Fair Join Pattern Matching for Actors

Philipp Haller
KTH Royal Institute of Technology, Stockholm, Sweden

Ayman Hussein
Technical University of Denmark, Lyngby, Denmark

Hernán Melgratti
University of Buenos Aires & Conicet, Argentina

Alceste Scalas
Technical University of Denmark, Lyngby, Denmark

Emilio Tuosto
Gran Sasso Science Institute, L'Aquila, Italy

Abstract

Join patterns provide a promising approach to the development of concurrent and distributed message-passing applications. Several variations and implementations have been presented in the literature – but various aspects remain under-explored: in particular, how to specify a suitable notion of message matching, how to implement it correctly and efficiently, and how to systematically evaluate the implementation performance.

In this work we focus on actor-based programming, and study the application of join patterns with conditional guards (i.e., the most expressive and challenging version of join patterns in literature). We formalise a novel specification of *fair and deterministic join pattern matching*, ensuring that older messages are always consumed if they can be matched. We present a *stateful, tree-based join pattern matching algorithm* and prove that it correctly implements our fair and deterministic matching specification. We present a novel Scala 3 actor library (called `JoinActors`) that implements our join pattern formalisation, leveraging macros to provide an intuitive API. Finally, we evaluate the performance of our implementation, by introducing a systematic benchmarking approach that takes into account the nuances of join pattern matching (in particular, its sensitivity to input traffic and complexity of patterns and guards).

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Software and its engineering → Domain specific languages; Software and its engineering → Concurrent programming languages; Software and its engineering → Distributed programming languages; Theory of computation → Process calculi

Keywords and phrases Concurrency, join patterns, join calculus, actor model

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.17

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.8`

Funding *Ayman Hussein*: Research supported by the Horizon Europe grant 101093006 (TaRDIS). *Alceste Scalas*: Research partly supported by the Horizon Europe grant 101093006 (TaRDIS). *Emilio Tuosto*: Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233, the PRIN PNRR project DeLICE (P20223T2MF), "by the MUR dipartimento di eccellenza", and by PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA – Advanced Space Technologies and Research Alliance.

Acknowledgements This work was inspired by the group discussion on "Join patterns / synchronisation – the next generation" [4, page 54] at the Dagstuhl Seminar 21372; we thank the organisers of the meeting and Schloss Dagstuhl – Leibniz Center for Informatics for making this work possible.

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 17; pp. 17:1–17:28

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We thank Omar Inverso for the technical support he provided for our experimental evaluation, Roland Kuhn for fruitful discussions on the shop floor use case, António Ravara for some useful suggestions, and Antoine Sébert for an implementation of join patterns using Scala 3 macros [25]. We thank the anonymous reviewers for their comments and suggestions.

1 Introduction

Programming concurrent and distributed message-passing applications is difficult, especially in scenarios where multiple concurrent processes need to synchronise and exchange data when complex conditions are satisfied. The join calculus [8] introduced *join patterns*, an intriguing construct for concurrent programming that can help address these scenarios. A *join pattern with conditional guard* is reminiscent of a clause in a typical pattern matching construct: it has the form "J if $\gamma \triangleright P$" – where J is a *message pattern* describing a combination of incoming messages and binding zero or more variables, and γ is a *guard*, i.e., a boolean expression that may use the variables bound in J. A program using join patterns can wait until a desired combination of messages arrives (in any order); when some of the messages are matched by the message pattern J and their payloads satisfy the guard γ, the process P is executed. We now illustrate programming with join patterns with an example emerging from an industrial case study where a monitoring program handles a variety of messages emitted by machines and devices deployed on a factory shop floor. (To illustrate our proposal, we only show a representative sample of the the actual monitoring application.)

```
def monitor() = Actor[Event, Unit] {
    receive { (self: ActorRef[Event]) => {
        case ( Fault(_, fid1, _, ts1),
               Fix(_, fid2, ts2) ) if fid1 == fid2 =>
            updateMaintenanceStats(ts1, ts2)
            Continue

        case ( Fault(mid, fid1, descr, ts1),
               Fault(_, fid2, _, ts2),
               Fix(_, fid3, ts3) ) if fid2 == fid3 && ts2 > ts1 + TEN_MIN =>
            updateMaintenanceStats(ts2, ts3)
            log(s"Fault ${fid1} ignored for ${(ts2 - ts1) / ONE_MIN} minutes")
            self ! DelayedFault(mid, fid1, descr, ts1) // For later processing
            Continue

        case ( DelayedFault(_, fid1, _, ts1),
               Fix(_, fid2, ts2) ) if fid1 == fid2 =>
            updateMaintenanceStats(ts1, ts2)
            Continue

        case Shutdown() => Stop
    } }
}
```

Listing 1 Simplified factory shop floor maintenance monitor, written using our Scala 3 library `JoinActors` (presented in Section 4).

Example: Monitoring a Factory Shop Floor. The Scala 3 Listing 1 is structured as an actor [1] that uses our join patterns library `JoinActors` (as introduced in Section 4). Its coding style is reminiscent of popular libraries like Akka and Pekko.[1] The constructor `Actor[Event, Unit]` (line 1) means that the actor's mailbox receives messages of type `Event` (which has various subtypes), and whenever the actor stops running, it yields a `Unit` value. The "`receive { ... }`" block (lines 2–22) executes whenever messages are received, binding "`self`" to a reference to the monitor actor itself (usable to send messages to its mailbox).

The monitor actor in Listing 1 is used in a scenario where machines on the factory shop floor may occasionally require human intervention, so they may emit messages like `Fault(3, 42, "Motion sensor error", 10:31)` carrying information such as the machine and fault identifiers as well as a description and timestamp. When such an event occurs, a technician is expected to reach the machine and report that the fault is being fixed, by using a handheld device to emit a message like `Fix(35, 42, 10:33)` (carrying the worker id, fault id being fixed, and timestamp).

The key difference between the actor depicted in Listing 1 and a "standard" actor in libraries like Akka/Pekko lies in their message processing mechanisms. While the latter can only react to *individual* messages arriving in its mailbox, the actor in Listing 1 reacts whenever a *combination* of messages in its mailbox matches one of the *join patterns with guards* specified within its "`receive { ... }`" block.

- The case on lines 3–4 is triggered when the monitor detects in its mailbox both a `Fault` and a `Fix` message referring to the same fault (guard "`fid1 == fid2`"). In this case, the messages are removed from the mailbox, the monitor updates certain maintenance statistics (line 5), and then resumes execution by returning `Continue` (line 6).
- The case on lines 8–10 activates when the monitor sees *two* `Fault` message and a `Fix` message that handles the most recent fault, with the older fault being emitted more than 10 minutes earlier (guard "`fid2 == fid3 && ts2 > ts1 + TEN_MIN`"). In this case, the monitor also logs a warning and resends the unhandled fault to its own mailbox (as a `DelayedFault`) for later processing (lines 12–13);
- The case on lines 16–17 is similar to the first case above, except that it consumes the `DelayedFault`s emitted by the second case;
- The case on line 21 reacts to a `Shutdown` message by `Stop`ping the monitor.

Notice that the join pattern matching cases do not depend on the order of messages in the mailbox: for instance, the first and second cases in Listing 1 (lines 3–4, 8–10) can be triggered even if, due to network delays or temporary partitions, the `Fix` message reaches the monitor mailbox *before* the corresponding `Fault`.[2]

The monitor in Listing 1 has a declarative and rather intuitive flavour – but writing it without a library (like ours) supporting join patterns is much harder. E.g., to just implement the first and second case (lines 3–4, 8–10), a programmer writing a "regular" Akka/Pekko actor would need to write code for processing one incoming message at a time, remembering how many `Fault`s and/or `Fix`es it has seen thus far, and checking whether any combination creates a match with the newly-arrived message, and satisfies the guards; this handcrafted pattern matching logic should not "forget" any message combination, and should also support messages arriving out-of-order. As the number and complexity of message patterns increases, the handcrafted pattern matching code can become complicated, bug-prone, and inefficient.

[1] `https://akka.io/`, `https://pekko.apache.org/`
[2] The program in Listing 1 only assumes that each device has an accurate-enough clock, so message timestamps can be compared (with some tolerance) to determine which event happened first.

Open Problems. Although promising, join patterns are still subject of research and their adoption has yet to become "mainstream" in programming. In this work we tackle three aspects that, we believe, have been under-explored thus far.

1. *Formalising how a join pattern matching construct should select messages when multiple options are available.* Existing work (both theoretical and implementation-oriented, discussed in Section 2) leaves the message selection unspecified (i.e., allowing for non-determinism in the matching semantics), or follows a "first matching pattern wins" approach – which may cause older messages to be "forgotten" in the mailbox to the advantage of newer messages (we will discuss this in Section 3). This may yield "unfairness" towards the messages in the mailbox: a message in the box is perpetually neglected when "newer" messages are used in the matching.[3]
2. *Implementing join patterns with guards in a correct and efficient way.* Most existing implementations address message patterns without guards [9, 2, 23, 27, 16]. However, supporting guards is much harder: finding a combination of messages in a mailbox that satisfies a guard may require computing up to a factorial number of message combinations, and in order to reduce such computations, it becomes necessary to maintain the state of partial matches. Other authors have considered this issue [12, 20, 22] – but unlike us, they have either not provided a specific notion of matching nor demonstrated that their optimisation approaches (if any) *correctly* adhere to a desired matching specification (see problem 1 above). In fact, the papers [12, 20] do not define a notion of "preferred matching" while such a notion is mentioned in the doctoral thesis [21] without a formal definition nor proof of the properties of their algorithm.
3. *Systematically evaluating join pattern matching performance.* The performance of join pattern matching is highly dependant on the input message traffic and on the complexity of patterns and guards – but these aspects have not been systematically explored, and there is no standardised benchmarking suite for join pattern implementations (akin to Savina [13] for actor implementations). For instance, the measurements in [27] focus on classic synchronisation problems, with simple patterns, and without guards.

Contributions and Structure of the Paper. We address the aforementioned challenges by presenting a novel formalisation and implementation of join pattern matching with guards. After the background and related work (Section 2), we introduce our contributions.

- In Section 3, we present a formal specification of *fair and deterministic join pattern matching* guaranteeing that oldest messages are always consumed if they can be used (Defs. 3.8 and 3.10). We also introduce a *stateful tree-based matching algorithm* (Defs. 3.20 and 3.23), and we prove that it respects the formal specification of fair matching (Theorem 3.25).
- In Section 4 we present `JoinActors`, our Scala 3 library for actors with join patterns, including both a "brute-force" and a stateful tree-based implementation of our deterministic fair matching semantics. `JoinActors` uses macros to provide an intuitive API. `JoinActors` is the companion artifact of this paper.
- In Section 5 we evaluate the relative performance of the matching algorithms implemented in `JoinActors`, including (in Section 5.6) a comparison with an alternative implementation of our fair matching policy that uses the RETE algorithm [6]. Our evaluation explores variations of the input traffic and the complexity of join patterns and guards: we see this

[3] This can be considered a form of *fairness of instruction* according to the terminology in [11].

as a step towards a standardised and systematic benchmarking approach for future join pattern implementations. Overall, our experiments show that the performance of our implementation of the stateful tree-based matching is suitable for applications like floor shop monitoring (described above) or smart house automation (described in Section 5.3).

We conclude and discuss our future work in Section 6 – including alternative matching policies. In this work we have chosen to formalise a "oldest messages first" matching policy because it fits many scenarios – in particular, our factory shop floor monitor (where, as in many application domains, a "first-arrived-first-served policy" is required, and non-deterministic matching would be inadequate), and the examples we could find in literature.

2 Background and Related Work

The Join calculus [8], emerging in the late 1990s as a variant of the asynchronous pi-calculus, aimed at enhancing the implementability of process calculi by introducing disciplined rules regarding locality and scoping. Its distinctive feature is the integration of restriction, recursion, and synchronization into a single language primitive: the *join definition*. A join definition comprises a list of *reaction rules* of the form $J \triangleright P$, where J is the *join pattern* and P is the process associated with the rule. Essentially, a join pattern specifies the message pattern necessary to activate the process P. For instance, in the construct $c_1(x) \wedge c_2(y) \triangleright P$, we have that c_1 and c_2 represent communication channels, and the process P is activated when messages are present in both channels. Hence, if messages like $c_1(m_1)$ and $c_2(m_2)$ are detected, they are consumed, and the process P is executed by substituting the variables x and y with the corresponding values m_1 and m_2 (i.e., P is executed as $P\{^{m_1,m_2}/_{x,y}\}$). A reaction rule can be seen as an evolution of a function definition in a concurrent message-passing setting: a function activates its body upon invocation from another function, through variable substitution – whereas a reaction rule $J \triangleright P$ activates P only when the join pattern J is "invoked" by one or more concurrent processes that send the required input messages; when this happens, P is executed through variable substitution.

