`when` Behaviours Have to Happen

An Axiomatic Model of Causality in Behaviour-Oriented Concurrency

Luke Cheeseman
Uppsala University, Sweden

Elias Castegren
Uppsala University, Sweden

Tobias Wrigstad
Uppsala University, Sweden

Sophia Drossopoulou
Imperial College London, UK

Matthew J. Parkinson
Azure Research, London, UK

Abstract

Behaviour-oriented concurrency (BoC) is a recently established programming model in which programmers define concurrent operations that execute atomically across multiple isolated resources. This allows for expressive interactions but introduces complex causal dependencies determined by dynamic resource overlap. Previous work defines the causal guarantees of BoC operationally, but mixes intended design constraints with incidental implementation details, leading to unintended causal orders. BoC is now being implemented across multiple languages and runtimes, all relying on the operational descriptions of causality. This paper develops an axiomatic model of BoC executions that makes the intrinsic orders explicit and derives the *intended* causal relation from their interaction. Using a set of representative programs and candidate executions, we motivate the design of this causal relation. We then prove that a representative minimal core calculus for BoC is sound with respect to this axiomatic model. Together, these results provide an implementation-independent foundation for reasoning about BoC causality across runtimes, schedulers and optimisation decisions.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Concurrency, Parallelism, Language Design, Causality

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2026.23

Supplementary Material *Software (Proofs)*: `https://github.com/fxpl/axiomatic_boc` [7]
archived at `swh:1:dir:24350cf7167ab1b9d00f8a2d64cc3fe32dfb4560`

Acknowledgements We thank David Black-Schaffer and the reviewers for helpful discussions and feedback on the paper.

1 Introduction

Modern concurrent programming must balance the efficiency benefits of parallelism against the complexity of reasoning about program execution. Behaviour-oriented concurrency (BoC) is a programming model in which tasks declare required isolated resources; tasks with disjoint requirements may execute concurrently, while tasks with overlapping requirements are subject to explicit ordering guarantees, simplifying reasoning about program execution [10]. Previous work on BoC defines these ordering guarantees operationally, but mixes intended design constraints with incidental implementation details, inducing unintended causal orders [10]. BoC is now being developed in multiple languages and runtimes, including C++ and Python runtimes, and research languages such as Verona [22, 12, 13, 4]. As these efforts evolve, they rely on the informal and overly restrictive descriptions of causality inherent in the operational

```
1 def main(src: cown[Account], dst: cown[Account]) {
2   when(src) { A src.balance += 10; }
3   when(dst) { B dst.frozen = true; }
4   when(src, dst) { C
5     if (!dst.frozen && !src.frozen && src.balance >= 10) {
6       src.balance -= 10;
7       dst.balance += 10;
8 } } }
```

Figure 1 Bank account operations using cowns and behaviours.

Initially, both `src` *and* `dst` *have balance 0.*

(a) Update path (`src` then `dst`).

```
1 when(src){ D src.balance += 10; }
2 when(dst){ E dst.balance += 10; }
```

(b) Diagnostic path (`dst` then `src`).

```
1 when(dst){ F print(dst.balance); }
2 when(src){ G print(src.balance); }
```

Figure 2 Independent concurrent behaviours with *unintended* induced causality.

semantics. This impedes the development of optimisations and tooling, and makes it difficult to reason about the intended causal guarantees of BoC across implementations. In this paper, we introduce an implementation-agnostic axiomatic account of BoC that captures the intrinsic orders as well as the intended causal order.

In BoC, an isolated resource is called a *cown* (pronounced "cone") and the concurrent tasks are called *behaviours*. Figure 1 shows an example of BoC where cowns `src` and `dst` each own a unique bank account and behaviours perform asynchronous atomic tasks over accounts. The keyword **when** denotes spawning a behaviour which requires the declared cown set. The behaviours will execute asynchronously once all previously spawned behaviours that require overlapping cowns have completed (and thus the cowns are available). Behaviour A deposits money into the source account, behaviour B freezes the destination account (making future operations on it unavailable), and behaviour C performs a transfer from the source to the destination account if neither is frozen and the source has sufficient balance. The intention of this program is that the deposit and freeze behaviours (A and B) always occur before the transfer behaviour (C) (thus observing the effects of the deposit and freeze), while the deposit and freeze behaviours may proceed concurrently. This example relies on the causal order guaranteed by BoC to ensure that these intentions are met: because the transfer behaviour is spawned after the deposit and freeze behaviours, it will only execute after they have completed, and thus will see their effects.

Figure 2 illustrates a different example in BoC, where the intended behaviour is that the update and diagnostic paths are independent and their behaviours can execute in any order. However, the operational semantics for BoC, presented in earlier work [10], induces an undesirable causal order between the two paths, which is more restrictive than intended. In the operational semantics, it is not possible to have both G execute before D and E execute before F, which would print `src = 0` and `dst = 10`. This outcome stems from the structures and relations defined in the operational semantics, which are not intrinsic to the BoC model but rather reflect a particular implementation strategy.

In order to separate intrinsic causal guarantees from implementation artefacts, we will define causality in BoC using an axiomatic model. Our model follows the event-and-relation methodology used in modern weak-memory axiomatic frameworks [2], but instantiates it with the lifecycle events of behaviours and their dynamic ordering. We introduce the model

Table 1 Sequential executions for Figure 2, grouped by printed output. In the status row, ✓ denotes desirable and permitted under the operational semantics; ▲ denotes desirable but excluded.

	src=0, dst=0	src=0, dst=10	src=10, dst=0	src=10, dst=10
Order	FEGD FGDE FGED GDFE GFDE GFED	EFGD EGDF EGFD GDEF GEDF GEFD	DFEG DFGE DGFE FDEG FDGE FEDG	DEFG DEGF DGEF EDFG EDGF EFDG
Status	✓	▲	✓	✓

through a series of example programs and their candidate executions. We then demonstrate that a minimal core calculus, inspired by the prior operational semantics, is sound with respect to this axiomatic model. This establishes the axiomatic model as a foundation for reasoning about BoC causality across runtimes, schedulers and optimisation decisions.

Contributions

- An informal characterisation of the intended BoC ordering constraints (Section 2).
- An axiomatic model for BoC capturing these constraints precisely (Section 3).
- A representative minimal core calculus for BoC, together with a mechanised Lean proof that its executions are sound with respect to the axiomatic model (Section 4).
- An analysis of the practical utility of the axiomatic model for reasoning about tooling, optimisation, and program transformations (Section 5).

