Prefix-Based Tracing in Message-Passing Concurrency*

Juan José González-Abril and Germán Vidal[0000−0002−1857−6951]

MiST, VRAIN, Universitat Politècnica de València
juagona6@vrain.upv.es,gvidal@dsic.upv.es

Abstract. The execution of concurrent applications typically involves some degree of nondeterminism, mostly due to the relative speeds of concurrent processes. An essential task in state-space exploration techniques for the verification of concurrent programs consists in finding points in an execution where alternative actions are possible. Here, the nondeterministic executions of a program can be represented by a tree-like structure. Given the *trace* of a concrete execution, one first identifies its branching points. Then, a new execution can be steered up to one of these branching points (using, e.g., a *partial* trace), so that an unexplored branch can be considered. From this point on, the execution proceeds nondeterministically, eventually producing a trace of the complete execution as a side-effect, and the process starts again. In this paper, we formalize this operation—partially driving the execution of a program and then producing a trace of the entire execution—, which we call *prefix-based tracing*. It combines ideas from both *record-and-replay* debugging and execution tracing. We introduce a semantics-based formalization of prefix-based tracing in the context of a message-passing concurrent language like Erlang. Furthermore, we also present an implementation of prefix-based tracing by means of a program instrumentation.

1 Introduction

Message-passing concurrency mainly follows the so-called *actor model*. At runtime, concurrent processes can only interact through message sending and receiving, i.e., there is no shared memory. In this work, we further assume that communication is asynchronous and that each process has a local mailbox (a queue), so that each sent message is eventually stored in the target process' mailbox. Moreover, we consider that processes can be dynamically spawned at runtime. In particular, we consider a subset of the programming language Erlang [3] for our developments. We note that, in practice, some Erlang built-in's involve shared-memory concurrency; nevertheless, we will not consider them in this work.

* This work has been partially supported by grant PID2019-104735RB-C41 funded by MCIN/AEI/ 10.13039/501100011033, by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), and by French ANR project DCore ANR-18-CE25-0007.

In the context of a message-passing concurrent language, computations are typically nondeterministic because of the relative speeds of processes. Consider, for instance, three processes, p1, p2, and p3. If p1 and p2 both send a message to process p3, the order in which these messages are received may not be fixed (e.g., when the actions of p1 and p2 are unrelated). In such a case, we say that the messages *race* (for p3). Exploring all alternatives for message races is a key ingredient of state-space exploration techniques like *stateless model checking* [5] or *reachability testing* [13].

In order to identify message races, state-space exploration methods usually consider some kind of execution *trace* (e.g., *interleavings* in [1] or *SYN-sequences* in [13]). An execution trace can be seen as an abstraction of an execution which still contains enough information to identify sources of nondeterminism and, in particular, message races. Every time a race is identified, alternative executions are considered so that all feasible executions are systematically explored.[1] A new execution of the program should be driven in such a way that it reproduces the previous execution up to the point where the race was found (as in *record-and-replay* debugging techniques), then chooses a different message and, from this point on, follows the usual nondeterministic semantics. Furthermore, a trace of the new execution should be eventually produced as a side-effect, so that the process can start again. In the following, we refer to this operation combining replay and tracing as *prefix-based tracing*.

In this work, we formalize the notion of prefix-based tracing in the context of a message-passing concurrent language like Erlang. Despite the fact that prefix-based tracing is ubiquitous in state-space exploration methods, we are not aware of any previous semantics-based formalization. In particular, a similar operation is called *prefix-based replay* in [8], though no formal definition is given. Other approaches, like [2] in the context of stateless model checking of Erlang programs, insert preemptive points in the code and, then, force the program to follow a particular scheduling up to a given point, then proceeding nondeterministically. However, as in [8], no semantics-based formalization is presented.

We note that prefix-based tracing can be seen as a generalization of traditional tracing and replay techniques. In particular, when no input trace is provided, the technique boils down to standard tracing. On the other hand, if the trace of a complete execution is provided, then it behaves as a replay debugger, so that the entire execution follows the given trace. Therefore, both tracing and replay can be seen as particular instances of the notion of prefix-based tracing.

Furthermore, besides the instrumented semantics, we also present an implementation of prefix-based tracing as a *program instrumentation*. In this case, given a program, we produce an instrumented version that is parametric w.r.t. a

[1] In practice, *dynamic partial order reduction* techniques [4] are used to avoid exploring alternative executions which are *causally equivalent* to an already considered execution. Loosely speaking, two executions are causally equivalent if they produce the same outcome no matter if the sequence of actions is different. See, e.g., [11,12] for a formal definition of causal equivalence in the context of the language Erlang.

particular (possibly partial) trace.[2] Then, given a particular trace, the program can be executed in the standard runtime environment so that it follows the actions in this trace and, then, proceeds nondeterministically, eventually producing a trace of the complete execution as a side-effect.