Multiple reaction rules can be combined, such as: $c_1(x) \wedge c_2(y) \triangleright P_1 + c_1(z) \wedge c_3(w) \triangleright P_2$. When multiple rules share channels (as c_1 in the previous example), there may be conflicting synchronisations, as rules contend for messages. For example, if channels c_1, c_2, and c_3 in the previous example have a message available, either P_1 or P_2 can be activated. Significantly, all conflicting synchronisations are defined within the same combination of reaction rules: consequently, all consumers of messages within a channel are locally introduced by a definition, eliminating the need for global consensus in synchronisation.

Since its inception, the join calculus has inspired implementations in various programming languages [9, 7, 5, 14, 2, 23, 24, 27]. Early implementation approaches [5] were centered around *matching automata*, where join definitions are compiled into deterministic automata. In this cases, state corresponds to the state of message queues and transitions to the arrival of messages. Although the foundational principles of this approach have since been adopted in other implementations [2, 23, 24], newer methods have evolved to avoid the explicit construction of automata. Initially, most implementations relied on coarse-grained synchronization to guarantee the atomic consumption of messages. However, this strategy has been refined [27] by employing fine-grained concurrency for enhanced scalability. This involves the utilization of lock-free data structures and minimizing message enqueuing whenever possible. Subsequent optimizations have further been explored in implementations incorporating session types [10] to prune the size of matching automata.

The initial implementations adhere to the original join pattern model and not support pattern matching on consumed values. However, subsequent implementations expanded matching capabilities. This line of work has been started in [19], where join patterns were enriched with matching on constant values. This approach has been extended in [16] to incorporate pattern matching on algebraic data structures. An example of this extended approach involves message patterns such as $pop(e) \wedge stack(x :: xs)$ where the pattern associated with the channel *stack* expects a message containing a non-empty list. In such scenarios, efficient pattern satisfaction can be achieved by translating these extended join patterns into equivalent programs. These programs utilize conventional join patterns in their definitions while incorporating ML-style pattern matching in the processes executed after a join pattern match. Also, [16] showed that *linear* message patterns (i.e., where each bound variable occurs once) without guards can be implemented efficiently by checking bit flags.

The implementation of more expressive forms of pattern matching have been studied in [12, 20, 22]. These works are conceptually more similar to ours: unlike [16], these works support join patterns that include *conditional guards*, i.e., their reaction rules may look like $c_1(x) \wedge c_2(y)$ if $x < y \triangleright P$ (resembling pattern matching guards in Erlang or Scala); moreover, these works adapt join patterns to an actor-based setting: in the example above, P is activated when the mailbox of the running actor contains, e.g., the messages $c_1(1)$ and $c_2(2)$ (in any order, and possibly among other messages). The introduction of conditional guards significantly improves the usability of join patterns, but also significantly complicates the implementation of join pattern matching; these works adopt different approaches for finding and selecting a match among incoming messages. Both [12] and [20] adopt a "first-match" approach [26], i.e., given a combination of reaction rules, they select the first one that successfully matches the messages in the mailbox; to find that match, [12] adopts a "stateless brute force" approach (i.e., when the mailbox contains a set of messages that might potentially be matched by a join pattern, it tries all message combinations), while [20] maintains a state containing a cache of partial matches, to reduce unnecessary computations. Also, [22] reportedly adopts a variant of the RETE algorithm [6] to maintain a cache of partial matches (as a *discrimination network*) – but its implementation is not publicly available.

In the join calculus, join definitions have non-deterministic matching policies: when multiple message combinations or patterns are enabled (as we will show in Example 3.5), one option is chosen non-deterministically. Correspondingly, existing work and implementations based on join calculus leave matching policies unspecified, or pick the first pattern that completes a match. However, in scenarios where the message selection policy is critical (as in our factory automation example in Section 1, where earlier events must be handled first), the programmer has to encode the selection logic and maintain complex states to achieve the desired outcome. This paper addresses the issue by formalising *fair and deterministic* join pattern matching, inspired by matching mechanisms in functional languages. Drawing from our real-world factory automation use case, we propose an approach that ensures fair message consumption based on messages "age." While other application scenarios may require different resolution policies, we argue that such policies should be enforced by library mechanisms. (We discuss some alternative policies in Section 6.) In contrast to prior work, we emphasise the formalisation of properties guaranteed by the matching mechanism: we introduce a formal specification of fair matching and prove that a stateful algorithm effectively implements that specification. We also contribute a comprehensive evaluation, as a first step toward a standard benchmark suite for join pattern implementations.

An interesting effect-handlers-based language is formalised in [3] to program different styles of matching across different message streams. Besides a common connection to the join calculus, a key design difference with our work is that we focus on matching messages within

an actor mailbox. The resulting features and applications are very different (e.g. a message in a mailbox may be matched at a later time, after other messages from the same emitter – which is not possible in a stream-based framework. Also, the work [3] does not address a notion of fair matching among juxtaposed "joins over asynchronously arriving events" that compete over the same input messages: in their modelling, no event binding takes precedence over the other, all iterations proceed independently and concurrently [3, page 67:14-15].

3 Formalisation

In this section we present the formalisation and properties of our approach to join pattern matching. In Section 3.1 we formalise the necessary notation, and in Section 3.2 we present a specification of *deterministic and fair matching*, covering both individual join patterns (Def. 3.8) and definitions (Def. 3.10). Then, in Section 3.3 we present a stateful matching algorithm that implements our fair matching specification (Theorem 3.25) while avoiding unnecessary computations.

3.1 Syntax

To abstractly represent messages, we assume a set $\mathbb{C}$ of *constructors* equipped with a map arity assigning a natural number to each constructor; then, $\mathsf{arity}(c) \geq 0$ is the *arity* of $c \in \mathbb{C}$ (c is a constant symbol if $\mathsf{arity}(c) = 0$). A *message* is either a constructor of arity 0 or a term of the form $c(m_1, \ldots, m_n)$ where $c \in \mathbb{C}$, $\mathsf{arity}(c) = n > 0$, and m_i is a message (for $i \in 1..n$). For instance, for the example in Section 1, `Fault(3, 42, "Motion sensor error", 10:31)` is a message, with `Fault` being a constructor of arity 4 (numbers, strings, and timestamps are constant symbols represented in the usual way for readability). Through the paper we use *boolean guards* as pure expressions denoted with γ, using the syntax of boolean expressions of Scala; we also use *mailboxes* denoted as $\mathcal{M}$, as sequences of messages $m_1 \cdot \ldots \cdot m_n$. We will also denote *variables* with the symbols y, w, z, ...

Intuitively, a join pattern is a combination of "messages with variables" that binds the variables occurring therein. Multiple alternative join patterns can be composed in a *join definition*. In Def. 3.1 we formalise join patterns equipped with guards, and join definitions.

▶ **Definition 3.1** (Join patterns and join definitions). *The syntax of* join patterns Π *and* join definitions D *is given by the following grammar:*

$$\Pi ::= J \text{ if } \gamma \qquad \textit{where} \quad J ::= \mu \mid \mu \wedge J \quad \textit{and} \quad \mu ::= m \mid x \mid c(\mu_1, \ldots, \mu_{\mathsf{arity}(c)})$$
$$\mathsf{D} ::= \Pi \mid \Pi + \mathsf{D}$$

We postulate that J if γ must be well-formed*, namely: (i)* linear*, i.e., no variable in J occurs more than once, and (ii)* closed*, i.e., each variable occurring in the guard γ also occurs in J. We will often simply write J as shorthand for J if* `true`*.*

Assumption (i) in Def. 3.1 is quite standard: e.g., Scala and F# require linear use of pattern matching variables, and non-linear use can be simulated using guards (see Example 3.2 below). Assumption (ii) does not limit our results: in fact, if a guard contains variables bound elsewhere in the surrounding program, then all such variables would be substituted by values (thus "closing" the join pattern) *before* any match is attempted.

▶ **Example 3.2** (Well-formedness of join patterns). The join patterns shown in Example 3.4 (and also in Listing 1) are well-formed, since they are both linear and closed, whereas

$$\texttt{Fault}(mid_1, fid, descr_1, ts_1) \wedge \texttt{Fix}(wid_2, fid, ts_2)$$

is not-well formed, since the double occurrence of variable fid violates linearity. Intuitively, repeating fid can be a convenient way to state that the same fault id fid must appear in both messages. This is not supported by our formalisation – but the same effect can be obtained by linearising the join pattern: it is sufficient to rename the variables into fid_1 and fid_2 and introduce a guard $fid_1 = fid_2$, obtaining the first pattern shown in Example 3.4 below. ⌟

We write $\{m_1,\ldots,m_n/x_1,\ldots,x_n\}$ for a *substitution*, that is a map that replaces each variable x_i for message m_i. A substitution σ can be applied to join patterns and guards; for instance, the application of the substitution $\sigma = \{42/x\}$ to the join pattern $J = \texttt{Message}(x)$, written $J\sigma$, yields $\texttt{Message}(42)$. Similarly $(isOdd(x))\sigma = isOdd(42)$.

▶ Remark 3.3. *A typical join pattern rule has the form J if $\gamma \triangleright P$. Following the formalisation of ML-style pattern matching in [17], we omit the continuation process P to focus on the matching semantics. Adding continuations P to our formalisation is routine: it would be enough to apply substitutions σ produced by a match to the omitted process, as $P\sigma$.*

Intuitively, a mailbox $\mathcal{M} = m_1 \cdot \ldots \cdot m_n$ yields a match for the join pattern $\Pi = \mu_1 \wedge \ldots \wedge \mu_m$ if γ in D if there is a substitution σ replacing all the variables in Π with some of the messages in the mailbox, such that $\gamma\sigma$ holds true; each message in $\mathcal{M}$ can be used at most once. A *variable* x matches any message, whereas a message constructor pattern like $\texttt{Fault}(x, 42, y, w)$ can only match a message built with a corresponding constructor, like `Fault(3, 42, "Sensor error", 10:31)` with the substitution $\{3,\text{"Sensor error"},123456/x,y,w\}$. (For the precise matching semantics, see in Section 3.2.) Observe that when all variables in a join pattern Π are substituted we obtain one or more $\wedge$-separated messages; likewise, when all variables in a guard γ are substituted, we can evaluate the boolean expression (e.g., a predicate like $\gamma\sigma = isOdd(42)$ might evaluate to false).

A join definition $\mathsf{D} = \Pi_1 + \ldots + \Pi_k$ specifies a pattern matching operation among one of the join patterns with guards $\Pi_1 \ldots \Pi_k$. This formal notation abstracts the construct `receive {...}` shown in Listing 1: each `case` in the `receive {...}` is a join pattern in D.

▶ **Example 3.4** (Syntax of join definitions)**.** Assuming $\texttt{Fault}, \texttt{Fix} \in \mathbb{C}$ and adopting the syntax in Def. 3.1 and of boolean Scala expressions, the first two join patterns in Listing 1 are:

$$\begin{array}{lll} & \texttt{Fault}(mid_1, fid_1, descr_1, ts_1) \wedge \texttt{Fix}(wid_2, fid_2, ts_2) & \text{if } fid_1 = fid_2 \\ + & \texttt{Fault}(mid_1, fid_1, descr_1, ts_1) \wedge \texttt{Fault}(mid_2, fid_2, descr_2, ts_2) & \\ & \quad \wedge \texttt{Fix}(wid_3, fid_3, ts_3) & \text{if } fid_2 = fid_3 \;\&\&\; ts_2 > ts_1 + \texttt{TEN\_MIN} \end{array}$$

where `TEN_MIN` is a Scala constant representing 10 minutes; for readability, similar constants are silently assumed throughout our examples. ⌟

3.2 Fair and Deterministic Matching Semantics for Join Patterns

We now define the notion of pattern matching for a join pattern $\Pi = \mu_1 \wedge \ldots \wedge \mu_n$ if γ. If we have $n = 1$ and a single message m, we can apply standard definitions from functional programming languages [17] and say that Π *matches* m if there is a substitution σ such that (i) $\mu_1\sigma = m$, and (ii) $\gamma\sigma$ evaluates to true. (Clearly, such a match can only happen if σ substitutes *all* variables occurring in μ.) Instead, if $n > 1$ and we have multiple messages available in a mailbox $\mathcal{M}$, things are more difficult: there may be multiple ways for the message pattern $\mu_1 \wedge \ldots \wedge \mu_n$ to match different subsets of messages in $\mathcal{M}$ while satisfying the guard γ; moreover, a join definition D might contain several join patterns that match (part of) the mailbox contents. Example 3.5 illustrates this.