2 An informal description of causality

In earlier work, we provided an informal definition of the happens-before guarantee in the operational semantics for BoC [10]. It is reproduced in Definition 1.

▶ **Definition 1** (Restrictive informal happens-before). *A behaviour b will happen before another behaviour b' iff b and b' require overlapping sets of cowns, and b is spawned before b'.*

This definition restricts behaviours to happen before each other based on their spawning order, which may just be an artefact of an execution, rather than an intrinsic property of the program. Under this definition, we get the desired ordering of behaviours in Figure 1, but we also get the unintended ordering in Figure 2. Table 1 presents the possible interleavings of behaviours for the program in Figure 2 and whether the order is a permissible execution. Only 16 of the 24 possible orders are permissible, and all of the permissible orders have either D happen before G or F happen before E. We summarise this as Observation 2:

▶ **Observation 2.** *In Figure 2* D *happens-before* G ∨ F *happens-before* E

In every execution, D will be spawned before E and F before G, reflecting the program order. Now, assume that we want to print `src = 0`. It must be that G is spawned before D. This means in such an execution, transitively, F will be spawned before E, and thus we will always see `dst = 0` for the restrictive definition of happens-before.

(a) Apply and log account update.

```
1 def update(account: cown[Account], amount: int) {
2   when(account) { H
3     if (account.balance + amount >= 0) {
4       account.balance += amount
5       val id = account.id
6       when(log) { I
7         log.record(id, amount)
8 } } } }
```

(b) Deposit money.

```
1 update(a, +100)
```

(c) Withdraw money.

```
1 update(a, -100)
```

Figure 3 Independent concurrent behaviours with *intended* induced causality.

Table 2 Sequential executions for Figure 3, grouped by update and logging order (+ behaviours corresponds to deposit, − to withdraw). ✓ desirable and permitted; ✗ undesirable and excluded.

	upd(+-), log(+-)	upd(+-), log(-+)	upd(-+), log(+-)	upd(-+), log(-+)
Order	H+ I+ H− I− H+ H− I+ I−	H+ H− I− I+	H− H+ I+ I−	H− H+ I− I+ H− I− H+ I+
Status	✓	✗	✗	✓

Now consider Figure 3, where the restrictive order provides the desired guarantees for a program. The `update` method uses nested **`when`** blocks to access the account, determine whether the update is valid, and then log the update. The recorded log can then be used for auditing or rebuilding the state of the account.

Two independent concurrent updates are shown on the right of Figure 3, one to deposit money and one to withdraw money. Accounts cannot be withdrawn from if they do not have sufficient balance, and such operations will be rejected (Line 3). For example, with an initial balance of 0, if the deposit is applied first, then the final balance after the withdrawal will be 0; whereas, if the withdraw update is applied first, then the final balance after the deposit will be 100. This means the order that these updates are applied will affect the final state of the account, and we want that whichever update is applied first must also be logged first to ensure the log is correct and useable.

Table 2 shows the possible interleavings of the program in Figure 3, whether they are possible under the operational semantics, and whether this is desirable. We denote the deposit, update and log behaviours using a "+" suffix, and the withdraw behaviours using a "−" suffix. In this instance, the desirable and permitted executions are the same, and the undesirable and excluded executions are the same. As summarised by Observation 3, whichever update is applied first will also be logged first.

▶ **Observation 3.**

In Figure 3 (H+ *happens-before* H−) ⟺ (I+ *happens-before* I−)
∧ (H− *happens-before* H+) ⟺ (I− *happens-before* I+)

These examples demonstrate that whilst the operational semantics for BoC captures some of the intended causal order, it does not capture all of it. It is also more restrictive than it needs to be. We could modify the operational semantics to permit the desired executions in Figure 2, but this would complicate the semantics. Instead, in the next section we abstract away from the operational semantics and define the intended causal orders in terms of the intrinsic orders of the programming model.

3 An Axiomatic Model of Causality in BoC

We now use the intuition of causality from the previous section to construct an axiomatic model of BoC executions and judge whether candidate executions are *valid* or *invalid*. In particular, we define the structure of events and relations that form the model, and then define axioms on these relations that determine which executions are valid. This includes defining the more precise happens-before relation that permits the desired executions for our motivating examples in Figure 2 and Figure 3.

3.1 Candidate Executions

Candidate executions represent the possible sequences of events in a BoC program.

▶ **Definition 4** (Structure of BoC executions). *Assuming two sets* BId *and* CId *of behaviour and cown identifiers respectively, a candidate execution is a tuple* $(\mathbb{E}, \mathtt{po}, \mathtt{co})$*:*

(1) $Spawn \subseteq \{S_i \mid i \in \text{BId}\}$
(2) $Run \subseteq \{R_i \mid i \in \text{BId}\}$
(3) $Complete \subseteq \{C_i \mid i \in \text{BId}\}$
(4) $\mathbb{E} = Spawn \cup Run \cup Complete$
(5) $\mathtt{co} \subseteq (\text{CId} \times Complete \times Run)$
(6) $\mathtt{po} \subseteq (Run \times Spawn) \cup (Run \times Complete) \cup (Spawn \times Spawn) \cup (Spawn \times Complete)$

Definition 4 defines the structure of a candidate execution, including the events and relations. A candidate execution is a tuple of a set of events $\mathbb{E}$, a program order relation po, and a cown order relation co. There are three types of events: spawn events, start events, and completion events. These events are self explanatory, with the subscript indicating the behaviour they relate to. The program order (po) relation captures the static order in which a behaviour starts, spawns subsequent behaviours, and completes. The cown order (co) captures the dynamic order in which behaviours using the same cowns complete and start. We do not define the cowns that are used in an execution directly; this information is attached to the order of events in the cown order. This structure assumes that all behaviours have unique identifiers, and that the events of a behaviour are uniquely identified by the behaviour's identifier and the type of event.

For the remainder of this section, we fix an execution $(\mathbb{E}, \mathtt{po}, \mathtt{co})$. All derived relations and functions are defined relative to this fixed execution.

▶ **Definition 5** (Derived relations). *The following relations are derived:*

(1) $\mathtt{r} = \{(S_i, R_i) \mid S_i \in \mathbb{E}, R_i \in \mathbb{E}\}$
(2) $\mathtt{co}_c = \{(C_i, R_j) \mid (c, C_i, R_j) \in \mathtt{co}\}$
(3) $\mathtt{co}_* = \bigcup_{c \in \text{CId}} \mathtt{co}_c$

In Definition 5 we derive three additional relations from the structure of candidate executions: spawn-run order (r), a cown-specific cown order ($\mathtt{co}_c$), and an global cown order ($\mathtt{co}_*$). The second relation projects the cown order onto specific cowns, allowing us to determine the order of events that use the same cowns. The third relation aggregates the cown-specific orders into a single global order, capturing the overall ordering of events across all cowns.