The paper is organized as follows. Section 2 presents a summary of the concurrent features of the considered language and its semantics. Then, Section 3 introduces the notions of trace and log, and formalizes prefix-based tracing using an instrumented semantics. In turn, Section 4 presents the details of an implementation of prefix-based tracing as a program instrumentation. Finally, Section 5 concludes and points out some directions for future work.

2 A Message-Passing Concurrent Language

In this section, we present the semantics of a message-passing concurrent language which can be seen as a subset of the Erlang language [3]. Following [15,10], we consider a layered semantics: an *expression semantics* and a *system semantics*. The expression semantics is essentially a typical *call-by-value* functional semantics defined on *local states*, which include an environment (i.e., a mapping from variables to values), an expression (to be reduced) and a stack; see [6] for more details. Since this is orthogonal to the topics of this paper, we will only introduce the following notation: $ls \xrightarrow{z} ls'$ denotes a reduction step, where ls, ls' are *local* states and z is a label with some information associated to the reduction step.

So-called *local* steps are denoted with the label ι and do not perform any side-effect at the system level. In contrast, the reduction of some—typically concurrent—actions may require a side-effect at the system level. Here, we consider the following *global* actions with side-effects:

- $\mathsf{spawn}(mod, fun, args)$: this expression dynamically creates a new process to evaluate function *fun* (defined in module *mod*) with arguments *args* (a list). E.g., $\mathsf{spawn}(test, client, [S, c1])$ spawns a process that evaluates the expression $client(S, c1)$, where function *client* is defined in module *test*.[3] In the expression semantics, a call to spawn reduces to a fresh identifier, called pid (for p*rocess* id*entifier*), that uniquely identifies the new process. The step is labeled with $\mathsf{spawn}(\kappa, ls_0)$, where ls_0 is the initial local state for the new process and κ is a special variable (a sort of *future*) that will be eventually bound—in the system semantics—to the pid of the spawned process.
- $p \,!\, v$: it sends value v (the *message*) to process p (a pid). The expression reduces to v and eventually stores this value in the mailbox of process p as a side-effect. Sending a message is an asynchronous operation, so the process

[2] Hence, the program is only instrumented once.

[3] As in Erlang, functions and atoms (constants) begin with a lowercase letter while variables start with an uppercase symbol. The language has no user-defined data constructors, but allows the use of *lists*—following the usual Haskell-like notation—and *tuples* of the form $\{e_1, \ldots, e_n\}$, $n \geq 1$ (a polyadic function).

```
main() ->
  S = spawn(bank, [0]),
  spawn(customer, [S]).

bank(B) ->
  receive
    {deposit,A}
       -> bank(B+A);
    {C,{withdraw,A}} when A=<B
       -> C ! {ok,B-A},
          bank(B-A);
    _  -> C ! error, bank(B)
  end.

customer(S) ->
  S ! {deposit,120},
  S ! {deposit,42},
  S ! {self(),{withdraw,100}},
  receive
    {ok,B} -> io:format("Current balance: ~p~n",[B]);
    error -> io:format("Insufficient balance")
  end.
```

Fig. 1: A simple Erlang program.

continues immediately with the evaluation of the next expression. In this case, the step in the expression semantics is labeled with $\mathsf{send}(p, v)$, which suffices for the system semantics to perform the corresponding side-effect.

- $\mathsf{receive}\ p_1 \to e_1; \ldots; p_n \to e_n\ \mathsf{end}$: this expression looks for the *oldest* message in the process mailbox that matches some pattern p_i and, then, continues with the evaluation of e_i. As in Erlang, messages are matched sequentially against the patterns from top to bottom. When no message matches any pattern, execution is *blocked* until a matching message reaches the mailbox of the process. In this case, the step is labeled with $\mathsf{rec}(\kappa, cs)$, where cs are the branches of the receive statement (i.e., $p_1 \to e_1; \ldots; p_n \to e_n$ above). Here, κ will be bound to the expression e_i of the selected branch in the system semantics.
- $\mathsf{self}()$: it reduces to the pid of the current process. Here, the step is labeled with $\mathsf{self}(\kappa)$, so that κ is bound to the pid of the current process in the system semantics.

Example 1. Consider the simple client-server program shown in Figure 1. Here, we consider that the execution starts with the call `main`(). Function `main` then spawns two new processes that will evaluate `bank`(`0`) and `customer`(`S`), respectively, where `S` is the pid of the first process (the *server*).