▶ **Example 3.5** (Multiple options for join pattern matching). Let $\mathsf{D} = \Pi_1 + \Pi_2$ where:

$$\Pi_1 = \texttt{Fault}(id_1, \_) \wedge \texttt{Fix}(id_2) \text{ if } id_1 = id_2$$
$$\Pi_2 = \texttt{Fault}(\_, t_1) \wedge \texttt{Fault}(id_2, t_2) \wedge \texttt{Fix}(id_3) \text{ if } id_2 = id_3 \;\&\&\; t_2 > t_1 + \texttt{TEN\_MIN}$$

(observe that D corresponds to the first two cases in Listing 1 modulo the the omission of unused messages and variables). Suppose we have the following mailbox, where subscripts show the arrival order of messages (the lower the subscript, the older the message):

$$\mathcal{M} = \texttt{Fault}_1(1, 10{:}35) \cdot \texttt{Fault}_2(2, 10{:}39) \cdot \texttt{Fault}_3(3, 10{:}56) \cdot \texttt{Fix}_4(3)$$

Before message $\texttt{Fix}_4$ lands in the mailbox, none of the join patterns matches any message combination in the mailbox. Instead, when $\texttt{Fix}_4$ arrives, we have the following options:

- the first join pattern matches the messages $\texttt{Fault}_3$ and $\texttt{Fix}_4$, via the substitution $\{^{3,3}/_{id_1,id_2}\}$
- the second join pattern matches both:
  - messages $\texttt{Fault}_1$, $\texttt{Fault}_3$, and $\texttt{Fix}_4$, via the substitution $\{^{3,3,10:35,10:56}/_{id_1,id_3,t_1,t_2}\}$;
  - messages $\texttt{Fault}_2$, $\texttt{Fault}_3$, and $\texttt{Fix}_4$, via the substitution $\{^{3,3,10:39,10:56}/_{id_1,id_3,t_1,t_2}\}$. ⌟

Existing implementations of join patterns leave the resolution of non-deterministic choices unspecified, or pick the first matching pattern as the "winner." Our approach is different: we formalise a deterministic matching policy to give programmers control over the selection process. Consequently, we specify how to deterministically choose the messages matched by a join pattern in Example 3.5, as well as deterministically decide which join pattern "wins" when both match messages in the mailbox.

Def. 3.8 below formalises a *deterministic* and *fair* notion of join pattern matching: when a join pattern can match multiple combinations of messages in the mailbox, we prioritise the combination that consumes the oldest messages. To this end, we first introduce some notation in Def. 3.6 that we will use to reason about mailbox contents.

▶ **Definition 3.6** (Sequence length, indexing, slicing, and ordering). *Given a set $\mathbb{S}$ and a sequence $\mathcal{S} = s_1 \cdot \ldots \cdot s_n$ containing elements of $\mathbb{S}$, we write $|\mathcal{S}|$ for the length of $\mathcal{S}$, ϵ for the empty sequence, and $\mathcal{S}[i]$ with $i \in 1..|\mathcal{S}|$ for the element at the i^{th} position of $\mathcal{S}$. An* indexing sequence*, denoted by $\mathcal{I}$, is a non-empty sequence of pairwise-distinct natural numbers greater than 0. Given a sequence $\mathcal{S}$ and an indexing sequence $\mathcal{I} = i_1 \cdot \ldots \cdot i_n$ such that $i_h \in 1..|\mathcal{S}|$ for each $h \in 1..n$, we write $\mathcal{S}[\mathcal{I}]$ for the $\mathcal{I}$-*slice of $\mathcal{S}$, *which is the sequence $\mathcal{S}[i_1] \cdot \ldots \cdot \mathcal{S}[i_n]$.*

Let $\mathbb{S}$ be a set with a total order $\sqsubseteq$. Then, the lexicographic order $\leq_{\mathsf{lex}}$ *is the relation on sequences in $\mathbb{S}^*$ inductively defined as: (note that $\leq_{\mathsf{lex}}$ only relates sequences of equal length)*

$$\frac{}{\epsilon \leq_{\mathsf{lex}} \epsilon} \qquad \frac{s \sqsubseteq s' \quad s \neq s'}{s \cdot \mathcal{S} \leq_{\mathsf{lex}} s' \cdot \mathcal{S}'} \qquad \frac{\mathcal{S} \leq_{\mathsf{lex}} \mathcal{S}'}{s \cdot \mathcal{S} \leq_{\mathsf{lex}} s \cdot \mathcal{S}'}$$

Letting $sort(\mathcal{S})$ be the function returning the sorted sequence of elements of $\mathcal{S}$ based on the total order $\sqsubseteq$, we also define the sorted length-biased lexicographic order $\leqslant_{\mathsf{slex}}$ *as:*

$$\frac{n = |S'| \leq |S| \qquad sort(\mathcal{S})[1 \cdot \ldots \cdot n] \leqslant_{\mathsf{lex}} sort(\mathcal{S}')}{\mathcal{S} \leqslant_{\mathsf{slex}} \mathcal{S}'}$$

We also define $=_{\mathsf{slex}}$ as $\leqslant_{\mathsf{slex}} \cap \leqslant_{\mathsf{slex}}^{-1}$. Note that $\leqslant_{\mathsf{slex}}$ is a preorder.

▶ **Example 3.7.** Using relations $\leq_{\mathsf{lex}}$ and $\leqslant_{\mathsf{slex}}$ in Def. 3.6 on sequences of integers, we have:

$$1 \cdot 2 \cdot 3 \leq_{\mathsf{lex}} 1 \cdot 3 \cdot 2 \leq_{\mathsf{lex}} 2 \cdot 1 \cdot 3 \leq_{\mathsf{lex}} 2 \cdot 3 \cdot 1 \leq_{\mathsf{lex}} 3 \cdot 1 \cdot 2 \leq_{\mathsf{lex}} 3 \cdot 2 \cdot 1$$
$$1 \cdot 2 \cdot 3 \cdot 4 =_{\mathsf{slex}} 4 \cdot 3 \cdot 2 \cdot 1 <_{\mathsf{slex}} 1 \cdot 2 \cdot 3$$

▶ **Definition 3.8** (Fair join pattern matching). *We define the following judgements*

$\mathcal{M} \models_\sigma \Pi$ — *the join pattern* Π exactly matches *mailbox* $\mathcal{M}$ *via substitution* σ
$\mathcal{M} \models_\mathcal{I} \Pi$ — *the join pattern* Π sparsely matches *mailbox* $\mathcal{M}$ *via slice* $\mathcal{I}$
$\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}$ — *the join pattern* Π fairly matches *mailbox* $\mathcal{M}$ *via slice* $\mathcal{I}$

by the following inference rules:

$$\frac{\forall i \in \{1,\ldots,n\} : \ \mu_i\sigma = m_i \quad \gamma\sigma}{m_1 \cdot \ldots \cdot m_n \ \models_\sigma \ \mu_1 \wedge \ldots \wedge \mu_n \text{ if } \gamma} \qquad \frac{\mathcal{M}[\mathcal{I}] \models_\sigma \Pi}{\mathcal{M} \models_\mathcal{I} \Pi} \qquad \frac{\mathcal{M} \models_\mathcal{I} \Pi \quad \forall \mathcal{I}' : \mathcal{M} \models_{\mathcal{I}'} \Pi \implies \mathcal{I} \leqslant_{\text{lex}} \mathcal{I}'}{\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}}$$

In Def. 3.8, the judgement $\mathcal{M} \models_\sigma J \text{ if } \gamma$ holds if J exactly matches *all* the messages in $\mathcal{M}$ *in the same order* they occur therein, through a substitution σ such that the guard $\gamma\sigma$ holds.

This exact matching is used in the premise of judgement $\mathcal{M} \models_\mathcal{I} J \text{ if } \gamma$, which matches a *slice* $\mathcal{I}$ of the mailbox $\mathcal{M}$: i.e., the message patterns in J may only match a (possibly reordered) subsequence $\mathcal{M}[\mathcal{I}]$ of the mailbox $\mathcal{M}$. Notice that the slice $\mathcal{I}$ and the pattern J contain the same number of messages. Finally, the judgement $\mathcal{M} \models J \text{ if } \gamma \rightsquigarrow \mathcal{I}$ selects the smallest slice $\mathcal{I}$ of $\mathcal{M}$ w.r.t the order $\leqslant_{\text{lex}}$ in Def. 3.6 such that $\mathcal{M} \models_\mathcal{I} J \text{ if } \gamma$ holds. The selected slice $\mathcal{I}$ represents the "fairest" possible match: $\mathcal{I}$ indexes the oldest messages in $\mathcal{M}$ that match J and satisfy the guard γ. This matching policy ensures that no message is left indefinitely in the mailbox if it *can* be used to match the join pattern. Note that, if two slices $\mathcal{I}$ and $\mathcal{I}'$ in the premise of the judgement contain the same indexes in different order (i.e., they may be deemed "equally fair"), the ordering $\leqslant_{\text{lex}}$ deterministically selects the slice which minimises reordering between messages in $\mathcal{M}$ and message patterns in J.

▶ **Example 3.9** (Fair join pattern matching). Let Π_1, Π_2, and $\mathcal{M}$ be as in Example 3.5. By Def. 3.8 we have that:

- There is a single match for Π_1: $\mathcal{M}[3 \cdot 4] \models_\sigma \Pi_1$ via $\sigma = \{^{3,3}/_{id_1,id_2}\}$. Hence, we also have $\mathcal{M} \models_{[3\cdot4]} \Pi_1$ and we also get $\mathcal{M} \models \Pi_1 \rightsquigarrow [3 \cdot 4]$ (i.e., the fairest way to match the join pattern Π_1 is to consume messages $\texttt{Fault}_3$ and $\texttt{Fix}_4$).
- There are two matches for Π_2:
  - $\mathcal{M}[1 \cdot 3 \cdot 4] \models_\sigma \Pi_2$ via $\sigma = \{^{3,3,10:35,10:56}/_{id_1,id_3,t_1,t_2}\}$. Therefore, $\mathcal{M} \models_{[1\cdot3\cdot4]} \Pi_1$;
  - $\mathcal{M}[2 \cdot 3 \cdot 4] \models_\sigma \Pi_2$ via $\sigma = \{^{3,3,10:39,10:56}/_{id_1,id_3,t_1,t_2}\}$. Therefore, $\mathcal{M} \models_{[2\cdot3\cdot4]} \Pi_2$.

  Hence, since $1 \cdot 3 \cdot 4 \leq_{\text{lex}} 2 \cdot 3 \cdot 4$, we conclude $\mathcal{M} \models \Pi_2 \rightsquigarrow [1 \cdot 3 \cdot 4]$ (i.e., the fairest way to match the join pattern Π_2 is to consume messages $\texttt{Fault}_1$, $\texttt{Fault}_3$, and $\texttt{Fix}_4$). ⌟

Def. 3.10 concludes this section by extending the notion of fair join pattern matching (Def. 3.8) to join definitions. The idea is that if a mailbox $\mathcal{M}$ allows for multiple fair matches on different patterns in a join definition D, we pick the i^{th} join pattern in D that matches $\mathcal{M}$ via the slice $\mathcal{I}$ with the highest number of oldest messages w.r.t. the alternatives; and if two patterns in D yield equally fair matches, we pick the first in D. Since join patterns in D may match slices of different length, we rank the matches using $\leqslant_{\text{slex}}$ (Def. 3.6). This approach makes the choice of patterns deterministic, while ensuring fairness.

