac{\mathit{finished}(E)}{R \uplus (\bar{\iota}, E), P, h \rightsquigarrow_{\mathrm{b}} R, P, h} \textsc{End}$$

$$\frac{\{\iota \mid (\iota, \_) \in R \cup P_1\} \cap \bar{\iota} = \emptyset}{R, P_1 :: (\bar{\iota}, E) :: P_2, h \rightsquigarrow_{\mathrm{b}} R \uplus (\bar{\iota}, E), P_1 :: P_2, h} \textsc{Start}$$

$$\frac{\bar{\iota} \vdash E, h \hookrightarrow_{\text{spawn } \bar{\iota}', E''} E', h'}{R \uplus (\bar{\iota}, E), P, h \rightsquigarrow_{\mathrm{b}} R \uplus (\bar{\iota}, E'), P :: (\bar{\iota}', E''), h'} \textsc{Spawn}$$

Figure 5: Syntax and semantics of behaviour-oriented concurrency. We write $E$ and $h$ instead of $E^b$ and $h^b$ to avoid clutter.

exactly one resource we get a concurrency model that simulates the actor model: each actor is represented by a singleton resource and each message is represented by a behaviour on that resource. We formalise this intuition in Section 5.

### 4.5 Formalising Isolated State

In all our semantics, the underlying language is kept abstract. This means that the concept of isolation is also kept abstract. While many actor systems achieve data-race freedom through isolation, it is not required for reasoning about actors as a concurrency model. In this section we give an example of how the underlying language of the actor semantics in Figure 2 could be defined so that each actor operates on its own local heap. For simplicity, we still keep all other details about the underlying language abstract.

The actor semantics works on a global heap $h$. We instantiate this heap as $h \triangleq \overline{\iota \times H}$, a set of tuples containing an actor identifier $\iota$ and a *heaplet* $H$, which could for example map names to values. The intuition is that the tuple $(\iota, H)$ contains the isolated state of actor $\iota$ in the heaplet $H$. We assume the existence of a relation $E, H \hookrightarrow E', H'$ representing a small-step evaluation of a sequential process $E$ in heaplet $H$. We use this relation to instantiate the previously assumed small-step evaluation used in the Step rule (using $\uplus$ in a similar manner as before):

$$\frac{E, H \hookrightarrow E', H'}{\iota \vdash E, h \uplus (\iota, H) \hookrightarrow E', h \uplus (\iota, H')} \textsc{Eval}$$

This rule uses the $\iota$ to select the right heaplet $H$ from the global heap $h$ and evaluates only under that heaplet in a frame-rule-like fashion. The rule with the

create effect $\iota \vdash E, h \hookrightarrow_{\text{create } \iota'} E', h'$ would extend $h$ with a new empty heaplet associated with the new actor $\iota'$, meaning an actor would always be created together with its heaplet. The rule for sending messages does not need to be aware of heaplets.

Again, we note that our actor semantics does not *require* actors operating on isolated state, but it supports it as exemplified here. Section 6.2 brings up some existing actor systems that enforce isolated state.

## 5 Meta-Theory: Simulation Theorems

In this section we prove a number of simulation theorems about the semantics in Section 4 with the final corollary that the semantics of behaviour-oriented concurrency in Figure 5 simulates the actor semantics in Figure 2.

### 5.1 Shared-Queue Processes

The actor semantics in Figure 3 is equivalent to the semantics of shared-queue processes in Figure 2. We could prove a bisimulation, but since our end goal is showing that BoC can simulate actors we will be stop at proving a simulation here as well. We define a similarity relation $\mathcal{R}_1$ which relates actors from $A$ with processes in $A^M$ and their global message queue $M$:

**Definition 3.** *Similarity of traditional actors and shared-queue processes.* A set of traditional actors $A$ is similar to a set of processes $A^M$ with a shared message queue $M$, written $\mathcal{R}_1(A, A^M, M)$, iff:

1. $A$ has an actor $(\iota, e, q)$ for some $q$ iff $A^M$ has a process $(\iota, e)$,
2. for each message $(\iota, E)$ in $M$, there is some process $(\iota, e)$ in $A^M$,
3. for each actor $(\iota, e, q)$, filtering only the messages for $\iota$ from $M$ yields $q$.

The first property states that dropping the local queues from the actors in $A$ result in the same queue-less processes as in $A^M$. The second property states that every message in $M$ has a valid address (the identifier of an existing process). The third property states that the local queue of each actor $\iota$ can be recreated by filtering only the messages addressed to $\iota$ in $M$.