Let us take the example in Figure 3 and illustrate the structure of the candidate execution and the derived relations. We will take an execution where H+ happens-before H− and thus I+ happens-before I−. This execution is illustrated in Figure 4, where the events are ordered from left to right, and the relations are shown as labelled arrows between events.

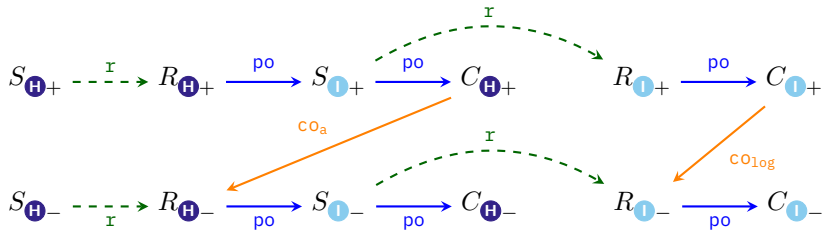


Figure 4 A valid execution of the program in Figure 3, with the events and relations illustrated.

3.2 Deriving Happens-before

We now have the intrinsic events and relations of a BoC execution that enable us to precisely define the intended causal ordering constraints.

▶ **Definition 6** (Cowns). *We define the function* cowns : BId $\to \mathcal{P}(\text{CId})$*, which maps a behaviour identifier to the set of cowns that behaviour uses, as follows:*

$$\text{cowns}(i) = \{c \mid (c, \_, R_i) \in \texttt{co} \ \vee (c, C_i, \_) \in \texttt{co}\ \}$$

▶ **Definition 7** (Happens-before). *We use the notation* $\_^+$ *to denote the transitive closure of a relation. Happens-before is defined as the relation* hb $\subseteq$ *Complete* $\times$ *Run such that:*

$$\texttt{hb} = \{(C_i, R_j) \mid S_i \ (\texttt{po} \cup \texttt{r} \cup \texttt{co})^+ \ S_j \ \wedge \ \text{cowns}(i) \cap \text{cowns}(j) \neq \emptyset \ \wedge \ \{C_i, R_j\} \subseteq \mathbb{E}\}$$

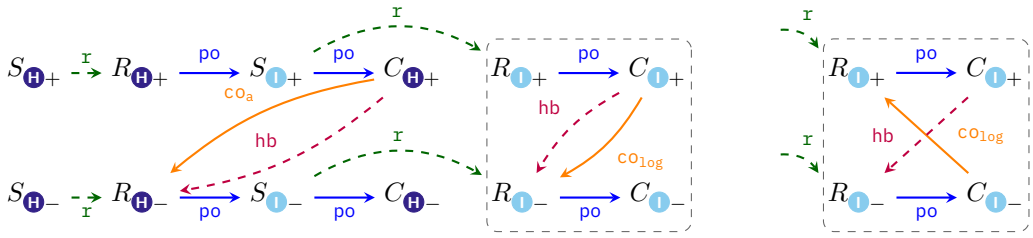
Definition 7 captures the happens-before guarantees which satisfy the intended causal ordering for our motivating examples. For a given complete event C_i and run event R_j, we can determine whether C_i happens-before R_j by finding the spawn events S_i and S_j that correspond to C_i and R_j, then traversing the program order, cown order, and spawn-run order relations to determine whether S_i is causally before S_j. This states that if behaviour i is *causally*, and not incidentally, spawned before behaviour j, and they use overlapping cowns, then the completion of behaviour i happens-before the run of behaviour j.

3.3 Axioms for Valid Candidate Executions

We want all executions which are well-formed, and whose event orders respect the happens-before orders, to be valid executions. In the execution in Figure 4, we can see that C_{I+} happens-before R_{I-}, since S_{I+} po C_{H+} co$_a$ R_{H-} po S_{I-}, and cowns($I+$)$\cap$cowns($I-$) = $\{log\}$. If we add the happens-before edges to the execution in Figure 4, presented in Figure 5a, we can see that the happens-before edges are consistent with cown order of the execution. Thus, this is a valid execution.

Invalid executions are those where the intended happens-before guarantees are not respected. One such execution is a variant of the execution in Figure 4 where C_{I-} co$_{log}$ R_{I+} instead of C_{I+} co$_{log}$ R_{I-}, which would correspond to the withdrawal being applied before the deposit but the log recording the deposit before the withdrawal. This is presented in Figure 5b, where the events and relations are the same as in Figure 4 except for the cown order between C_{I+} and R_{I-}. We also include the derived happens-before relation, which includes the intended happens-before between C_{I+} and R_{I-}.

To judge whether an execution is valid or invalid we use the axioms of the axiomatic model, which define the constraints that a valid execution must satisfy.



(a) A valid execution.

(b) An invalid execution (log order violation).

Figure 5 Two candidate executions of the program in Figure 3.

▶ **Definition 8** (Valid execution). *An execution* $(\mathbb{E}, \mathsf{po}, \mathsf{co})$ *is* **valid** *if it satisfies the following constraints:*

(1) $\forall e_1, e_2 \,.\, e_1 \;\mathsf{po}\; e_2 \;\vee\; e_1 \;\mathsf{co}\; e_2 \implies \{e_1, e_2\} \subseteq \mathbb{E}$ *Relations only relate events in the execution*

(2) $\forall e \in \mathbb{E} \,.\, \exists i \,.\, R_i \;\mathsf{po}^*\; e$ *Every event has a corresponding run event*

(3) $\forall R_i \in \mathbb{E} \,.\, C_j \in \mathbb{E} \,.\, R_i \;\mathsf{po}^*\; C_j \implies i = j$ *Corresponding run and complete events*

(4) $\forall e_1, e_2, e_3 \in \mathbb{E} \,.\, e_1 \;\mathsf{po}\; e_3 \;\wedge\; e_2 \;\mathsf{po}\; e_3 \implies e_1 = e_2$ *Unique ancestors in program order*

(5) $\forall e_1, e_2, e_3 \in \mathbb{E} \,.\, e_1 \;\mathsf{po}\; e_2 \;\wedge\; e_1 \;\mathsf{po}\; e_3 \implies e_2 = e_3$ *Unique descendants in program order*

(6) $\forall e_1, e_2, e_3 \in \mathbb{E}, c \,.\, e_1 \;\mathsf{co}_c\; e_3 \wedge e_2 \;\mathsf{co}_c\; e_3 \implies e_1 = e_2$ *Unique ancestors in cown order*