▶ **Definition 3.10** (Matching of join definitions). *The judgement* $\mathcal{M} \models \mathsf{D} \rightsquigarrow \mathcal{I}, i$ *(read: "*D *fairly matches mailbox* $\mathcal{M}$ *via slice* $\mathcal{I}$ *by its* i^{th} *join pattern") is defined as:*

$$\frac{\begin{array}{c} \mathit{Matches} = \left\{(\mathcal{I}, i) \;\middle|\; i \in \{1,\ldots,n\} \text{ and } \mathcal{M} \models \Pi_i \rightsquigarrow \mathcal{I}\right\} \\ (\mathcal{I}, i) \in \mathit{Matches} \qquad \forall (\mathcal{I}', i') \in \mathit{Matches} : \ \mathcal{I} <_{\text{slex}} \mathcal{I}' \text{ or } (\mathcal{I} =_{\text{slex}} \mathcal{I}' \text{ and } i \leq i') \end{array}}{\mathcal{M} \models \Pi_1 + \Pi_2 + \ldots + \Pi_n \rightsquigarrow \mathcal{I}, i}$$

▶ **Example 3.11** (Selecting the fairest match across join definitions (1))**.** Continuing Example 3.9, we can now apply Def. 3.10 to determine the fairest match for the join patterns sum $\mathsf{D} = \Pi_1 + \Pi_2$. Since we have both:

$$\mathcal{M} \models \Pi_1 \rightsquigarrow [3 \cdot 4] \qquad \text{and} \qquad \mathcal{M} \models \Pi_2 \rightsquigarrow [1 \cdot 3 \cdot 4]$$

we rank the selected slices as $1 \cdot 3 \cdot 4 <_{\mathsf{slex}} 3 \cdot 4$ (by Def. 3.6), i.e., the slice matched by Π_2 is fairer than the slice matched by Π_1. Therefore, we conclude $\mathcal{M} \models \mathsf{D} \rightsquigarrow (1 \cdot 3 \cdot 4), 2$. ⌟

▶ **Example 3.12** (Selecting the fairest match across join definitions (2))**.** To see why the relation $\leqslant_{\mathsf{slex}}$ (Def. 3.6) considers the lexicographical ordering of *sorted* sequences, consider this variation of Example 3.5:

$$\begin{array}{ll} & \texttt{Fault}(id_1, \_) \wedge \texttt{Fix}(id_2) \qquad\qquad\qquad\qquad \text{if } id_1 = id_2 \\ + & \texttt{Fix}(id_3) \wedge \texttt{Fault}(\_, t_1) \wedge \texttt{Fault}(id_2, t_2) \text{ if } id_2 = id_3 \;\&\&\; t_2 > t_1 + \texttt{TEN\_MIN} \end{array}$$

Let Π_1 and Π_2 be the two join patterns above. Take the same mailbox $\mathcal{M}$ used in Example 3.5. Observe that the message pattern constructors in Π_2 are reordered w.r.t. Example 3.5 – and therefore, we now have $\mathcal{M} \models \Pi_2 \rightsquigarrow [4 \cdot 1 \cdot 3]$ (i.e., the fairest match of Π_2 now consumes the slice $4 \cdot 1 \cdot 3$ of $\mathcal{M}$). Intuitively, this slice is lexicographically greater than the fairest slice $3 \cdot 4$ matched by Π_1 – but the slice $4 \cdot 1 \cdot 3$ consumes the older message at index 1. For this reason, by Def. 3.6 we have $4 \cdot 1 \cdot 3 =_{\mathsf{slex}} 1 \cdot 3 \cdot 4 <_{\mathsf{slex}} 3 \cdot 4$ – and consequently, the fairest match of Π_2 is ranked fairer than the fairest match of Π_1. As a result, we obtain $\mathcal{M} \models \mathsf{D} \rightsquigarrow (4 \cdot 1 \cdot 3), 2$ (by Def. 3.10) – i.e., despite the reordering of the message pattern constructors in Π_2, we match the same messages of Example 3.11 (albeit with a differently-ordered slice). ⌟

3.3 Stateful, Tree-Based Matching Semantics for Join Patterns

Def. 3.8 offers a high-level specification for our notion of "fair matching" – but this definition does not automatically lead to an efficient implementation. To the contrary, the direct implementation of Def. 3.8 is a "stateless" brute-force algorithm that, whenever a new message reaches the mailbox: (*i*) enumerates all possible matches; (*ii*) lexicographically sorts the matches satisfying the guard γ, depending on the messages they use; and (*iii*) picks the match on the smallest mailbox slice (using the lexicographical ordering $\leqslant_{\mathsf{lex}}$ in Def. 3.6). This may require computing up to $n!$ message combinations for a mailbox of length n, every time a new message arrives. A similar brute-force approach is adopted in previous implementations of join patterns in literature [12]. A source of inefficiency is that many message combinations may be uselessly recomputed and retried when a new message arrives, even if such combinations have previously failed by falsifying the guard γ. Furthermore, when a new message yields two or more possible matches, finding the fairest one may lead to redundant computations. These issues are illustrated in Examples 3.13 and 3.14 below.

▶ **Example 3.13** (Redundant matching computations)**.** Consider the join pattern Π_1 from Example 3.5 (recall that $\Pi_1 = \texttt{Fault}(id_1, \_) \wedge \texttt{Fix}(id_2)$ if $id_1 = id_2$), and the following mailbox, where a message $\texttt{Fix}_1$ (emitted by a factory worker's handheld device) is delivered to the monitor before the corresponding $\texttt{Fault}_4$ (emitted by a machine):

$$\mathcal{M} \quad = \quad \texttt{Fix}_1(3) \cdot \texttt{Fault}_2(1, 10{:}35) \cdot \texttt{Fault}_3(2, 10{:}36) \cdot \texttt{Fault}_4(3, 10{:}37)$$

We have to search for a match every time a new message lands in the mailbox:

- Π_1 cannot match message $\texttt{Fix}_1$ alone;
- when the message $\texttt{Fault}_2$ is delivered, the Π_1 matches $\texttt{Fix}_1 \cdot \texttt{Fault}_2$ – but the guard $id_1 = id_2$ is not satisfied;

- when the message $\texttt{Fault}_3$ is delivered, the Π_1 can match $\texttt{Fix}_1 \cdot \texttt{Fault}_2 \cdot \texttt{Fault}_3$ in two possible ways using the combinations $(\texttt{Fix}_1, \texttt{Fault}_2)$ and $(\texttt{Fix}_1, \texttt{Fault}_3)$, neither of which satisfies the guard – note that $(\texttt{Fix}_1, \texttt{Fault}_2)$ has already been attempted;
- when the message $\texttt{Fault}_4$ is delivered, the Π_1 matches $\texttt{Fix}_1 \cdot \texttt{Fault}_2 \cdot \texttt{Fault}_3 \cdot \texttt{Fault}_4$ in a third possible way besides the previous two: in fact, the combination $(\texttt{Fix}_1, \texttt{Fault}_4)$ satisfies the guard – again note that the two failing combinations were already attempted. ⌟

▶ **Example 3.14** (Redundant fairness computations). Consider Π_2 and mailbox $\mathcal{M}$ from Example 3.5:

$$\Pi_2 \;=\; \texttt{Fault}(\_, t_1) \;\wedge\; \texttt{Fault}(id_2, t_2) \;\wedge\; \texttt{Fix}(id_3) \text{ if } id_2 = id_3 \wedge t_2 > t_1 + \texttt{TEN\_MIN}$$

$$\mathcal{M} \;=\; \texttt{Fault}_1(1, 10{:}35) \cdot \texttt{Fault}_2(2, 10{:}39) \cdot \texttt{Fault}_3(3, 10{:}56) \cdot \texttt{Fix}_4(3)$$

When the message $\texttt{Fix}_4$ lands in $\mathcal{M}$, the join pattern Π_2 matches two combinations of messages (as previously shown in Example 3.9), and they should be compared to determine the fairest. However, as soon as we determine that the combination $(\texttt{Fault}_1, \texttt{Fault}_3, \texttt{Fix}_4)$ satisfies the guard, it is redundant to try and compare other combinations – because none of them consumes the message $\texttt{Fault}_1$, hence they cannot possibly be fairer, by Def. 3.8. ⌟

We present an algorithm to tackle these inefficiencies based on a *stateful* solution. Our algorithm keeps track of how the messages in a mailbox $\mathcal{M}$ can *partially* match a join pattern Π, through a tree structure whose nodes contains sets of message indexes in $\mathcal{M}$, decorated with information on how such messages may complete Π. The way the tree is incrementally constructed allows us to (1) avoid recomputing previously-failed matches, and (2) immediately produce the fairest match (if it exists) via a depth-first traversal.

We use *mailbox trees (m-trees)* (Def. 3.15) to arrange integers (representing the indexes of the messages in a mailbox) into sets that form the nodes of a tree, so that, for each branch (X, Y) in the tree, the child node Y is a superset of its parent node X and $\max X < \max Y$.

▶ **Definition 3.15** (Mailbox trees). *A* mailbox tree on a finite set of natural number I *(*m-tree on I *for short) is a tree $T = (N, E)$ where:*

- *$N \subseteq 2^I$ is the set of nodes and $\emptyset \in N$ is the root the tree*
- *the cardinality of each node equals its level in T and, for nodes X and Y at the same level, $X \cap Y = \emptyset$ and X precedes Y if $\max X \leq \max Y$*
- *for each edge $(X, Y) \in E$, $X \subset Y$ and $\max Y \notin X$.*

We write $X \in T$ if X is a node of T and $i \in T$ if there is $X \in T$ such that $i \in X$. An m-tree T on I is consistent *when, for each level $h > 0$ of T, $\bigcup\{X \in T \mid X \text{ is at level } h\} = I$.*

The "ramification" operation (Def. 3.16 below) is used to incrementally extend an m-tree by adding the index i of a new messages that has landed in a mailbox.

▶ **Definition 3.16** (Ramification). *Given a tree $T = (N, E)$ where the elements of N are subsets of numbers, and given a natural number $i \notin T$, let:*

$$N' = N \cup \{X \cup \{i\} \mid X \in N\} \qquad \text{and} \qquad E' = E \cup \{(Y \setminus \{\max Y\}, Y) \mid Y \in N'\}$$

Then, we call $\mathsf{r}(T, i) = (N', E')$ the ramification of T with i.

Note that the ramification of a tree T has twice as many nodes as T. Also, the construction of m-trees does not depend on the order in which messages indexes are added, as shown in Proposition 3.17 below. This allows us to abbreviate $\mathsf{r}(\ldots \mathsf{r}(T, i_1), \ldots, i_n)$ as $\mathsf{r}(T, \{i_1, \ldots, i_n\})$.

▶ **Proposition 3.17.** *Ramification is a commutative internal operation on m-trees.*

Proof. That ramification is internal on m-trees follows by construction given how edges are extended in Def. 3.16. Commutativity follows by induction on the structure of T. ◀

In Def. 3.20 below we decorate each node X in an m-tree with *assignments* that use the messages indexed by X to fill the variables in a join pattern. But first, we need some auxiliary constructions.

Given a mailbox $\mathcal{M}$ and a join pattern J if γ with $J = \mu_1 \wedge \ldots \wedge \mu_p$, we define the function $\mathsf{c} : \{1,\ldots,p\} \to 2^{\{1,\ldots,|\mathcal{M}|\}}$ that maps each $i \in \{1,\ldots,p\}$ to the set of indexes of messages in $\mathcal{M}$ that match μ_i; formally,

$$\mathsf{c}(i) \;=\; \left\{ j \in 1..|\mathcal{M}| \;\middle|\; \text{there is a substitution } \sigma \text{ such that } \mu_i\sigma = \mathcal{M}[j] \right\} \tag{1}$$

Also, an *$\mathcal{M}$-assignment for J* is an injective map $\mathsf{a} : \{1,\ldots,p\} \to \{1,\ldots,|\mathcal{M}|\}$ such that $i \in \mathsf{c}(i)$ for all $1 \le i \le p$. Let $\mathsf{asgn}(\mathcal{M}, J)$ be the set of all $\mathcal{M}$-assignments for J. (Note that $\mathsf{asgn}(\mathcal{M}, J) = \emptyset$ if $|\mathcal{M}| < p$.) The next Proposition 3.18 ensures that each assignment has a unique substitution induced by the matching; such a substitution can be effectively computed since the proof of Proposition 3.18 is constructive.

▶ **Proposition 3.18.** *For each* $\mathsf{a} \in \mathsf{asgn}(\mathcal{M}, \mu_1 \wedge \ldots \wedge \mu_p)$ *there is a unique substitution* σ_a *such that* $\mathcal{M}[\mathsf{a}(i)] = \mu_i\sigma_\mathsf{a}$*, for all* $i \in \{1,\ldots,p\}$*.*

Proof. Since each variable occurs at most once in $\mu_1 \wedge \ldots \wedge \mu_p$ (by well-formedness, Def. 3.1), it suffices to take $\sigma_\mathsf{a} = \bigcup_{i\in\{1,\ldots,p\}} \sigma_i$ where $\mathcal{M}[\mathsf{a}(i)] = \mu_i\sigma_i$ for all $i \in \{1,\ldots,p\}$. ◀

An assignment $\mathsf{a} \in \mathsf{asgn}(\mathcal{M}, J)$ is *valid for the guard* γ if $\gamma\sigma_\mathsf{a}$ evaluates to true (with σ_a from Proposition 3.18).

▶ **Example 3.19** (Assignments). Let Π_1 and $\mathcal{M}$ as in Example 3.13. We have $c(1) = \{2,3,4\}$ and $c(2) = \{1\}$, the assignment a such that $\mathsf{a}(1) = 4$ and $\mathsf{a}(2) = 1$ belongs to $\mathsf{asgn}(\mathcal{M}, \mathtt{Fault}(id_1, \_) \wedge \mathtt{Fix}(id_2))$ and it is valid for the guard $id_1 = id_2$ while for $\mathsf{a}[3/1]$ (the update of a mapping 1 to 3) we have $\mathsf{a}[1 \mapsto 3] \in \mathsf{asgn}(\mathcal{M}, \mathtt{Fault}(id_1, \_) \wedge \mathtt{Fix}(id_2))$ and $\mathsf{a}[1 \mapsto 3]$ is not valid for $id_1 = id_2$. ⌟

We are now ready to introduce *assignment trees*.