We show that each step with the traditional actors corresponds to a step from a similar configuration in the semantics with the centralised queue and that the resulting configurations are also similar:

**Theorem 1 (Processes with a shared queue simulate traditional actors).** *If* $\mathcal{R}_1(A, A^M, M)$ *and* $A, h \rightsquigarrow_{\text{a}} A_2, h_2$*, then there exists* $A_2^M$ *and* $M_2$ *such that* $A^M, M, h \rightsquigarrow_{\text{m}} A_2^M, M_2, h_2$ *and* $\mathcal{R}_1(A_2, A_2^M, M_2)$*.*

The proof is by cases on the rule used to step $A$. The cases for Step and End are trivial after using the first property of $\mathcal{R}_1$ to find the corresponding process in $A^M$. In the case for Create, in order to satisfy the third property of $\mathcal{R}_1$, we make use of the second property of $\mathcal{R}_1$ to ensure that there are no spurious

messages for the newly created process already in $M$. In the case for RECV we use third property of $\mathcal{R}_1$ to get that the first message in the local message queue is also the first message addressed to that process in $M$. Removing that message from both queues preserves similarity. Finally, in the case for SEND we note that similarity is preserved when adding a message to the end of the global queue $M$ and adding a message to the corresponding actor's local queue in $A$.

### 5.2 Ephemeral Processes

Because process creation is ignored, the ephemeral processes in Figure 4 are not equivalent to the persistent processes in Figure 3: creating the same process twice is not allowed with persistent processes. We can however prove that ephemeral processes simulate persistent processes by setting up a similarity relation $\mathcal{R}_2$ which relates processes in $A^M$ with ephemeral processes (written as $A^e$):

**Definition 4.** *Similarity of shared-queue processes and ephemeral processes.*
A set of processes $A^M$ with a shared queue is similar to a set of ephemeral processes $A^e$, written $\mathcal{R}_2(A^M, A^e)$, iff:

1. for all processes $(\iota, E)$ in $A^M$ we have $(\iota, E)$ in $A^e$,
2. for all idle processes $(\iota, \epsilon)$ in $A^M$ we have no process with identifier $\iota$ in $A^e$.

The first property ensures that each running persistent process corresponds to a running ephemeral process. The second property ensures that any idle persistent process does not have an ephemeral process running another behaviour.

We show that when starting from similar configurations, each step in $A^M$ has a corresponding step in $A^e$ that preserves similarity.

**Theorem 2 (Ephemeral processes simulate processes with a shared queue).**
*If* $\mathcal{R}_2(A^M, A^e)$ *and* $A^M, M, h \rightsquigarrow_{\mathrm{m}} A_2^M, M_2, h_2$*, then there exists* $A_2^e$ *such that* $A^e, M, h \rightsquigarrow_{\mathrm{e}} A_2^e, M_2, h_2$ *and* $\mathcal{R}_2(A_2^M, A_2^e)$.

The proof is by cases on which rule was used to step $A^M$. The cases for STEP and SEND are trivial. In the case for END the process turns idle which preserves similarity when its ephemeral counterpart is removed. Similarly, in the case for RECV we know that there is no ephemeral process in $A^e$ corresponding to the idle process in $A^M$, so the premise of the START rule is satisfied and the use of $\uplus$ is well defined. In the case for CREATE, an idle process is created, which preserves similarity with the unextended $A^e$ due to the second property of $\mathcal{R}_2$.

### 5.3 Behaviour-Oriented Concurrency

The similarity relation between ephemeral processes (Figure 4) and behaviour-oriented concurrency (Figure 5) is slightly more involved since we have different underlying languages. We start by assuming a similarity relation between the underlying languages. This can be thought of as the properties of an imagined compilation of $E$ processes into $E^b$ processes.