(7) $\forall e_1, e_2, e_3 \in \mathbb{E}, c \,.\, e_1 \;\mathsf{co}_c\; e_2 \wedge e_1 \;\mathsf{co}_c\; e_3 \implies e_2 = e_3$ *Unique descendants in cown order*

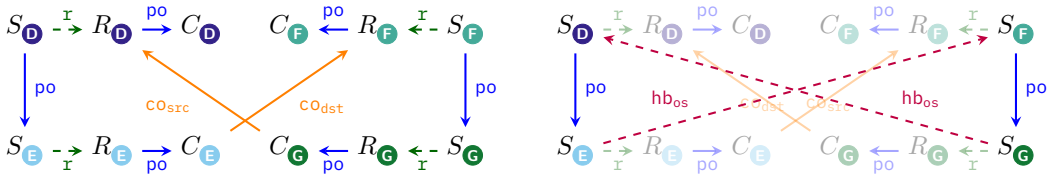
(8) $\forall e_1, e_2, e_3, e_4 \in \mathbb{E}, c \,.\, e_1 \;\mathsf{co}_c\; e_2 \wedge e_3 \;\mathsf{co}_c\; e_4 \wedge e_1 \neq e_3 \implies e_2(\mathsf{po} \cup \mathsf{co}_c)^* e_3 \vee e_4(\mathsf{po} \cup \mathsf{co}_c)^* e_1$ *Cown order forms a single path*

(9) $\forall e_1, e_2 \in \mathbb{E} \,.\, e_1(\mathsf{po} \cup \mathsf{co}_* \cup \mathtt{r} \cup \mathsf{hb})^+ e_2 \implies e_1 \neq e_2$ *Causal consistency*

Items 1–8 are sanity constraints on the structure of executions, ensuring that the relations only relate events in the execution (1), that every event has a corresponding run event (2), that a run event corresponds to the complete event of the same behaviour (3), that the relations do not form a branching structure (4–7), and that the cown order (together with program order) forms a single linear path (8). The final constraint, 9, ensures that the execution is causally consistent; this is achieved by ensuring the transitive closure of the union of the relations, *including the happens-before relation*, is acyclic. Note that we do not require a constraint that all run events have a corresponding spawn event; such a constraint paired with 2 would require an infinite chain of events (or a cycle precluded by 9).

We can now judge both that the execution in Figure 5a is valid, and that the execution in Figure 5b is invalid. The execution in Figure 5a is valid because it satisfies all of the constraints in Definition 8, including the final constraint of causal consistency, since there are no cycles in the union of the relations. The execution in Figure 5b is invalid, since the happens-before relation between $C_{\text{I}+}$ and $R_{\text{I}-}$ forms a cycle, which violates the causal consistency constraint (item 9).

Moreover, all of the executions in the earlier Table 1 are valid under Definition 8, including those excluded by the operational semantics. Figure 6a presents one such valid execution, excluded by the operational semantics, where E runs and completes before F, and G runs and completes before D. There is a deliberate and desired absence of happens-before edges, since no behaviour is causally before any other. In fact, no matter which order the behaviours run and complete, there would be no derived happens-before edges; there is no causal path from the spawn of one behaviour to the spawn of another that uses the same cowns.



(a) Definition 7 does not induce any happens-before relations from the candidate execution.

(b) The operational semantics happens-before, from Definition 9, induces a cycle.

Figure 6 A valid execution which cannot happen in the operational semantics.

```
1 when(src) { J
2   when(log) { K }
3 };
4 when(src, dst) { L };
5 when(dst) { M
6   when(log) { N }
7 };
```

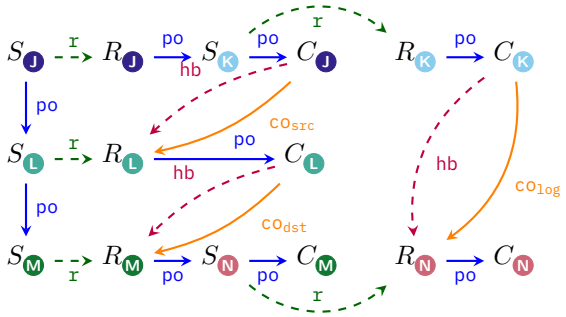


Figure 7 Leap-frogging causality from program order, transitive dependencies, and nesting.

3.4 Causality Through Transitivity

The happens-before relation can also leap-frog over program order, transitive dependencies, and nesting. This is an important result as it shows that the happens-before relation can capture orderings between behaviours whose parents do not use the same cowns. In Figure 7, we have a program where J and M are directly linked only by program order, but are indirectly linked by the intermediate behaviour L, which uses the same cowns as J and M. This means that the completion of J happens-before the run of M, creating a subsequent happens-before relation from the run of K to the run of N. This is illustrated in the execution on the right of Figure 7.

3.5 Capturing the Operational Semantics Happens-before

We can also capture the happens-before relation that would make the operational semantics complete with respect to the axiomatic model. In Definition 9, the happens-before relation acts to retroactively order the spawn events of behaviours based on the cown order of their completion and run events:

▶ **Definition 9** (Operational Semantics Happens-before). *If a behaviour i completes before a behaviour j runs and they use overlapping cowns, then it must have been that behaviour i was spawned before behaviour j.*

$$\mathsf{hb}_{\mathsf{os}} = \{(S_i, S_j) \mid C_i \ \mathsf{co}_* \ R_j\}$$

We illustrate the happens-before relation under the operational semantics definition, in Definition 9, in Figure 6b. Under this definition, the happens-before relation retroactively orders the spawn events of behaviours based on the cown order of their completion and run events, which induces ordering constraints between the behaviours that are not intended by

$$
\begin{array}{rcl@{\qquad}rcl}
i & \in & \text{BId} & e & \in & \text{Event} = \{\, S_i,\, R_i,\, C_i \mid i \in \text{BId} \,\} \\
c & \in & \text{CId} & H & \in & \text{History} = (\text{BId} \to \overline{\text{Event}}) \times (\text{CId} \to \overline{\text{Event}})
\end{array}
$$

$$
\begin{array}{rcll}
s & ::= & \textbf{when}(\overline{c})\{s\}; s \mid \textbf{done} & (\textit{Statements}) \\
b & ::= & (i, \overline{c}, s) & (\textit{Behaviours}) \\
cfg & ::= & \langle \overline{b}, \overline{b} \rangle & (\textit{Configuration})
\end{array}
$$

Figure 8 Syntax of histories and the core BoC calculus. We use $\overline{x}$ for zero or more x's in sequence.

the programmer. In particular, we can see that C_{G} $\texttt{co}_{src}$ R_{D} induces a happens-before edge from S_{G} to S_{D}, and C_{E} $\texttt{co}_{dst}$ R_{F} induces a happens-before edge from S_{E} to S_{F}. This forms a cycle in the relations of the execution, which would violate the causal consistency constraint and would make the execution invalid.