▶ **Definition 3.20** (Assignment trees, pattern resolution). *The* assignment tree of a join pattern J if γ w.r.t. mailbox $\mathcal{M}$ *is the pair* $(T, \mathfrak{a})$ *where, letting* $I = \mathsf{c}(\{1,\ldots,p\})$ *with* c *as in* (1)*,*

- $T = (N, E)$ *is the subtree up-to level* p *of* $\mathsf{r}((\{\emptyset\}, \emptyset), I)$*, the m-tree on* I *and*
- *the map of* candidate assignments $\mathfrak{a} : N \to 2^{\mathsf{asgn}(\mathcal{M},J)}$ *is such that, for each node* $X \in N$*,*

$$\mathfrak{a}(X) \;=\; \{\mathsf{a} \in \mathsf{asgn}(\mathcal{M}, J) \mid \mathsf{a} \textit{ is valid for } \gamma \textit{ and } \operatorname{cod}\mathsf{a} = X\}$$

Let $\mathsf{t}(\mathcal{M}, J \text{ if } \gamma)$ *denote the assignment tree of* J if γ *w.r.t* $\mathcal{M}$*. Pattern* J is resolved in $\mathcal{M}$ *if there is a leaf* X *in* $\mathsf{t}(\mathcal{M}, J \text{ if } \gamma)$ *such that* $\mathfrak{a}(X) \neq \emptyset$*.*

Proposition 3.21 below is a soundess result: the assignment tree for mailbox $\mathcal{M}$ and join pattern Π only contains combinations of message indexes from $\mathcal{M}$ that can contribute to matching Π. Instead, Theorem 3.22 is a completeness result: it says that if an assignment matches a join pattern Π for mailbox $\mathcal{M}$, then the m-tree of $\mathcal{M}$ contains a node made of exactly the messages used by that assignement. Taken together, these two results guarantee that, if we inspect assignment trees to find possible matches for Π in mailbox $\mathcal{M}$, we can only find possible matches (soundness), and we will not miss any possible match (completeness).

▶ **Proposition 3.21.** *The assignment tree* $\mathsf{t}(\mathcal{M}, \Pi)$ *is consistent.*

Proof. The union of nodes of the same level but 0 yields $\mathsf{c}(\{1, \ldots, p\})$. ◀

▶ **Theorem 3.22.** *For all* $\mathsf{a} \in \mathsf{asgn}(\mathcal{M}, J)$, $\operatorname{cod} \mathsf{a} \in \mathsf{t}(\mathcal{M}, J \text{ if } \gamma)$.

Proof. Assume $J = \mu_1 \wedge \ldots \wedge \mu_p$ and proceed by induction on p using Def. 3.16. ◀

We now need to rank the assignments in an m-tree to align with our "fair matching" policy (Def. 3.8). To this end, we define the total order $\preceq$ on $\mathsf{asgn}(\mathcal{M}, J)$ as follows:

$$\mathsf{a} \preceq \mathsf{a}' \quad \text{if} \quad \langle \mathsf{a}(1) \cdot \ldots \cdot \mathsf{a}(p) \rangle \leq_{\mathsf{lex}} \langle \mathsf{a}'(1) \cdot \ldots \cdot \mathsf{a}'(p) \rangle \qquad \text{where} \qquad J = \mu_1 \wedge \ldots \wedge \mu_p \tag{2}$$

Now, Def. 3.23 below formalises how a join pattern is "fairly" resolved in an assignment tree. Observe that, crucially, Def. 3.23 only considers the first node in a depth-first traversal of the assignment tree that yields some candidate assignments. This allows our algorithm to find the fairest matches first, and avoid unnecessary traversals.

▶ **Definition 3.23** (Fair resolution). *Let* X *be the first node in a depth-first visit of the assignment tree* $\mathsf{t}(\mathcal{M}, J \text{ if } \gamma)$ *at level* p *whose candidate assignment map* $\mathfrak{a}$ *is non-empty. The* fair resolution of $\mathsf{t}(\mathcal{M}, J \text{ if } \gamma)$ *is the minimal assignment in* $\mathfrak{a}(X)$ *w.r.t pre-order* $\preceq$ *in* (2).

Note that Def. 3.23 univocally identifies a fair resolution when a join pattern matches multiple slices, as shown in Example 3.24 below.

▶ **Example 3.24.** Let Π_1 be as in Example 3.5 and consider the mailbox:

$$\mathtt{Fault}_1(3, 10{:}35) \cdot \mathtt{Fault}_2(2, 10{:}39) \cdot \mathtt{Fault}_3(3, 10{:}56) \cdot \mathtt{Fix}_4(3)$$

The assignments $\mathsf{a} = \begin{cases} 1 \mapsto 1 \\ 2 \mapsto 4 \end{cases}$ and $\mathsf{a}' = \begin{cases} 1 \mapsto 3 \\ 2 \mapsto 4 \end{cases}$ are valid (observe that $\sigma_{\mathsf{a}} \neq \sigma_{\mathsf{a}'}$), and their fair resolution is a, since $\mathsf{a} \preceq \mathsf{a}'$. ⌟

Finally, Theorem 3.25 below shows that the tree-based algorithm correctly computes the fair join pattern matching according to Def. 3.8.

▶ **Theorem 3.25.** *Let* $\Pi = J \text{ if } \gamma$ *with* $J = \mu_1 \wedge \ldots \wedge \mu_p$, *then* $\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}$ *if and only if the fair resolution* a *of* $\mathsf{t}(\mathcal{M}, \Pi)$ *is such that* $\mathcal{I} = \mathsf{a}(1) \cdot \ldots \cdot \mathsf{a}(p)$.

Proof. ($\Longrightarrow$) Let $\mathcal{M}[\mathcal{I}] = m_1 \cdot \ldots \cdot m_p$. By Def. 3.8, there is substitution σ such that $\gamma\sigma$ holds and $\mu_i\sigma = m_i$ for all $i \in \{1, \ldots, p\}$ and any other slice with this property is greater than $\mathcal{I}$. Let a be the assignment such that $\mathsf{a}(i) = \mathcal{I}[i]$ for all $i \in \{1, \ldots, p\}$. By construction, $\mathsf{a} \in \mathsf{asgn}(\mathcal{M}, J)$, hence $\operatorname{cod} \mathsf{a} \in \mathsf{t}(\mathcal{M}, J \text{ if } \gamma)$ by Theorem 3.22. Let a' be the resolution of $\mathsf{t}(\mathcal{M}, J \text{ if } \gamma)$. By definition $\mathsf{a}' \preceq \mathsf{a}$ which, by (2), is equivalent to say that $\langle \mathsf{a}'(1) \cdot \ldots \cdot \mathsf{a}'(p) \rangle \leq_{\mathsf{lex}} \langle \mathsf{a}(1) \cdot \ldots \cdot \mathsf{a}(p) \rangle$. We then have the thesis since $\mathsf{a} = \mathsf{a}'$ because we also have $\langle \mathsf{a}(1) \cdot \ldots \cdot \mathsf{a}(p) \rangle \leq_{\mathsf{lex}} \langle \mathsf{a}'(1) \cdot \ldots \cdot \mathsf{a}'(p) \rangle$ by hypothesis.

($\Longleftarrow$) Let $\mathcal{I}'$ be such that $\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}'$, map a be the resolution of $\mathsf{t}(\mathcal{M}, \Pi)$, and $\mathcal{I} = [\mathsf{a}(1) \cdot \ldots \cdot \mathsf{a}(p)]$. We have $\mathcal{M} \models_{\mathcal{I}} \Pi$ since $\mathcal{M}[\mathcal{I}] \models_{\sigma_{\mathsf{a}}} \Pi$ by construction. Therefore $\mathcal{I}' \leq_{\mathsf{lex}} \mathcal{I}$ by Def. 3.8 and the codomain of the assignment a' such that $\mathsf{a}'(i) = \mathcal{I}'[i]$ for all $i \in \{1, \ldots, p\}$ belongs to $\mathsf{t}(\mathcal{M}, \Pi)$ by Theorem 3.22. Let $Y \in \mathsf{t}(\mathcal{M}, \Pi)$ containing this codomain and $X \in \mathsf{t}(\mathcal{M}, \Pi)$ be such that $\operatorname{cod} \mathsf{a} \in X$. We have the following cases: either X precedes Y, or Y precedes X, or else $X = Y$. In the first case, if $\max X < \max Y$ then $\mathcal{I}' \neq \mathcal{I}$ and $\mathcal{I}' \leq_{\mathsf{lex}} \mathcal{I}$, which contradicts the hypothesis $\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}$. In the second case, if $\max Y < \max X$ then $\mathcal{I}' \neq \mathcal{I}$ and $\mathcal{I} \leq_{\mathsf{lex}} \mathcal{I}'$, which contradicts the fact that a' is the resolution of $\mathsf{t}(\mathcal{M}, \Pi)$. It must therefore be $X = Y$ which implies $\mathsf{a}' = \mathsf{a}$. ◀

4 Implementation: the `JoinActors` Library

We now present `JoinActors`, our actor-based implementation of join patterns and fair matching algorithms in the Scala 3 programming language. `JoinActors` is the companion artifact of this paper, and its latest version is available at:

`https://github.com/a-y-man/join-actors`

In Section 4.1 we provide an overview of the join patterns API, and the main components of the library and motivation behind the choice of Scala 3. In Section 4.2 we present the implementation of the stateful tree-based matching, and in Section 4.3 we describe our prototype actor framework with fair join pattern matching.

4.1 Overview

The API of `JoinActors` allows programmers to write join patterns using (almost-)standard Scala pattern-matching syntax (as shown in the code snippet in Listing 1); at compile-time, the pattern matching code is transformed (using metaprogramming) into an internal data structure that is used by the matching algorithms to perform the join pattern matching. To use the library, the programmer calls the `receive` function (which is actually a macro) passing join patterns as regular Scala 3 pattern-matching expressions. This function also take as a parameter the type of matching algorithm to use. The syntax for join definitions is:

```
receive { (self: ActorRef[...]) =>
   case J1 if γ1 => RHS1
   case J2 if γ2 => RHS2    ... }
```

We selected Scala 3 for our join pattern library because its macros allow us to implement a straightforward API for join patterns, without necessitating specialized syntax or compiler extensions. This way, programmers can write join patterns using familiar language constructs, eliminating the need to learn a new language or syntax.

Our library uses a `Matcher` trait as a common interface to two matching algorithms:

- `BruteForceMatcher`: the brute-force matching algorithm; this is a translation of the declarative semantics with no state management, described in Section 3.2;
- `StatefulTreeMatcher`: the stateful tree-based matching algorithm described in Section 3.3.

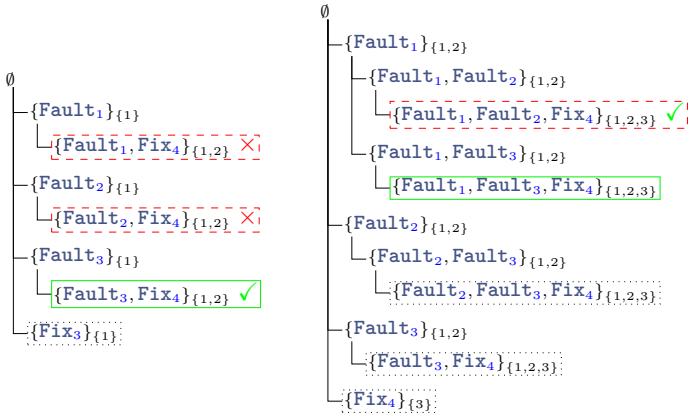
4.2 Implementing Stateful Tree-based Matching

M-trees (Def. 3.15) are the basic data structure of the our algorithm which are the cornerstone to build assignment trees (Def. 3.20). Given a join pattern $\Pi = J \text{ if } \gamma$ with $J = \mu_1 \wedge \ldots \wedge \mu_p$ and a mailbox $\mathcal{M}$, our implementation *lazily* builds $\mathtt{t}(\mathcal{M}, \Pi)$ using the ramification operation (Def. 3.16) starting from the tree $(\emptyset, \emptyset)$ and incrementally extending it in depth-first order. When the messages of $\mathcal{M}$ indexed by a leaf X at level p complete Π, we check the guard γ:

- if γ is satisfied by an assignment induced by X, we report the match, and remove the matched messages from $\mathcal{M}$ and from the assignment tree (stopping its ramification);
- otherwise, we optimise the tree by pruning the leaf X, and continue its ramification.

If no match is found, we wait for a new message to land in the mailbox, and repeat.

▶ **Example 4.1** (Assignment tree construction). The assignment trees for Π_1, Π_2, and $\mathcal{M}$ in Example 3.5 are pictorially shown below:



For readability we show the messages with their indexes in the nodes; we also decorate the nodes with the positions in the join pattern that match the node contents (e.g., in the leftmost tree, the node $\{\texttt{Fault}_1, \texttt{Fix}_4\}_{\{1,2\}}$ contains messages that fit in the positions 1 and 2 of Π_1).