**Parameter 1** *Similarity of underlying languages.* We assume the existence of a similarity relation between the two underlying languages, written $\mathcal{R}_{ul}(E, E^b)$, as well as a similarity relation between their respective heaps, written $\mathcal{R}_h(h, h^b)$. The similarity relations are assumed to have the following properties:

1. If $\mathcal{R}_{ul}(E, E^b)$, $\mathcal{R}_h(h, h^b)$ and $\iota \vdash E, h \hookrightarrow E', h'$ for some $\iota$, then there exists $E_2^b$ and $h_2^b$ such that $\{\iota\} \vdash E^b, h^b \hookrightarrow E_2^b, h_2^b$, $\mathcal{R}_{ul}(E', E_2^b)$ and $\mathcal{R}_h(h', h_2^b)$,
2. If $\mathcal{R}_{ul}(E, E^b)$, $\mathcal{R}_h(h, h^b)$ and $\iota \vdash E, h \hookrightarrow_{\text{create } \iota'} E', h'$ for some $\iota'$, then there exists $E_2^b$ and $h_2^b$ such that $\{\iota\} \vdash E^b, h^b \hookrightarrow E_2^b, h_2^b, \mathcal{R}_{ul}(E', E_2^b)$ and $\mathcal{R}_h(h', h_2^b)$,
3. If $\mathcal{R}_{ul}(E, E^b)$, $\mathcal{R}_h(h, h^b)$ and $\iota \vdash E, h \hookrightarrow_{\text{send } \iota', E''} E', h'$ for some $\iota'$ and $E''$, then there exists $E_2^b$, $E_3^b$ and $h_2^b$ such that $\{\iota\} \vdash E^b, h^b \hookrightarrow_{\text{spawn } \{\iota'\}, E_3^b} E_2^b, h_2^b$, $\mathcal{R}_{ul}(E', E_2^b)$, $\mathcal{R}_h(h', h_2^b)$ and $\mathcal{R}_{ul}(E'', E_3^b)$,
4. If $\mathcal{R}_{ul}(E, E^b)$ and *finished*$(E)$, then *finished*$(E^b)$.

The first property states that evaluation in the context of an process $\iota$ corresponds to evaluation in the context of a singleton resource $\{\iota\}$. The second property states that producing a create effect corresponds to some non-effectful computation. The third property states that producing a send effect corresponds to producing a spawn effect with a singleton resource and a similar behaviour. The fourth property states that a sequential process is *finished* when its similar counterpart is.

The simulation relation between BoC and ephemeral processes can now be formulated as follows.

**Definition 5.** *Similarity of ephemeral processes and behaviour-oriented concurrency.* A set of ephemeral processes $A^e$ with a shared queue $M$ is similar to a set of running behaviours $R$ and pending behaviours $P$, written $\mathcal{R}_3(A^e, M, R, P)$, iff:

1. for all processes $(\iota, E)$ in $A^e$ we have $(\{\iota\}, E^b)$ in $R$, such that $\mathcal{R}_{ul}(E, E^b)$,
2. the number of processes in $A^e$ is the same as the number of running behaviours in $R$,
3. for each message $(\iota, E)$ in $M$ there is a pending behaviour $(\{\iota\}, E^b)$ at the corresponding index of $P$ such that $\mathcal{R}_{ul}(E, E^b)$,
4. the number of pending messages in $M$ is the same as the number of pending behaviours in $P$.

The first two properties state that there is a one-to-one correspondence between running processes and running behaviours. The second two properties state that there is a one-to-one correspondence between pending messages and pending behaviours. With this similarity relation we can now show a simulation between BoC and ephemeral processes.

**Theorem 3 (Behaviour-oriented concurrency simulates ephemeral processes).** *If* $\mathcal{R}_3(A^e, M, R, P)$, $\mathcal{R}_h(h, h^b)$ *and* $A^e, M, h \rightsquigarrow_{\text{e}} A_2^e, M_2, h_2$, *then there exists* $R_2$, $P_2$ *and* $h_2^b$ *such that* $R, P, h^b \rightsquigarrow_{\text{b}} R_2, P_2, h_2^b$ *and* $\mathcal{R}_3(A_2^e, M_2, R_2, P_2)$ *and* $\mathcal{R}_h(h_2, h_2^b)$.

The proof is by cases on which rule was used to step $A^e$. In each case we use the first two properties of $\mathcal{R}_3$ to ensure that each process has a corresponding behaviour. By the first property of Parameter 1 this behaviour can take a

similarity-preserving step. In the case for STEP this is enough to show preservation of similarity. In the case for CREATE we additionally use the second property of Parameter 1 to get that we can take a corresponding step. In the case for END we use the fourth property of Parameter 1 to get that the corresponding behaviour is also finished.

In the case for START we use the second two properties of $\mathcal{R}_3$ to get that there is a corresponding pending behaviour. Because all behaviours have a singleton resource the premise is transferable to the set-disjointness formulation in BoC. Finally, in the case for SEND we use the third property of Parameter 1 to get that the send effect corresponds to a spawn effect of a similar behaviour with a singleton resource.