4 Connecting the Axiomatic and Operational Models of BoC

In this section we define a core calculus describing BoC. It is similar to a previous calculus for BoC [10], but differs in that it has a concrete (albeit minimal) sequential language and that it produces a log of events. We relate the core calculus to the axiomatic model and prove that every execution is valid. All definitions and proofs (modulo notational conveniences) have been been mechanised in the Lean theorem prover [7].

4.1 Syntax and Semantics of a Core BoC Calculus

Figure 8 shows the syntax of histories and our core calculus. A history H maps behaviour identifiers i and cown identifiers c to sequences of events. Through slight abuse of notation, we write $H(i)$ and $H(c)$ for looking up the event sequences of behaviours and cowns respectively. Events are defined as the Spawn (S), Run (R) or Complete (C) events of some behaviour, as in the previous section. The intuition is that $H(i)$ contains the history of events caused by behaviour i, while $H(c)$ contains run and completion events of behaviours that require cown c. A history is total and initially maps behaviours and cowns to empty sequences.

A statement s is a sequence of **when** blocks, each of which lists its required cowns and the statement to be run. A behaviour b has an identifier, a set of required cowns and its current statement. A configuration cfg consists of a collection of currently running behaviours and a queue of spawned but not yet running behaviours. In the semantics we use $\overline{b_r}$ and $\overline{b_p}$ for the collections of running and pending behaviours respectively. We write $\overline{b_r}[(i, \overline{c}, s)]$ for a collection $\overline{b'_r} : (i, \overline{c}, s) : \overline{b''_r}$, allowing us to focus on a single behaviour in a collection. We use the shorthands $H[i \mathrel{+}= e]$ and $H[\overline{c} \mathrel{+}= e]$ to append an event e to the history of a behaviour or a set of cown histories respectively.

Figure 9 shows the small-step semantics of our core calculus. The sequential part of the calculus has a single rule Step which executes the next **when** block, emitting a tuple of its required cowns $\overline{c}$ and internal statement s_1. The remaining three rules describe the concurrent semantics of the calculus. They each take the shape $cfg/H \rightsquigarrow cfg'/H'$, performing some scheduling action on cfg and extending the history H with the corresponding event.

The Spawn rule non-deterministically picks a running behaviour and executes its next **when** block (via the Step rule). A new behaviour with a fresh identifier j and the emitted cowns and statement are appended to the end of the queue of pending behaviours. The

$$\frac{}{\textbf{when}(\overline{c})\{s_1\}; s_2 \;\hookrightarrow\; s_2 \mid (\overline{c}, s_1)}\ \textsc{Step}$$

$$\frac{s_1 \;\hookrightarrow\; s_2 \mid (\overline{c_3}, s_3) \qquad j \textbf{ fresh}}{\langle \overline{b_r}[(i, \overline{c_1}, s_1)], \overline{b_p}\rangle \,/\, H \;\rightsquigarrow\; \langle \overline{b_r}[(i, \overline{c_1}, s_2)], \overline{b_p} : (j, \overline{c_3}, s_3)\rangle \,/\, H[i \mathrel{+}= S_j]}\ \textsc{Spawn}$$

$$\frac{C = \{c \mid c \in \overline{c'} \;\wedge\; (\_, \overline{c'}, \_) \in \overline{b_r} \cup \overline{b_{p1}}\} \qquad C \cap \overline{c} = \emptyset}{\langle \overline{b_r}, \overline{b_{p1} : (i, \overline{c}, s) : b_{p2}}\rangle \,/\, H \;\rightsquigarrow\; \langle \overline{b_r} : (i, \overline{c}, s), \overline{b_{p1} : b_{p2}}\rangle \,/\, H[i \mathrel{+}= R_i][\overline{c} \mathrel{+}= R_i]}\ \textsc{Run}$$

$$\frac{}{\langle \overline{b_r}[(i, \overline{c}, \textbf{done})], \overline{b_p}\rangle \,/\, H \;\rightsquigarrow\; \langle \overline{b_r}[\epsilon], \overline{b_p}\rangle \,/\, H[i \mathrel{+}= C_i][\overline{c} \mathrel{+}= C_i]}\ \textsc{Complete}$$

Figure 9 Semantics of the core BoC calculus with histories.

$$\frac{}{i \vdash_b \epsilon}\ \text{WFB-Empty} \qquad \frac{}{i \vdash_b R_i : \overline{S} : (C_i?)}\ \text{WFB-NEmpty}$$

$$\frac{}{\vdash_c \epsilon}\ \text{WFC-Empty} \qquad \frac{}{\vdash_c R_i}\ \text{WFC-Single} \qquad \frac{\vdash_c \overline{E}}{\vdash_c R_i : C_i : \overline{e}}\ \text{WFC-Pair}$$

Figure 10 Well-formedness rules for behaviour and cown histories.

history of behaviour i is extended with the spawn event of the new behaviour. The Run rule picks a pending behaviour whose required cown set does not overlap with the cown set of any running behaviour, nor any behaviour before it in the queue, and adds it to the set of running behaviours. The history of this behaviour is initiated with the run event of that behaviour. Similarly, the history of each required cown is extended with the same run event. Finally, the Complete rule picks a running behaviour whose statement has been reduced to **done** and removes it. This releases the cowns held by the behaviour and allows pending behaviours with overlapping cown sets to spawn in the future. The history of the behaviour as well as the histories of its required cowns are extended with the completion event of that behaviour. Note that the semantics cannot get stuck: it is always possible to apply at least one of the rules until there are no more running or pending behaviours left.

4.2 Properties of Histories

As a program executes, the resulting history describes how that program executed. In this section we define properties of the histories that may be produced by the semantics in the previous section.

A history consists of a set of histories for behaviours and a set of histories for cowns, each of which is a sequence of events. These components have their own well-formedness rules, shown in Figure 10. The behaviour history of behaviour i is well-formed if it has zero or more events, the first which is the run event of i. It is followed by zero or more spawn events and may end with the completion event of i. A cown history is well-formed if it consists of pairs of run and completion events for the same behaviour.