Leaves yielding a successful match (i.e., a combination of messages that complete the message pattern and satisfy its guard) are marked with green solid box and symbol ✓. Leaves where messages completes the join pattern without satisfying its guard are marked with red dashed boxes and symbol ×; such leaves are immediately pruned from the tree once computed. Leaves in dotted boxes are other *potential* message matches – which are *not* actually computed, because an earlier successful match is found while lazily ramifying the tree (and the earlier match is fairer than the later potential match). ⌟

In the implementation, the M-trees are represented using the `TreeMap`[4] data structure: it is a sorted map that takes an ordering on the keys, and for the ordering we use Def. 3.6. We use the following data structure types:

- `type PatternBins = TreeMap[PatternIdxs, MessageIdxs]` is a map from the positions of the patterns to the indices of the messages that match the pattern. These are the subscripts of the nodes in the trees of Example 4.1, where we associate the index of a message to the index of the pattern matching it. If a join pattern contains several messages with the same constructor (such as `Fault` in Π_2 in Example 4.1) then these messages will be grouped in the same bin where the key is the sequence of indices of the join pattern and the value will be the sequence of indices of the matched messages from the mailbox. For instance, the pattern bin of the leaf node with green solid box $\{\texttt{Fault}_1, \texttt{Fault}_2\texttt{Fix}_4\}_{\{1,2,3\}}$ in the rightmost tree in Example 4.1 would be represented as $[[1,2] \mapsto [1,2], [3] \mapsto [4]]$.
- `type MatchingTree = TreeMap[MessageIdxs, PatternBins]` is a map from the indices of the messages that have been matched so far to the `PatternBins`. Thus, the leaf node with green solid box in the rightmost tree in Example 4.1 would have the matching tree $[1,2,4] \mapsto [[1,2] \mapsto [1,2], [3] \mapsto [4]]$.

Note that these data structures contain only the indices of the partially-matched messages and patterns. The guard is checked only when a leaf node is completed (as described above).

4.3 Prototype Actor Framework

To showcase our join patterns implementation in the actor concurrency model, `JoinActors` offers a prototype actor framework in Scala. Notably, our implementation requires Java 21 or later to run due to the use of *virtual threads*. We use `LinkedTransferQueue`s as the mailbox implementation, and `ActorRef` objects for sending messages into a mailbox.

[4] `https://scala-lang.org/api/3.x/scala/collection/immutable/TreeMap.html`

```
1  class Actor[M, T](private val matcher: Matcher[M, Result[T]]):
2    private val mailbox = LinkedTransferQueue[M]
3    val self            = ActorRef(mailbox)
4
5    def start(): (Future[T], ActorRef[M]) =
6      val promise = Promise[T]
7      ec.execute(() => run(promise))
8      (promise.future, self)
9
10   @tailrec
11   private def run(promise: Promise[T]): Unit =
12     matcher(mailbox)(self) match
13       case Continue    => run(promise)
14       case Stop(value) => promise.success(value)
```

Listing 2 The `Actor` class implementation in the `JoinActors` library.

In Listing 2 (line 1) the actor's constructor takes a `Matcher` instance as a parameter. The `run` method processes the messages in the mailbox using the `matcher` instance built by the `receive` macro, which may be either a `BruteForceMatcher` or a `StatefulTreeMatcher` instance. Depending on the result of the right-hand side of the join pattern, the actor either continues processing messages or stops and returns a result.

5 Evaluation

In this section we present the evaluation of our implementation of join patterns and the matching algorithms. In Section 5.1 we describe the methodology used to evaluate the performance of the algorithms. In Section 5.2 we present the results of the custom synthetic benchmarks. In Section 5.3 we present the results of a smart house benchmark. In Section 5.4 we present the results of a bounded buffer benchmark. In Section 5.5 we analyse the correlation between the size of the actor mailbox and the size of the m-trees. In Section 5.6 we compare our implementation of the tree-based fair matching algorithm with an alternative implementation based on the RETE algorithm. Overall, our experiments show that:

- Our stateful tree-based algorithm outperforms the brute force strategy in "noisy" workloads, where messages forming the fairest match are interspersed with random and non-matching messages. On the contrary, when the messages for the fairest match arrive consecutively, the brute force strategy outperforms the stateful tree-based one, since the latter incurs the overhead of building the matching tree. It is worth noticing that non-noisy workloads are unlikely in distributed or asynchronous scenarios.
- Compared to a RETE-based implementation of fair matching, our tree-based algorithm avoids unnecessary production of matches (as it directly picks the "fairest match") and scales better when guards are computationally heavy. However, our tree-based algorithm implementation is an unoptimised proof-of-concept, and for simple guards may perform worse than an optimised RETE implementation (as the one we used).
- The number of matches per second measured in our experiments also shows that our implementation of the stateful tree-based matching is suitable for both our floor shop opening example and the smart house scenario – where the expected input traffic is in the order of tens of messages per second, with moderate "noise."

5.1 Methodology

The lack of a standard benchmark suite for join patterns makes the performance evaluation non-trivial, as the performance of matching algorithms is highly sensitive to the inputs (i.e. amount and order of incoming messages), the size of the message patterns, and the complexity of the guards. Similar sensitivity has been documented e.g. in [15], which compares the matching algorithms RETE [6] and TREAT [18] in the realm of expert and rule-based multi-agent systems. Given that our stateful tree-based algorithm is influenced by RETE and TREAT, our evaluation methodology is inspired by the assessment conducted in [15].

We have devised a set of benchmarks to draw insights into the performance comparison between our stateful tree-based algorithm and the naïve brute-force algorithm. We also identify the trade-offs between the two algorithms in different scenarios, i.e., the overhead of maintaining state versus the overhead of reprocessing messages. To this end, we have created custom synthetic benchmarks and adapted some benchmarks from the literature, such as a producers-consumers bounded buffer from the Savina benchmark suite [13] and a smart house benchmark adapted from [22]. We ran the experiments on a computer with dual Xeon E5-2687W (8-core, 3.10GHz) and 128GB of memory, with 64-bit GNU/Linux 5.10.27. We used OpenJDK 21 with default settings, maximum heap size set to 16 GB, and Scala 3.3.3.

5.2 Synthetic Benchmarks

The general setup of each benchmark involves a program with a single actor, which consists of precisely one join definition, which receives message sent incrementally without delays. A benchmark execution finishes once the actor has processed all matches for the messages contained in its mailbox. We start measuring time just before the first message is sent and stop when the actor halts, so to disregard the time for setting up the actor. The benchmarks are implemented in Scala 3, and are included in the companion artifact of this paper.

Each experiment is repeated 5 times. The plots in Figs. 1 and 2 are to be read as follows: the x-axis represents the join pattern size (i.e., the number of messages in the pattern), and the error bars show the standard deviation of the measurements; the solid lines (measured on the left y-axis, log scale) show the benchmark completion time; the dashed lines (measured on the right y-axis, log scale) show the number of matches per second.

The benchmark suite has been crafted to encompass the following three aspects.

1. **Pattern size.** We have considered actors with pattern sizes ranging from 1 to 6: this mirrors scenarios found in the literature, where join patterns are usually not very long.
2. **Message workload profile.** We have benchmarked two kinds of input traffic workload: (1) a "clean" workload where the messages delivered to the mailbox precisely match the join pattern; (2) a spectrum of "noisy" workloads, where varying amounts of messages delivered to the mailbox will not match any pattern. The noise is uniformly interspersed with matchable messages. The rationale behind this choice is that the first scenario should favour the brute-force algorithm, as it may minimize the advantages of maintaining state, allowing us to measure the overhead of state maintenance. Conversely, the second scenario will require the brute-force algorithm to analyse unusable combinations of messages, thereby enabling us to measure the benefits of maintaining state.
3. **Guard effect.** We evaluated actors with join patterns, with and without guards. The inclusion of patterns with guards is particularly tailored to the main goal of this paper, which is to assess the benefits of state-based algorithms in the presence of guards.

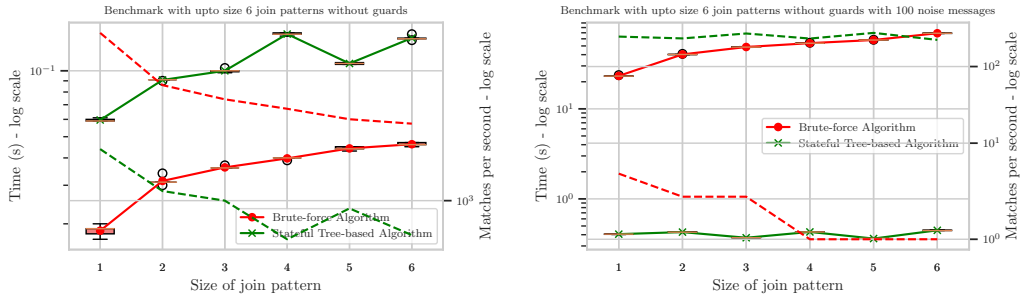


Figure 1 Pattern size without guards benchmark: without noise (left) and with noise (right).

5.2.1 Pattern Size and Workload without Guards

The first group of benchmarks compares the performance of brute-force and tree-based algorithms in cases where actors do not use guards. We consider actors with the following shape, for size 5 (i.e., a unique join pattern matching case for 5 messages). Note that messages have no payload, and the only rule has no guard.

```
Actor[SizeMsg, ...] {
  receive { (_: ActorRef[SizeMsg]) =>
    { case (A(), B(), C(), D(), E()) => ... }
} }
```

The corresponding benchmark evaluates the performance of such actor when fed with a stream of messages consisting on 100 repetitions of the sequence `A()`, `B()`, `C()`, `D()`, `E()`. Note that the mailbox is fed with messages in the exact order required for the match.

The results of the benchmark, considering actors of size 1 to 6, for both algorithms are shown in Fig. 1. The plot on the left of Fig. 1 shows that the brute-force approach significantly outperforms the stateful tree-based approach because the latter has the inherent overhead to build and to update trees – whereas the brute-force algorithm defers processing until a sufficient number of messages are received. Due to the nature of the traffic sent to the actor, the brute-force algorithm immediately finds a match every n messages, where n is the size of the pattern (e.g., 5 in the code snippet above). Instead, the stateful tree-based algorithm has to update its tree for each message, and only after n messages will it find a match and then prune the tree: hence, the retained state is only marginally utilised. However, as shown in the right plot of Fig. 1, if we change the shape of the messages sent to the actor, by augmenting the sequence of messages with noise (i.e. messages that do not match the pattern), the stateful tree-based algorithm outperforms the brute-force algorithm.

5.2.2 Pattern Size and Workload with Guards

The next benchmark addresses the effect of guards. Actors resemble the ones in Section 5.2.1, but now messages have payload and join patterns are augmented with guards, as follows:

```
Actor[SizeMsg, ...] {
  receive { (_: ActorRef[SizeMsg]) => {
    case (A(x), B(y), C(z), D(w), E(a))
        if x == y && y == z && z == w && w == a => ... }
} }
```

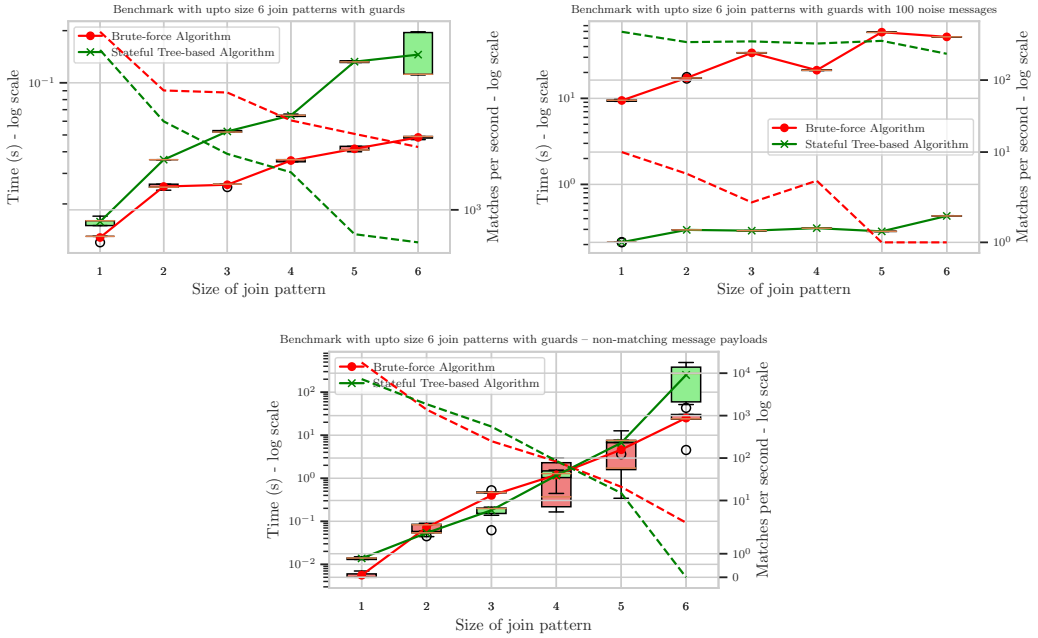


Figure 2 Benchmark results for varying pattern sizes with guards: without noise (top left), and with additional noise messages (top right) that cannot be matched by any join pattern. The bottom plot is a variation where the noise consists of messages that might be potentially matched by the join patterns, but whose payloads do not satisfy their guard.