We can now formulate our final corollary which states that BoC simulates actors when restricted to singleton resource sets.

**Theorem 4 (Behaviour-oriented concurrency simulates traditional actors).** *If* $\mathcal{R}_{ul}(E, E^b)$, $\mathcal{R}_h(h, h^b)$ *and* $\{(\iota, E, \epsilon)\}, h \rightsquigarrow_{\mathrm{a}}^* A', h'$, *then there exists* $R'$, $P'$ *and* $h_2^b$ *such that* $\{(\{\iota\}, E^b)\}, \emptyset, h^b \rightsquigarrow_{\mathrm{b}}^* R', P', h_2^b$ *and* $\mathcal{R}_h(h', h_2^b)$.

Proof is by induction over the $\rightsquigarrow_{\mathrm{a}}^*$ relation (the reflexive and transitive closure of $\rightsquigarrow_{\mathrm{a}}$). With a single actor and an empty message queue it is trivial to show that we can use $\mathcal{R}_3$, $\mathcal{R}_2$ and $\mathcal{R}_1$ to connect the two semantics. Through the simulation theorems we have shown that similarity is preserved by each step, so each step in the traditional actor semantics has a corresponding step in BoC.

## 6 Discussion

Comparing the actor semantics in Figure 2 and the semantics of behaviour-oriented concurrency in Figure 5, the flexibility that comes with decoupling the units of isolation and concurrency is visible in how each behaviour in BoC can run with an arbitrary number of resources $\bar{\iota}$ in rule STEP instead of just the one. Any set of running behaviours can be thought of as a set of temporary actors whose local state consists of the union of the held resources. When a behaviour finishes in rule END, these resources are freed up to be used by in other configurations by other behaviours. This reconfigurability is a key difference between actors and BoC.

From a programming point of view, even though we have shown that behaviour-oriented concurrency can simulate actors, there is a philosophical difference in how an actor is a persistent entity with a stable identity. In behaviour-oriented concurrency, a behaviour is anonymous and has no memory after it finishes, except for what may be encoded into the resources it handles. This difference is analogous to the difference between the strong encapsulation of objects in object-oriented programming and the handling of dynamic data in procedural programming.

Going back to the bank transfer example from Section 2 we can reinforce the differences between the actor and BoC models by comparing their executions. We do so using two diagrams in Figure 6: Figure 6a for actors and Figure 6b for BoC. These diagrams focus on the order of events rather than mirroring the syntactic

representation in the semantics. However, the syntactic representations allow the orders presented in the diagrams.

Figure 6a demonstrates the execution of three actors, the sender $s$, receiver $r$, and coordinator $c$. The actors coordinate the transfer of money from $s$ to $r$ using two-phase commit. As per Figure 2, each of the messages of an actor are totally ordered (with respect to each other) based on the order in which they were received. Once received, a message can be processed at some arbitrary time in the future, but not before earlier received messages, [5] nor whilst an actor is processing another message. Message sends are instantaneous. Note that there are several different possible executions but two-phase commit ensures the outcome is always the same.

Figure 6b demonstrates the execution of a BoC program with three cowns: the sender $s$, receiver $r$, and coordinator $c$. The program transfers money from $s$ to $r$, with $c$ initiating the transfer. As per Figure 5, all behaviours are totally ordered by a spawn order (but partially ordered when we consider the behaviours' required resources). We represent a behaviour having access to multiple resources by presenting the times in which a cown is available or in use by a running behaviour (which implicitly acquires and releases the cowns at the start and end of its execution).

The two figures demonstrate that the actor behaviour executes using only the actors' states whilst BoC behaviours can access multiple states. Actor messages have a partial order whilst BoC programs have a total spawn order, and – in this example – that the actor program requires more messages to achieve the same outcome as the BoC program.