Since we are interested in which event happens before another, we recreate an order using a timestamping function $\tau : \text{Event} \to \mathbb{N}$. We use this timestamping function to formulate the order requirements of histories.

▶ **Definition 10** (Well-timed history). *A history is well-timed with respect to a timestamping function τ, written $\tau \vdash_t H$, if the following conditions are met:*

(1) $\forall i, e_1, e_2 \;.\; H(i) = \_ : e_1 : e_2 : \_ \implies \tau(e_1) < \tau(e_2)$ *Behaviour histories are ordered*

(2) $\forall c, e_1, e_2 \;.\; H(c) = \_ : e_1 : e_2 : \_ \implies \tau(e_1) < \tau(e_2)$ *Cown histories are ordered*

(3) $\forall i, j \;.\; S_i \in H(j) \;\wedge\; R_i \in H(i) \implies \tau(S_i) < \tau(R_i)$ *Spawns are ordered before runs*

(4) $\forall c, i, j \;.\; C_i \in H(c) \;\wedge\; R_j \in H(c) \;\wedge\; \tau(C_i) < \tau(R_j) \implies \tau(S_i) < \tau(S_j)$ *Spawns over the same cowns are ordered by how they ran and completed*

In a well-timed history, events in behaviour and cown histories are ordered before subsequent events (1 and 2), spawn events are ordered before their corresponding runs (3), and spawn events for behaviours using the same cown are ordered according to their completion and run events (4). Note that the last timing requirement corresponds to the $\mathsf{hb}_{\mathsf{os}}$ relation in Definition 9.

▶ **Definition 11** (Well-formed history). *A history is well-formed with respect to a timestamping function, written $\tau \vdash H$, if the following conditions are met:*

(1) $\forall i \;.\; i \vdash_b H(i)$ **(2)** $\forall c \;.\; \vdash_c H(c)$ **(3)** $\tau \vdash_t H$ *See Figure 10 and Definition 10*

(4) $\forall i, j, k \;.\; j \neq k \;\wedge\; S_i \in H(j) \implies S_i \notin H(k)$ *Spawn events are unique*

(5) $\forall c, e \;.\; e \in H(c) \implies \exists i \;.\; e \in H(i)$ *Cown history events are also in some behaviour history*

(6) $\forall c, i \;.\; R_i \in H(c) \;\wedge\; C_i \in H(i) \implies C_i \in H(c)$ *Cown histories don't miss completion events*

A well-formed history combines the three previous relations (1–3) and adds three additional requirements: the same spawn event cannot appear in multiple behaviour histories (4); each event in a cown history corresponds to some event in a behaviour history (5); and if a behavior ran on a cown and later completed, then the completion event appears in that cown's history (6).

In order to connect the semantics to the axiomatic model we define a translation from histories to candidate executions and show that such executions are always valid.

▶ **Definition 12** (Translation to axiomatic model). *A history H is translated to a model $[\![H]\!]$ by getting the program order from adjacent events in behaviour histories and the cown order from adjacent completion and run events in cown histories:*

$$
\begin{aligned}
&[\![H]\!] = (\mathbb{E}, \mathsf{po}\,, \mathsf{co}\,) \\
&\quad \textbf{where} \\
&\qquad \mathbb{E} \;\; = \{e \mid \exists i . e \in H(i)\} \\
&\qquad \mathsf{po} \;\; = \{(e_1, e_2) \mid \exists i . H(i) = \_ : e_1 : e_2 : \_\} \\
&\qquad \mathsf{co} \;\; = \{(C_i, R_i, c) \mid H(c) = \_ : C_i : R_i : \_\}
\end{aligned}
$$

▶ **Theorem 13** (Translation soundness). *A well-formed history translates into a valid execution:*

$$\vdash H \implies [\![H]\!] \textbf{ valid}$$

Proof. Most constraints of Definition 8 are straightforward properties of lists – the more involved parts of the proof are items 8 and 9. For item 8, we prove that there is always a path from one event in a cown history to a later, either via co_c (adjacent complete/run events) or via po^* (the run and completion events in the same behaviour history). For item 9, we prove acyclicity using the well-timedness in Definition 10 to show that the timestamp always increases along the four kinds of relations. We refer to the mechanisation for details [7]. ◀

4.3 Soundness of Core Calculus According to the Axiomatic Model

In the previous section, we showed that a well-formed history translates to a valid execution in the axiomatic model. In this section we show that the semantics is sound according the axiomatic model by proving that it always produces a well-formed history.

In addition to general well-formedness, the history of an execution is going to match the specific configuration that it was produced together with.

▶ **Definition 14** (Matching history). *We say that a history matches a configuration, written* $H; \tau \vdash \langle \overline{b_r}, \overline{b_p} \rangle$*, if the following conditions are met:*

(1) $|b_r| = |\{i \mid (i, \_, \_) \in b_r\}|$ *Running behaviours have unique identifiers*

(2) $\forall i \;.\; (i, \_, \_) \in \overline{b_r} \iff R_i \in H(i) \;\wedge\; C_i \notin H(i)$ *Running behaviours have started*

(3) $\forall (i, \_, \_) \in \overline{b_p} \;.\; R_i \notin H(i)$ *Pending behaviours have not started*

(4) $\forall (i, \_, \_) \in \overline{b_p} \;.\; \exists j \;.\; S_i \in H(j)$ *Pending behaviours have been spawned*

(5) $\forall i, j \;.\; S_i \in H(j) \implies (i, \_, \_) \in \overline{b_p} \;\vee\; (i, \_, \_) \in \overline{b_r} \;\vee\; C_i \in H(i)$
Spawned behaviours are pending, running or have completed

(6) $\forall i \;.\; C_i \in H(i) \implies (j, \_, \_) \notin \overline{b_r} \cup \overline{b_p}$ *Completed behaviours are neither running nor pending*

(7) $\forall i, c \;.\; ((i, \overline{c}, \_) \in \overline{b_r} \;\wedge\; c \in \overline{c}) \iff (R_i \in H(c) \;\wedge\; C_i \notin H(c))$
Cown histories reflect the cown set of running behaviours

(8) $\forall c, i \;.\; R_i \in H(c) \implies \exists j \;.\; S_i \in H(j)$ *Running behaviours requiring cowns were spawned*

(9) $\left[\tau(S_i) \mid (i, \_, \_) \in \overline{b_p}\right]$ **increasing** *The pending queue is sorted on spawn order*

(10) $\forall i, j, c \;.\; R_i \in H(c) \;\wedge\; (j, \overline{c}, \_) \in b_p \;\wedge\; c \in \overline{c} \implies \tau(S_i) < \tau(S_j)$
Pending behaviours were spawned after behaviours that have started