Function `bank` implements a simple server that takes only two types of requests: {`deposit`, `A`}, to make a deposit of amount `A`, and {`C`, {`withdraw`, `A`}}, to

make a withdraw of amount `A`, where `C` is the pid of the customer that makes the request. For simplicity, we assume that the bank has only one account (that of the customer), which is initialized to zero.

Given a request of the form $\{\texttt{deposit}, \texttt{A}\}$, the server simply performs a recursive call with the updated balance. If the request has the form $\{\texttt{C}, \{\texttt{withdraw}, \texttt{A}\}\}$ and the amount A is less than or equal to the current balance,[4] it sends a message $\{\texttt{ok}, \texttt{B} - \texttt{A}\}$ back to the customer and calls function `bank` with the updated balance. In any other case (denoted with the pattern "_"), the message `error` is sent back to the customer.

The implementation of the customer is very simple. It only performs three requests to the server. Note that the third one simulates a synchronous communication since it suspends the execution until a message from the server is received. Here, the built-in function `format` (module `io`) is used for printing messages.

In the remainder of this paper, a process is denoted as follows:

Definition 1 (process). *A process is denoted by a configuration of the form $\langle p, ls, q\rangle$, where p is the pid (process identifier) of the process, which is unique in a system, ls is the local state and q is the process mailbox (a list).*

A *system* is then defined as a pair $\Gamma; \Pi$, where Γ represents the network (sometimes called the *global mailbox* [10] or the *ether* [16]) and Π is a pool of processes. In the following, we often say "process p" to mean "process with pid p".

The network, Γ, is defined as a set of queues, one per each pair of (not necessarily different) processes. For instance, if we have two processes with pids p_1 and p_2, then Γ will include four queues associated to the pairs (p_1, p_1), (p_1, p_2), (p_2, p_1), and (p_2, p_2), representing all possible communications in the system. We use the notation $\Gamma[(p, p') \mapsto qs]$ either as a condition on Γ or as a modification of Γ, where p, p' are pids and qs is a (possibly empty) queue; for simplicity, we assume that queues are initially empty for each pair of processes. Queues are denoted by (finite) sequences, which are denoted as follows: $a_1, a_2, \ldots, a_n$, $n \geq 0$, where $[\,]$ denotes an empty sequence. Here, $es{+}es'$ denotes the concatenation of sequences es and es'; by abuse, we use the same notation when a sequence has only a single element, i.e., $e_1{+}(e_2, \ldots, e_n) = (e_1, \ldots, e_{n-1}){+}e_n = e_1, \ldots, e_n$.

The second component, Π, is denoted as $\langle p_1, ls_1, q_1\rangle \mid \cdots \mid \langle p_n, ls_n, q_n\rangle$, where "$\mid$" represents an associative and commutative operator. We often denote a *system* as $\Gamma; \langle p, ls, q\rangle \mid \Pi$ to point out that $\langle p, ls, q\rangle$ is an arbitrary process of the pool (thanks to the fact that "$\mid$" is associative and commutative).

The rules of the system semantics can be found in Figure 2. They are similar to the those in [10], with only a few differences:

- The local state is abstracted in our semantics, so that it can be instantiated to Core Erlang (as in [10]) but also to Erlang (as in [6]).

[4] Here, we consider the full syntax for receive statements, $\mathsf{receive}\ p_1\ [\mathsf{when}\ g_1] \rightarrow e_1; \ldots; p_n\ [\mathsf{when}\ g_n] \rightarrow e_n\ \mathsf{end}$, where each branch might have a *guard* g_i that must be evaluated to *true* in order to select this branch.

$$(\mathit{Exit}) \quad \frac{\mathit{final}(ls)}{\Gamma; \langle p, ls, q\rangle \mid \Pi \hookrightarrow \Gamma; \Pi}$$

$$(\mathit{Local}) \quad \frac{ls \xrightarrow{\iota} ls'}{\Gamma; \langle p, ls, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, ls', q\rangle \mid \Pi}$$

$$(\mathit{Self}) \quad \frac{ls \xrightarrow{\mathsf{self}(\kappa)} ls'}{\Gamma; \langle p, ls, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, ls'\{\kappa \mapsto p\}, q\rangle \mid \Pi}$$

$$(\mathit{Spawn}) \quad \frac{ls \xrightarrow{\mathsf{spawn}(\kappa, ls_0)} ls' \text{ and } p' \text{ is a fresh pid}}{\Gamma; \langle p, ls, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, ls'\{\kappa \mapsto p'\}, q\rangle \mid \langle p', ls_0, [\,]\rangle \mid \Pi}$$