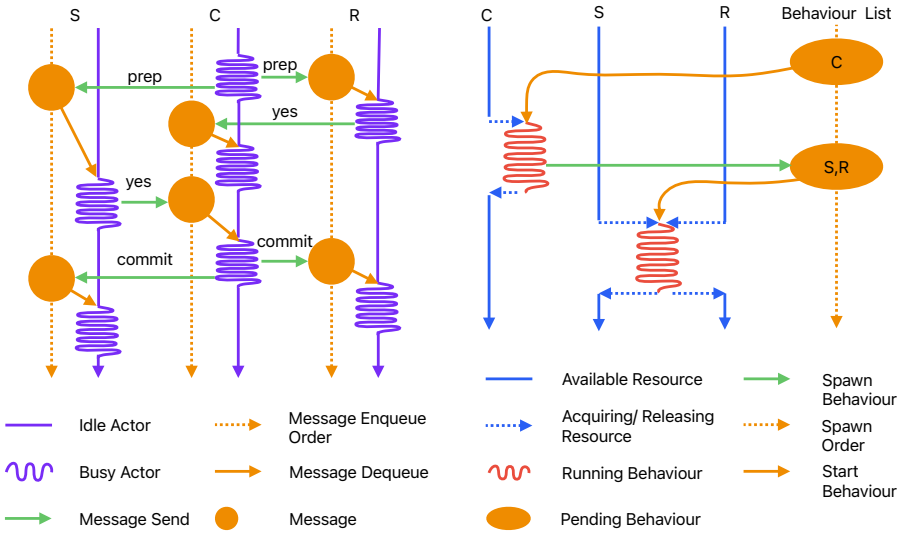
### 6.1 Working Around the Actor Model

Many actor systems have features that sidestep the pure actor model. Erlang ensures isolation by copying all data passed in messages but also features the Erlang Term Storage (ETS), a mutable map data type that can be shared between actors [3]. With the ETS, actors can communicate without using message passing. Operations on the ETS are synchronised, meaning that they are subject to race conditions but not data races. The same properties are inherited by languages building on the Erlang runtime, such as Elixir [12].

A survey on actor programs in Scala shows that programmers frequently mix the actor model with other concurrency constructs such as futures and threads, either due to limitations of the actor libraries used or limitations of the actor model itself [24]. There are extensions of actors in Scala to support transactional memory [15] and fork/join parallelism [17]. The Chocola language, an extension of Clojure, also allows mixing actors with futures and transactions [23]. The C++ Actor Framework (CAF) complements message passing with streams that can be used to send data between actors [5].

An approach to avoiding the performance problem of coarse-grained actors is permitting actors to encapsulate several threads of control. This either comprom-

[5] Recall that these semantics do not consider selective receives




(a) An Actor Bank Transfer (b) A BoC Bank Transfer

Figure 6: Comparing bank transfer executions

ises simplicity by relying on means other than isolation to ensure that behaviours do not interfere, or compromises correctness if no such means is added. This complexity may spill into the implementation. Examples of actor systems to go down this route include Joelle [18] which relied on an effect system to reason about what behaviours were permitted to execute in parallel in a single actor and Encore [7]. The latter introduced the notion of a "hot object" whose behaviours were implemented using lock-free algorithms [6] and turned message sends into synchronous method calls under the hood. The latter controversial design stemmed from Encore's runtime system being based on Pony's, which scheduled actors rather than individual messages (which would be a more reasonable design when actors are parallel). Encore notably also needed to invent a separate memory management scheme [26] to handle concurrent accesses inside an actor. These examples are anecdotal evidence that trying to address performance issues in actor systems by making actors parallel comes with significant complexity costs.

### 6.2 Isolation in Practice

Ensuring isolation (or similar concepts) has been the focus of a number of research and engineering efforts. Here we explore a few of these efforts to convince the reader that the required isolation we have claimed thus far is a realistic expectation.

Behaviour-oriented concurrency was co-designed with a static type system called Reggio which guarantees that the heap is a forest of regions, each region with a single external entry point [4]. This type system enforces strong guarantees

on the references that may exist in a program, and how data can be moved around the program. In the semantics in Figure 5, each $\iota$ could represent the unique entry point to the root of a region tree. A dynamic application of this type system is also being pursued in Python [20].

Erlang and Elixir ensure that there is no shared data by passing all data by value [3, 12]. Another approach is taken by Pony and Encore which both feature a shared heap, but whose type systems ensure that data passed by reference in messages is either immutable or is passed together with its ownership so that at most one actor at a time can access the data [7, 10, 22]. Other actor systems that support ownership transfer of data between actors include Joelle [18], Kilim [21] and LaCasa [14].

Rust has been introducing [11] many programmers to ownership and borrowing. In Rust, data must have at most one owner, providing a single unique entry point to that data, but mutable and immutable references can be borrowed by those that need it (with the guarantee that the reference cannot escape).