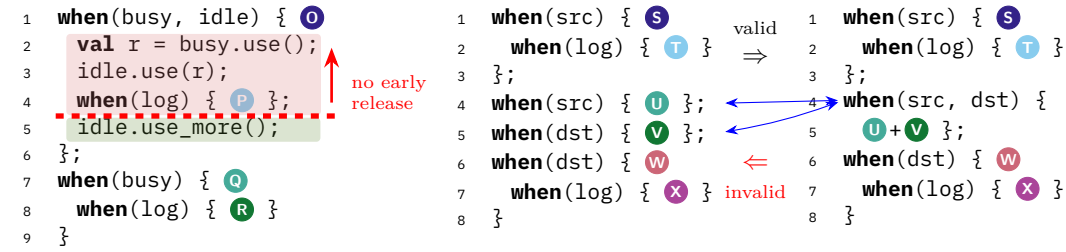
In a history H that matches some configuration, we have that: no two running behaviours share the same identifier (1); a behaviour is currently running if and only if H contains its run event but not its completion event (2); if a behaviour is pending then H does not have its start event (3) but some behaviour has its spawn event (4); if there is a spawn event for some behaviour in H then that behaviour is either pending, running, or its history has a completion event (5); if there is a completion event for some behaviour in H then this behaviour is neither running nor pending (6); a running behaviour is holding cowns $\overline{c}$ if and only if the history of each of these cowns has a run event but no completion event for this behaviour (7); and if a behavior ran on a cown, it must have been spawned by some behavior (8). The last requirement prevents out-of-thin-air behaviours, but avoids the chicken-and-egg problem of how the initial behaviour was spawned by allowing run events to exist without a corresponding spawn event as long as these behaviours do not require any cowns. Furthermore, the pending behaviours must be strictly increasing according to the timestamps of their spawns (9) and a running behaviour that holds some cown must have been spawned before any pending behaviour requiring the same cown (10).

The semantics preserves both well-formed and matching histories.

▶ **Theorem 15** (Preservation of well-formed and matching histories). *Starting from a well-formed and matching history, evaluation produces another well-formed and matching history:*

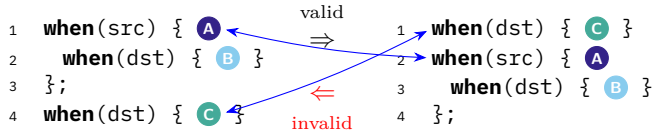
$$\tau \vdash H \;\wedge\; H; \tau \vdash cfg \;\wedge\; cfg/H \rightsquigarrow cfg'/H' \implies \exists \tau' \;.\; \tau' \vdash H' \;\wedge\; H'; \tau' \vdash cfg'$$

Proof. By cases on the evaluation relation. For each case we select a τ' that is τ extended so that the new event produced by that evaluation step is given a timestamp larger than any previous timestamp. Preservation of each of the two relations can then be proved separately. We refer to the mechanisation for details [7]. ◀



(a) We can only release `busy` after O has spawned its last behaviour.

(b) We can merge two behaviours, but we cannot necessarily split a behaviour into two.



(c) We can swap two behaviours when the second (and its children) do not causally depend on the first.

Figure 11 Potential transformations of behaviours with respect to causality.

Together with Theorem 13 this proves that the semantics is sound with respect to the axiomatic model.

5 Discussion

The theorems in Section 4 establish that the operational semantics is sound with respect to the axiomatic model, but not complete. This gap is not a flaw: the axiomatic model characterises the full space of valid executions through intrinsic orders and the derived happens-before relation, serving as a specification for implementations and optimisations beyond what the operational semantics alone permits. We discuss two aspects: program transformations that the model enables and how the model's design could be extended and refined.

5.1 Program Transformations with Respect to Causality

Like Java and C/C++, many languages use axiomatic models to capture relaxed memory semantics and define clear optimisation boundaries. Our axiomatic model of BoC can play a similar role: it defines optimisation boundaries within which compilers and runtimes may optimise while preserving the causal dependencies between behaviours.

Releasing Cowns Early. One potential optimisation is to release cowns early, once they are no longer needed, but before the completion of a behaviour. This can allow more concurrency between behaviours that require the same cowns. Such an optimisation would be useful in cases where a cown is heavily contended. Figure 11a shows an example where the behaviour O uses cowns `busy` and `idle`, and then spawns a behaviour P that uses `log`. Assume that the use of the two cowns in O is such that they must appear together, but once `busy` has been used, `idle` can be used independently. We can use the model of causality to understand that we cannot release `busy` until after O has spawned P. If we were to release `busy` before spawning P, then Q could run and spawn R before O has spawned P, which would

violate the intended causal dependencies between P and R. However, we can release `busy` after spawning P (Line 4) and before the second use of `idle` (Line 5), as this will not violate any causal dependencies. This means that early release of a cown is sound as long as the cown is only released after the last behaviour spawn of a behaviour holding the cown.

Merging and Splitting Behaviours. It may also be useful to merge or split behaviours. This could reduce the overhead of spawning and scheduling small behaviours (for example when multiple behaviours require the same cowns), or to increase the granularity of concurrency. Merging two behaviours into one behaviour that requires the union of their cowns is always sound, but splitting a behaviour into two behaviours that require subsets of the original cowns is not always sound. Figure 11b shows an example of this imbalance. It is sound to merge U and V into a single behaviour that requires both `src` and `dst`, but it is not sound to split this merged behaviour back into two separate behaviours, as this would violate the causal dependencies between T and X. To split the behaviours would require a transformation where the behaviours are split to enable concurrency, and then followed by an empty behaviour that requires both cowns to preserve the causal dependencies. Note also that this would violate the atomicity of the original behaviour U+V.

Reordering Behaviours. Figure 11c shows how this transformation is also asymmetric. The example contains two behaviours A and C that require the different cowns, and B which is spawned by A and requires the same cowns as C. We can swap the order of A and C from left to right without violating the causal dependencies. This will induce a new happens-before edge between C and B, but the executions before and after the transformation will still be valid. However, we cannot swap the order of A and C from right to left, as this would violate the causal dependency between C and B: this transformation would allow B to be spawned and run before C has completed.

5.2 Refining the Axiomatic Model

The model we have introduced in this paper defines the causal dependencies between behaviours. We can use these guarantees to further understand the outcomes of a program, for example, to understand cown read and writes. To model these cown operations explicitly, we could add corresponding events and constrain the causal relations of these access events from the existing behaviour relations. For example, we could require that each read observes the most recent write in happens-before order.