As before, we first address the case of "perfect" workload, by feeding the actor with sequences of messages as `A(1)`, `B(1)`, `C(1)`, `D(1)`, `E(1)`, `A(2)`, `B(2)`, `C(2)`, `D(2)`, `E(2)`, ...

The results of the benchmark are shown in Fig. 2 (top-left). Similarly to the benchmark without guards, the brute-force algorithm outperforms the stateful tree-based algorithm on well-behaved input traffic. However, when we augment the sequence of "noise" messages, the results are similar to the benchmark without guards. Namely, the stateful tree-based algorithm outperforms the brute-force algorithm. This can be seen in Fig. 2 (top-right).

Moreover, we have performed a variation of this benchmark with a different type of noise: this time, the sequence of messages sent to the actor is augmented with payloads that do *not* satisfy the guard. An example of such a sequence of messages is:

`A(1)`, `A(-3)`, `B(1)`, `B(-4)`, `C(1)`, `C(-5)`, `D(1)`, `D(-6)`, `E(1)`, `E(-7)`, `A(2)`, ...

where only the messages highlighted in green match the pattern.

The benchmark results are shown in Fig. 2 (bottom). The performance of both algorithms is similar, which aligns with our expectations. Noise messages will always be reprocessed by both algorithms, as they persist as partial matches in the m-tree and as unprocessed messages in the brute-force algorithm. These noise messages can only be discarded if they satisfy the guard condition, which is not the case in this benchmark.

5.3 Smart House Benchmark

This benchmark is a real-world scenario adapted from [22]. In this setup, a single actor represents the smart house monitor and controller, tasked with managing various smart devices within a household. Specifically, the actor (1) activates the lights when someone enters and the ambient light is below 40 lux, and (2) detects arrivals or departures based on specific message sequences and reacts accordingly. The actor is shown in Listing 3.

```
Actor[Action, ...] {
  receive { (selfRef: ActorRef[Action]) =>
    case (
          Motion(_: Int, mStatus: Boolean, mRoom: String, t0: Date),
          AmbientLight(_: Int, value: Int, alRoom: String, t1: Date),
          Light(_: Int, lStatus: Boolean, lRoom: String, t2: Date)
         )  if bathroomOccupied(...) => ...
    case (
          Motion(_: Int, mStatus0: Boolean, mRoom0: String, t0: Date),
          Contact(_: Int, cStatus: Boolean, cRoom: String, t1: Date),
          Motion(_: Int, mStatus1: Boolean, mRoom1: String, t2: Date)
         )  if occupiedHome(...) => ...
    case (
          Motion(_: Int, mStatus0: Boolean, mRoom0: String, t0: Date),
          Contact(_: Int, cStatus: Boolean, cRoom: String, t1: Date),
          Motion(_: Int, mStatus1: Boolean, mRoom1: String, t2: Date)
         )  if emptyHome(...) => ...
    ...
} }
```

Listing 3 Smart house actor (arguments of predicates in the guards are omitted for readability). The type annotations in the join patterns are not necessary, but they are included for clarity.

Each of the mentioned scenarios is implemented as a separate join pattern with guards to ensure the correct behaviour is triggered. The performance evaluation of the implementation is conducted by measuring the time taken to process a number of messages that activate the patterns 1000 times, interspersed with a number of random messages intended to mimic various real-world workloads. When the number of random messages is 0, the smart house actor receives only exact matches: thus, for a join pattern size of 3, the actor will process 3000 such messages to get 1000 matches. Instead, in the case of 16 random messages, the actor receives 1000 sequences of messages, each one consisting of 16 random messages plus 3 matching messages distributed throughout the sequence. The benchmark concludes once the smart house actor has performed the expected 1000 matches.

Fig. 3 shows the results of our experiments: our implementation of the stateful tree-based algorithm quickly outperforms the naïve brute-force one, as it can quickly reuse partial matches and discard failed matches – whereas the brute-force algorithm has to compute all possible matches for each new incoming message. The plot also shows that the tree-based matching algorithm performs $\sim 10^5$ matches/sec. on non-noisy traffic (i.e., every input message is used in a join pattern match), and degrades to $\sim 10^2$ as more noise is injected in the input traffic, until $\sim$90% of the messages are noise. This suggests that the implementation is suitable for a real-world application where a smart house controller may expect tens of messages per second with some amount of noise.

5.4 Producers-Consumers Bounded Buffer

This benchmark is an example of a multi-process synchronization problem. The benchmark involves producers and consumers represented by actors and a buffer actor. The buffer actor acts as a manager: (1) it monitors whether the data buffer is full or empty; (2) when consumers request work from an empty buffer, they are placed in a queue until work becomes

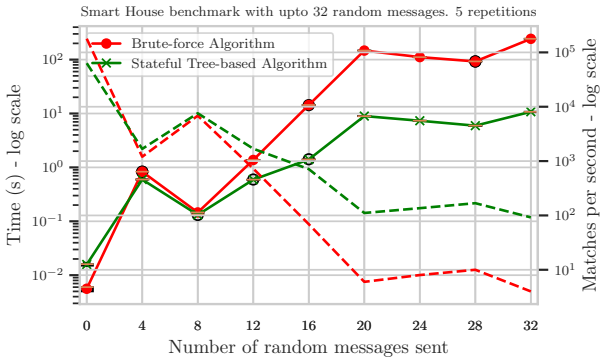


Figure 3 Smart House Benchmark. The x-axis represents the number of random "noise" messages sent with each group of 3 matchable messages; the solid lines (measured on the left y-axis, log scale) show the completion time; the dashed lines (measured on the right y-axis, log scale) show the number of matches per second. For each data point, the smart house actor performs 1000 matches, and the benchmark is repeated 5 times. The error bars show the standard deviation of the measurements.

available; (3) when producers are prepared to produce data but the buffer is full, they are queued until space opens up in the buffer; (4) it alerts producers to generate more work when space becomes available in the data buffer.

Fig. 4 shows the results of the benchmark for a buffer size of 1000. Since the join patterns of the bounded buffer are simple and without any guards and the messages are well-behaved, the brute-force algorithm outperforms the stateful tree-based algorithm. The overhead of maintaining state in the stateful tree-based algorithm is not justified in this case.

5.5 Analysis of Mailbox Size vs. Match Tree Size

We now focus on the relationship between the size of mailboxes and the size of the matching trees maintained in-memory by our stateful matching algorithm. Our analysis uses the smart house benchmark according to three different workload scenarios:

- **No noise**: each batch of 3 incoming messages triggers a match, emptying the mailbox after each match.
- **50% noise**: each batch consists of 3 messages suitable for matching and 3 "noise" messages that cannot be used in the matching, and thus, just accumulate in the mailbox.
- **66% noise**: each batch consists of 3 messages suitable for matching and 6 "noise" ones.

For each scenario, we send 10 batches of messages, triggering 10 join pattern matches. The results are collected in Fig. 5, where the plots in the each row correspond to one of our scenarios. Despite the potential for m-trees to grow exponentially with mailbox size, the plots show a mostly flat and almost-linear correlation between mailbox size and m-tree size. For instance, the left plot in the first row of Fig. 5 illustrates the relationship between the number of messages processed by the actor and both the mailbox and m-tree sizes. Spikes indicate a join pattern match, leading to message removal and m-tree pruning. The right plot in the first row further explores the correlation between mailbox and m-tree sizes, demonstrating, for example, that a mailbox with two messages correlates to an m-tree size of 5-7 nodes, depending on the messages and applicable join patterns.

The results for the noise scenarios are respectively in the second and third row of Fig. 5: they show that mailbox and m-tree sizes decrease after each match similarly to the no-noise scenario. However, "noise" messages accumulate generating partial matches in the m-tree. Flat lines in the m-tree size plot signify unmatchable messages, leaving m-trees unchanged.

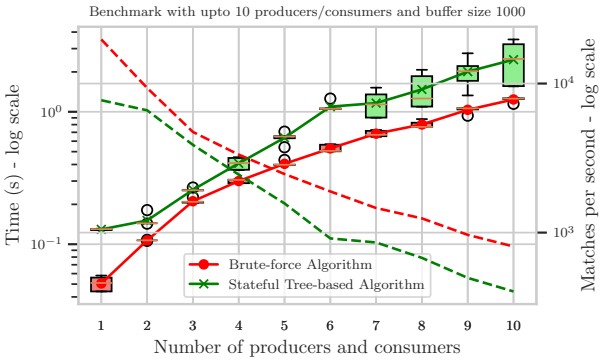


Figure 4 Producers-consumers bounded buffer benchmark: time to produce and consume data in a buffer of size 1000 against the number producers and consumers. The solid lines (measured on the left y-axis, log scale) show the benchmark completion time; the dashed lines (measured on the right y-axis, log scale) show the number of matches per second. The benchmark is repeated 5 times. The error bars show the standard deviation of the measurements.

5.6 Comparison with a RETE-based Fair Matching Implementation

In this section we compare our implementation of our stateful tree-based fair join pattern matching algorithm against an implementation based on the RETE algorithm. The use of RETE takes inspiration from [28], but here we use the Evrete library for Java.[5] To implement an Evrete-based actor that realises our fair matching policy, we proceed as follows.

1. We set up an Evrete *session* where:
   - each incoming message is a modelled as a *fact* with a unique id (denoting the order of arrival), and
   - each pattern matching case is encoded as a *rule* that Evrete will check against all combinations of "message facts" in the session. If a combination of "message facts" satisfies a rule, their message ids are stored in a collection of matches for that rule.
2. We implement an actor (as a JVM virtual thread) that, when a new message arrives:
   a. stores the message as a "message fact" in the aforementioned Evrete session,
   b. fires the session rules, and
   c. if one or more successful matches are produced by any of the rules, then:
      i. sorts the collections of successful matches (if any) by fairness (using Def. 3.6 on the "message fact" ids);
      ii. finds the fairest match;
      iii. removes from the session all the "message facts" used by such a fairest match; and
      iv. clears the collections of successful matches.

A key difference between RETE and our stateful tree-based fair matching algorithm is that RETE exhaustively computes *all* possible matches when the rules are fired, and such matches must be sorted to find the fairest (see item 2(c)i above). Instead, our algorithm only computes matches "on demand" by finding the fairest first, through a lazy depth-first traversal of the match tree. This suggests that our tree-based fair matching algorithm is computationally less expensive than the RETE-based implementation outlined above. However, the comparative benchmarks are also influenced by multiple implementation differences:

[5] `https://www.evrete.org`

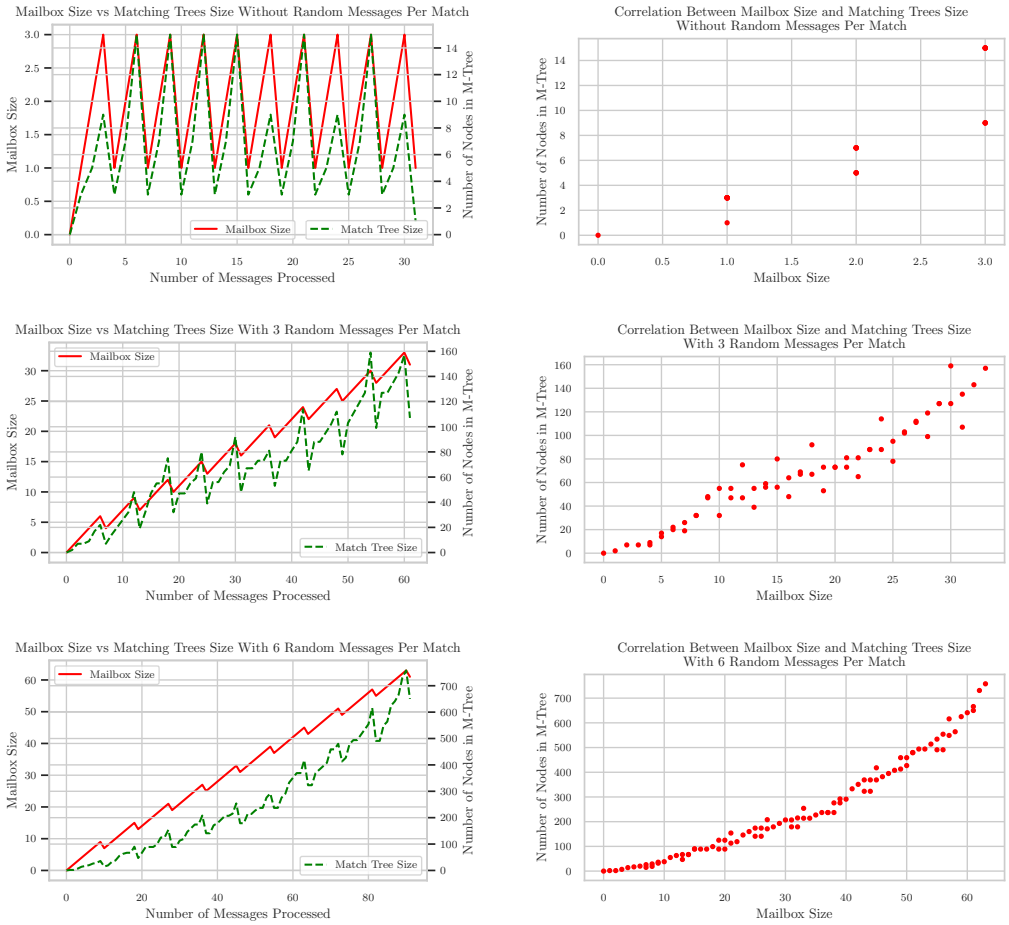


Figure 5 Mailbox size against the size of the matching tree: sizes based on the number of processed messages (left column) and mailbox/tree size correlation (right column).