## 7 Conclusion

In this paper we have examined behaviour-oriented concurrency through the lens of the actor model. We have shown that the key difference between the two comes from the decoupling of the unit of isolation from the unit of concurrency. Using Actors, programmers have to decide on the granularity of isolation at the point of construction of an actor. In BoC on the other hand, this decision can be delayed until the point of construction of the behaviour. This frees the programmer from the need for implementing complicated synchronisation protocols.

The actor model can be thought of as a natural extension of object-oriented programming to a concurrent setting; an actor encapsulates its state just as an object does, but synchronises concurrent messages by running them with its own thread of control, thereby avoiding data races. In the same way, BoC can be thought of as an extension of procedural programming to a concurrent setting. A behaviour is a concurrent one-off procedure which temporarily encapsulates its required resources. In order to synchronise behaviours and avoid data races, the runtime tracks the dependencies of the required resources of each behaviour and schedules them based on these dependencies.

An important feature of the actor model is that it scales from single-node concurrent programs to distributed programs. There is currently no story for distribution in BoC, and the reconfigurability of isolated state and the need for a centralised view of the available work makes this non-trivial. It remains a venue of future work to investigate an adaptation of BoC that works for distributed computing.

# Bibliography


[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT press, 1986.

[2] Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, September 1990.

[3] Joe Armstrong. A history of Erlang. In *HOPL III*, pages 6–1–6–26. ACM, 2007.

[4] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. Reference capabilities for flexible memory management. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[5] The C++ actor framework. `https://www.actor-framework.org/`. Accessed February 2025.

[6] Elias Castegren and Tobias Wrigstad. Relaxed linear references for lock-free data structures. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 6:1–6:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[7] Elias Castegren and Tobias Wrigstad. Encore: Coda. In Frank S. de Boer, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Eduard Kamburjan, editors, *Active Object Languages: Current Research Trends*, volume 14360 of *Lecture Notes in Computer Science*, pages 59–91. Springer, 2024.

[8] Luke Cheeseman. *Behaviour-Oriented Concurrency.* PhD thesis, Imperial College London, Mar 2024.

[9] Luke Cheeseman, Matthew J. Parkinson, Sylvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. When concurrency matters: Behaviour-oriented concurrency. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[10] Sylvan Clebsch. *Pony: Co-designing a Type System and a Runtime.* PhD thesis, Imperial College London, 2017.

[11] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. A grounded conceptual model for ownership types in Rust. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[12] The Elixir programming language. `https://elixir-lang.org`. Accessed February 2025.

[13] Irene Greif. *Semantics of Communicating Parallel Processes.* PhD thesis, Massachusetts Institute of Technology, 1975.

[14] Philipp Haller and Alex Loiko. LaCasa: Lightweight affinity and object capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 272–291. ACM, 2016.

[15] Yaroslav Hayduk, Anita Sobe, Derin Harmanci, Patrick Marlier, and Pascal Felber. Speculative concurrent processing with transactional memory in

the actor model. In *Principles of Distributed Systems: 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings 17*, pages 160–175. Springer, 2013.

[16] Carl Hewitt, Peter Bishop, and Richard 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.

[17] Shams Imam and Vivek Sarkar. Habanero-Scala: Async-finish programming in Scala. In *The Third Scala Workshop (Scala Days 2012)*, 2012.

[18] Johan Östlund. *Language Constructs for Safe Parallel Programming on Multi-cores*. PhD thesis, Department of Information Technology, Uppsala University, Jan 2016.

[19] Pony tutorial. `https://tutorial.ponylang.io/`. Accessed June 2025.

[20] Explorations into a programming model for BoC in the Python runtime. `https://github.com/matajoh/pyrona`. Accessed February 2025.

[21] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP*, volume 8, pages 104–128. Springer, 2008.

[22] George Steed and Sophia Drossopoulou. A principled design of capabilities in pony. *https://www.ponylang.io/media/papers/a_prinicipled_design_of_capabilities_in_pony.pdf*, 2016.

[23] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. Chocola: Composable concurrency language. *ACM Trans. Program. Lang. Syst.*, 42(4), January 2021.

[24] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why do Scala developers mix the actor model with other concurrency models? In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*, pages 302–326. Springer, 2013.

[25] Research programming language for concurrent ownership. `https://github.com/microsoft/verona`. Accessed February 2025.

[26] Albert Mingkun Yang and Tobias Wrigstad. Type-assisted automatic garbage collection for lock-free data structures. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. ACM, 2017.