In this paper, we assume that all behaviours can both read and write the cowns they require. This is the model used in prior BoC work, but richer access modes are possible. For example, some behaviours could require a cown only for reading, while others require it for writing. This would permit finer-grained concurrency: read-only behaviours on the same cown could run concurrently, while writers would still require exclusive access. The model could capture this either by relaxing the cown-order axioms to allow fan-out and fan-in, or by enriching the cown order with access mode information. In either case, cown events must remain consistent with each behaviour's declared access mode.

6 Related Work

Memory Models. Memory models are often defined axiomatically, and they have been used to define the relaxed behaviour of concurrent programs on modern hardware architectures and programming languages [14, 2, 17]. These models are similar to our axiomatic model in

that they define a set of events and relations between them, and they specify the allowed executions of a concurrent program. Our model is not focused on memory consistency, but rather on the causal dependencies between behaviours in a concurrent program. Because BoC requires isolation of cowns, the *coherence order* and *reads-from* employed in traditional memory models are derivable from the program order and the cown order (cf. Section 5.2).

Actor Model Programming. The Actor model is a well-known paradigm [11, 1], widely implemented in practical actor runtimes [3, 19]. Its operational presentation is natural because the original model has no explicit inter-behaviour or inter-actor causal dependencies. Some implementations of actor models do provide mechanisms or guarantees for causal consistency and ordering behaviours [21, 20]. In earlier work, we compared actors with BoC and showed that BoC decouples isolation from concurrency, enabling more expressive synchronisation [9] This decoupling also introduces richer inter-behaviour causal dependencies, which are better captured in an axiomatic style than in a purely operational one, where these dependencies must be encoded directly in the semantic rules.

Causal Consistency in Distributed Systems. Causal consistency is a well-known consistency model in distributed systems, ensuring that operations with causal relationships are observed in the same order by all processes; it is classically defined using a happens-before relation [15] and formalised in modern treatments of causal consistency for shared objects [18]. This resembles our derived happens-before relation in that both capture causal dependencies between events.

Consistency Models for Databases. Work on consistency models for databases uses axiomatic models to characterise the visibility of updates between transactions [8]. These models often abstract the nested structure of transactions, treating them as flat, atomic units. While BoC is also concerned with the visibility of updates, the program structure and nesting of behaviours are intrinsic to ordering and causality.

Deterministic Shared-memory Concurrency. Shared-memory deterministic concurrency models, such as Deterministic Parallel Java [5, 6] and Deterministic Threads [16], have been proposed to provide deterministic behaviour in concurrent programs. They typically enforce determinism through combinations of static analysis and runtime checking of shared-memory accesses. By contrast, BoC does not target full determinism nor expect shared memory; it provides a causality-oriented account of concurrent behaviours accessing isolated resources, grounded in dynamic execution.

7 Conclusion

In this paper, we presented an axiomatic model for BoC that captures the intended causal dependencies between behaviours. Using representative example programs and executions, we motivated the model design and showed how the derived happens-before relation separates intrinsic causality from incidental operational ordering. We then established that operational semantics presented in earlier work is sound with respect to this model via a representative core calculus. Taken together, these results provide an implementation-independent foundation for reasoning about BoC causality, while enabling more permissive yet valid executions to be recovered through scheduling and compilation.

References

1 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, December 1986. doi:10.7551/mitpress/1086.001.0001.

2 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2), July 2014. doi:10.1145/2627752.

3 Joe Armstrong. *Concurrent Programming in ERLANG*. Pearson Education. Prentice Hall, 1996. URL: https://books.google.se/books?id=EqZQAAAAMAAJ.

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. doi:10.1145/3622846.

5 Robert L Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. Parallel programming must be deterministic by default. *Usenix HotPar*, 6(10.5555):1855591–1855595, 2009.

6 Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1640089.1640097.

7 Elias Castegren and Luke Cheeseman. A mechanised axiomatic model for behaviour-oriented concurrency, 2026. Software, swhId: swh:1:dir:24350cf7167ab1b9d00f8a2d64cc3fe32dfb4560 (visited on 2026-08-13). URL: https://github.com/fxpl/axiomatic_boc, doi:10.4230/artifacts.27655.

8 Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In Luca Aceto and David de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 58–71, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2015.58.

9 Luke Cheeseman, Elias Castegren, Sophia Drossopoulou, Tobias Wrigstad, Sylvan Clebsch, and Matthew Parkinson. *Decoupling Isolation and Concurrency: An Actor-Centric View of Behaviour-Oriented Concurrency*, pages 165–186. Springer Nature Switzerland, Cham, 2026. doi:10.1007/978-3-032-05291-9_7.

10 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. doi:10.1145/3622852.

11 Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance papers of the conference*, volume 3, page 235. Stanford Research Institute Menlo Park, CA, 1973.

12 Matthew Johnson. Explorations into a programming model for BoC in the Python runtime. https://github.com/matajoh/pyrona, 2025. Accessed February 2025.

13 Matthew Johnson, Matt Parkinson, Sylvan Clebsch, Fridtjof Stoldt, and Tobias Wrigstad. Research programming language for concurrent ownership. https://pep-previews--4468.org.readthedocs.build/pep-0795/, 2025. Accessed December 2025.

14 ISO/IEC JTC1/SC22/WG14. Programming languages — C. Technical Report ISO/IEC 9899:2011, International Organization for Standardization, 2011. Section 6.2.4.

15 Leslie Lamport. *Time, clocks, and the ordering of events in a distributed system*, pages 179–196. Association for Computing Machinery, New York, NY, USA, 2019. doi:10.1145/3335772.3335934.

16 Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems*

Principles, SOSP '11, pages 327–336, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2043556.2043587.

17 Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1040305.1040336.

18 Matthieu Perrin, Achour Mostefaoui, and Claude Jard. Causal consistency: beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016. doi:10.1145/2851141.2851170.

19 Akka Team. Akka: A toolkit for building concurrent, distributed, and resilient message-driven applications on the jvm. https://akka.io/, 2025. Accessed March 2025.

20 Pony Team. Pony: A capabilities-secure, actor-model, high-performance programming language. https://www.ponylang.io/, 2025. Accessed March 2025.

21 Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001. doi:10.1145/583960.583964.

22 Tobias Wrigstad. Pyerlang: A stepping stone towards behaviour-oriented concurrency in python (keynote). In *Proceedings of the 24th ACM SIGPLAN International Workshop on Erlang*, Erlang '25, page 1, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3759161.3771262.