- our implementation of tree-based fair matching is a proof-of-concept, is not optimised, is single-threaded, and is almost completely written in functional Scala. In particular, adding partial matches to the m-tree is a rather expensive operation, because the m-tree is currently implemented as an immutable data structure;
- instead, Evrete is being developed since 2020 and is significantly optimised, multi-threaded, and written in imperative Java using high-performance mutable data structures.

Consequently, Evrete can produce a significantly higher amount of matches-per-second w.r.t. our implementation – and thus, its exhaustive production of matches can be often faster than our "on-demand, fairest-first" production.

For these reasons, we have designed a benchmark (based on the "smart house" in Section 5.3) that discriminates the computational characteristics of our tree-based fair matching algorithm and the Evrete-based implementation, despite implementation differences:

1. we send n groups of "prefix" messages that create a partial match for a join pattern;
2. then, we send one message that can complete the join pattern with any of the previous "prefix" messages.

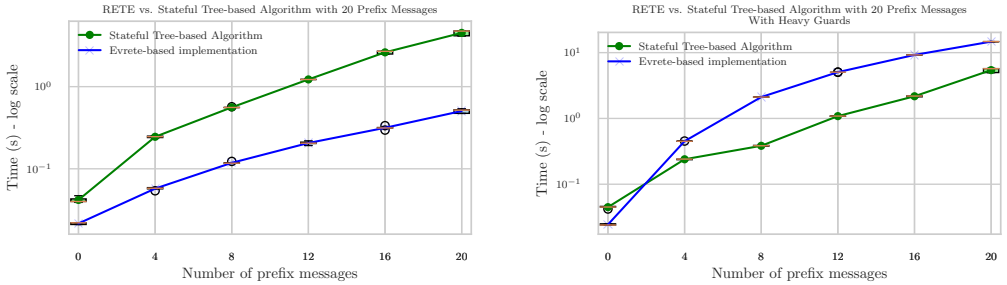


Figure 6 Comparison of tree-based vs. Evrete-based implementation of fair matching. The x-axis shows the number of groups of "prefix" messages sent before a completing message. The y-axis shows the time taken to perform 10 matches.

E.g., for the pattern `Motion`$(\dots) \wedge$ `AmbientLight`$(\dots) \wedge$ `Light`$(\dots)$, we send n times the "prefix" messages `Motion(...), AmbientLight(...)` (which partially match the pattern), and finally we send a message `Light(...)`, which can be combined with any previous "prefix" message to complete the pattern. In this situation, the Evrete-based implementation will compute all possible matches and then find the fairest – while our stateful tree-based fair matching algorithm will immediately produce the fairest match between `Light(...)` and the oldest `Motion(...), AmbientLight(...)` messages. The benchmark measures the time taken to process up to n groups of messages followed by a completing message. In total, we send $(2n+1)\times 10$ messages, thus triggering 10 matches. We perform two variants of this benchmark:

- one with simple guards (the ones used in Section 5.3), and
- one where we artificially slow down the time necessary to evaluate the guards, simulating computationally-intensive "heavy guards" that take ∼0.1 milliseconds to be computed.

The results are shown in Fig. 6. The plot on the left (with "simple" guards) shows that the Evrete-based implementation of fair matching outperforms our stateful tree-based implementation. The plot on the right shows that, with "heavy guards", our stateful tree-based implementation outperforms the Evrete-based one: this is because our tree-based algorithm evaluates the "heavy guards" only once (after finding the fairest match), whereas Evrete computes the "heavy guards" repeatedly, for each possible match contained in the actor mailbox. Observe that the overall execution time of our algorithm in the plots of Fig. 6 does not change significantly. This suggest that our algorithm is less sensitive to "heavy guards" than the Evrete-based implementation.

6 Conclusion

We have addressed the problem of formalising and implementing join pattern matching for actor-based concurrent and distributed systems. We have contributed a novel specification of *fair and deterministic join pattern matching* guaranteeing that the oldest messages in an actor's mailbox are eventually consumed, if allowed by the patterns. We have presented a novel *stateful tree-based join pattern matching algorithm* that avoids wasteful recomputations across matches, and we have proved that our algorithm correctly implements the fair matching specification. We have presented a novel actor library for Scala 3 that implements fair join pattern matching, with both stateless and stateful algorithms. We have presented a systematic benchmark suite for join-pattern-based systems, and we have applied it to evaluate our implementation. We have assessed the performance of our stateful tree-based algorithm in

comparison to the brute-force algorithm and a RETE-based implementation, under various conditions. The findings reveal a performance trade-off: the brute-force algorithm excels when dealing with well-behaved workloads, where maintaining state is an unnecessary overhead, whereas the stateful tree-based algorithm outperforms in scenarios with relative noise in the input messages (which is to be expected in many real-world applications) and complex guards, as evidenced in the smart house benchmark. The synthetic benchmarks in Sections 5.2.1 and 5.2.2 underscore the high sensitivity of the matching algorithms to their workload and guard complexity. These insights should be taken into account when choosing the matching algorithm, depending on the nature of the application and anticipated workload.

Future work. Our specification of fair join pattern matching includes several design decisions that may be fine-tuned depending on the application context. E.g., in some settings it may be better to select the pattern with the longest match, and in some settings it may be useful to match the *newest* messages in a mailbox (e.g., if the input traffic volume is too high for processing every message in real-time). We can specify and implement these alternative policies with minimal adjustments to our definitions, results, and library implementation: we plan to study them, and explore other possibilities. Also, ensuring "fair choice" when multiple join patterns are enabled is another intriguing notion of fairness that would require a significantly different formalisation of matching semantics; we leave this as future work.

We plan to study the problem of *join pattern unreachability*, i.e., whether a pattern will always be preempted by its alternatives. E.g., if a join definition contains the two join patterns $A(x) + A(x) \wedge B(y)$, the former may be always preferred to the latter, or not, depending on the nuances of the matching policy across patterns (e.g., first-match vs. longest-match). We plan to study how to statically verify whether a join pattern is unreachable, and extend our Scala 3 implementation to issue a compile-time warning when this occurs.

Our evaluation shows that selecting the best-performing strategy for join pattern matching is not trivial: depending on the expected input traffic and the complexity of the patterns and guards, stateful matching may be faster than stateless matching, or *vice versa*. Our Scala 3 library `JoinActors` can be easily tweaked to allow programmers choose a specific matching strategy *per pattern*; it could also be extended to switch matching strategy "on the fly" (i.e., between matches), and it could be interesting to study how to automatically switch strategy depending on input traffic observations. We plan to adapt `JoinActors` to let programmers select a suitable matching strategy based on a custom heuristic.

`JoinActors` is a proof-of-concept prototype, and we plan to heavily optimise it – in particular, by using a more efficient mutable data structure to represent m-trees, allowing for faster updates when new messages arrive, and faster traversals for finding the fairest match. The need for such optimisations is highlighted by the results shown in Section 5.6, and we plan to study the internals of the Evrete library to inspire future improvements.

We see our evaluation in Section 5 as a first necessary step towards establishing a standard, comprehensive benchmark suite for existing and future join pattern implementations, in the spirit of Savina [13] for actor implementations. We will study how to further improve our benchmark suite, and we welcome feedback and suggestions from the community.

References

1 Gul A. Agha. ACTORS - a model of concurrent computation in distributed systems. In *MIT Press series in artificial intelligence*, 1986.

2 Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004. doi:10.1145/1018203.1018205.

3 Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. Versatile event correlation with algebraic effects. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018. doi:10.1145/3236762.

4 Mariangiola Dezani, Roland Kuhn, Sam Lindley, and Alceste Scalas. Behavioural Types: Bridging Theory and Practice (Dagstuhl Seminar 21372). *Dagstuhl Reports*, 11(8):52–75, 2022. doi:10.4230/DagRep.11.8.52.

5 Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205–224, 1998. doi:10.1016/S1571-0661(04)00143-4.

6 Charles Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Expert Systems*, pages 324–341, 1991.

7 Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In Johan Jeuring and Simon L. Peyton Jones, editors, *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2002. doi:10.1007/978-3-540-44833-4_5.

8 Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 372–385. ACM Press, 1996. doi:10.1145/237721.237805.

9 Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR'96: Concurrency Theory: 7th International Conference Pisa, Italy, August 26–29, 1996 Proceedings 7*, pages 406–421. Springer, 1996.

10 Rosita Gerbo and Luca Padovani. Concurrent Typestate-Oriented Programming in Java. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 24–34, 2019. doi:10.4204/EPTCS.291.3.

11 Rob Van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4), August 2019. doi:10.1145/3329125.

12 Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In Doug Lea and Gianluigi Zavattaro, editors, *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2008. doi:10.1007/978-3-540-68265-3_9.

13 Shams M. Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2687357.2687368.

14 G. Stewart Von Itzstein and Mark Jasiunas. On implementing high level concurrency in Java. In Amos Omondi and Stanislav Sedukhin, editors, *Advances in Computer Systems Architecture, 8th Asia-Pacific Conference, ACSAC 2003, Aizu-Wakamatsu, Japan, September 23-26, 2003, Proceedings*, volume 2823 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2003. doi:10.1007/978-3-540-39864-6_13.

15 Eryk Lagun. Evaluation and implementation of match algorithms for rule-based multi-agent systems using the example of jadex. *MSc Thesis, University of Hamburg*, 2009. URL: https://swa.informatik.uni-hamburg.de/files/abschlussarbeiten/Diplomarbeit_Eryk_Lagun.pdf.

16 Qin Ma and Luc Maranget. Compiling pattern matching in join-patterns. In *International Conference on Concurrency Theory*, pages 417–431. Springer, 2004. doi:10.1007/978-3-540-28644-8_27.

17 Luc Maranget. Compiling pattern matching to good decision trees. In Eijiro Sumii, editor, *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, pages 35–46. ACM, 2008. doi:10.1145/1411304.1411311.

18 Daniel P. Miranker. *TREAT: a new and efficient match algorithm for AI production systems*. PhD thesis, Columbia University, USA, 1987. UMI Order No. GAX87-10209.

19 Martin Odersky. Functional nets. In *European Symposium on Programming*, pages 1–25. Springer, 2000. doi:10.1007/3-540-46425-5_1.

20 Hubert Plociniczak and Susan Eisenbach. Jerlang: Erlang with joins. In *Coordination Models and Languages: 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings 12*, pages 61–75. Springer, 2010.

21 Humberto Rodriguez Avila. *Orchestration of Actor-Based Languages for Cyber-Physical Systems*. PhD thesis, Vrije Universiteit Brussel, 2021.

22 Humberto Rodríguez-Avila, Joeri De Koster, and Wolfgang De Meuter. Advanced join patterns for the actor model based on CEP techniques. *Art Sci. Eng. Program.*, 5(2):10, 2021. doi:10.22152/programming-journal.org/2021/5/10.

23 Claudio V. Russo. The Joins concurrency library. In Michael Hanus, editor, *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, volume 4354 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2007. doi:10.1007/978-3-540-69611-7_17.

24 Claudio V. Russo. Join patterns for visual basic. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 53–72. ACM, 2008. doi:10.1145/1449764.1449770.

25 Antoine Louis Thibaut Sébert. Join-patterns for the actor model in Scala 3 using macros. Master's thesis, DTU Department of Applied Mathematics and Computer Science, 2022. Available at https://findit.dtu.dk/en/catalog/62f83d3680aa6403a4ccc0ab.

26 Martin Sulzmann, Edmund S. L. Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In Doug Lea and Gianluigi Zavattaro, editors, *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2008. doi:10.1007/978-3-540-68265-3_20.

27 Aaron Joseph Turon and Claudio V. Russo. Scalable join patterns. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 575–594. ACM, 2011. doi:10.1145/2048066.2048111.

28 Louise Van Verre, Humberto Rodríguez-Avila, Jens Nicolay, and Wolfgang De Meuter. Florence: A hybrid logic-functional reactive programming language. *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, 2022.