$$(\mathit{Send}) \quad \frac{ls \xrightarrow{\mathsf{send}(p', v)} ls'}{\Gamma[(p, p') \mapsto qs]; \langle p, ls, q\rangle \mid \Pi \hookrightarrow \Gamma[(p, p') \mapsto qs{+}v]; \langle p, ls', q\rangle \mid \Pi}$$

$$(\mathit{Deliver}) \quad \frac{}{\Gamma[(p', p) \mapsto v{+}qs]; \langle p, ls, q\rangle \mid \Pi \hookrightarrow \Gamma[(p', p) \mapsto qs]; \langle p, ls, q{+}v\rangle \mid \Pi}$$

$$(\mathit{Receive}) \quad \frac{ls \xrightarrow{\mathsf{rec}(\kappa, cs)} ls' \text{ and } \mathsf{matchrec}(ls', \kappa, cs, q) = (ls'', q')}{\Gamma; \langle p, ls, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, ls'', q'\rangle \mid \Pi}$$

Fig. 2: System semantics

– The network, Γ, is defined as a set of queues, so that the order of the messages between any two given processes can be preserved (while Γ was defined as a set of triples (*sender*, *target*, *message*) in [10] and the order could not be preserved).

Moreover, in contrast to the system semantics in [12], we have process' mailboxes and a rule for message delivery, which are abstracted away in [12], where messages are directly consumed from Γ by receive statements. We note that this is not a limitation of [12] since this work focuses on *replay* (reversible) debugging and the trace of an actual execution is always provided. Therefore, their system semantics needs not implement the actual semantics of the language but may rely on the order of message reception given in the considered trace.

Let us briefly explain the transiton rules of our system semantics (Figure 2):

– Rule *Exit* removes a process from the pool when the local state is *final*, i.e., when the expression to be reduced is a data term. If Γ contains some nonempty queue for (p, p'), where p' is the removed process, these messages will never be delivered (which is coherent with the behavior of Erlang).
– Rule *Local* just updates the local state of the selected process according to a transition of the expression semantics, while rule *Self* binds κ to the pid of the current process.
– Rule *Spawn* updates the local state, binds κ to the pid of the new process and adds a new initial process configuration with local state ls_0 as a side-effect.

- Rule *Send* updates the local state and, moreover, adds a new message to the corresponding queue of the network as a side-effect. For simplicity, we implicitly assume that Γ is extended with a new queue for the pair (p, p') whenever it does not already exist.
- Rule *Deliver* nondeterministically (since Γ might contain several nonempty queues with the same target process p) takes a message from the network and moves it to the corresponding process mailbox.
- Finally, rule *Receive* consumes a message from the process mailbox using the auxiliary function matchrec that takes the local state ls', the *future* κ, the branches of the receive expression cs, and the queue q. It then selects the oldest message in q that matches a branch in cs (if any), and returns a new local state ls'' (where κ is bound to the expression in the selected branch) and a queue q' (where the selected message has been removed).

Note that the tracing semantics has two main sources of nondeterminism: selecting a process to apply a reduction rule, and selecting the message to be delivered from the network (rule *Deliver*). Regarding the first point, one can for instance implement a *round-robin* algorithm that performs a fixed number of transitions (assuming the process is not blocked), then moves to another process, etc. As for the selection of a message to be delivered, there are several possible strategies. For instance, the CauDEr debugger [10,9,6] implements both a user-driven strategy (where the user selects any of the available messages) and a random selection.

Given systems α_0, α_n, we call $\alpha_0 \hookrightarrow^* \alpha_n$ a *derivation*; it is a shorthand for

$$\alpha_0 \hookrightarrow \ldots \hookrightarrow \alpha_n,\ n \geq 0$$

One-step derivations are simply called *transitions*. We use $\delta, \delta', \delta_1, \ldots$ to denote derivations and $t, t', t_1, \ldots$ for transitions. A system α is said *initial* if it has the form $\mathcal{E}; \langle p, ls, [] \rangle$, where $\mathcal{E}$ denotes a network with an empty for (p, p), p is the pid of some initial process and ls is an initial local state containing the expression to be evaluated. In the following, we assume that all derivations start with an initial system.

3 Prefix-Based Tracing Semantics

In this section, we formalize the notion of prefix-based tracing for message-passing concurrent programs. In order to trace a running application, [11] introduces *message tags*, so that one can identify the sender and receiver of each message, even if there are several messages with the same value. To be precise, each message value v is now wrapped in a tuple of the form $\{\ell, v\}$, where ℓ is a message tag which is unique in the considered execution.

Following [7], we consider that an execution *trace* is a mapping from pids to sequences of *terms* denoting global actions (so we often refer to these terms as *actions*). These terms can be seen as an abstraction of the corresponding actions, including only some minimal information (but still enough for our purposes):

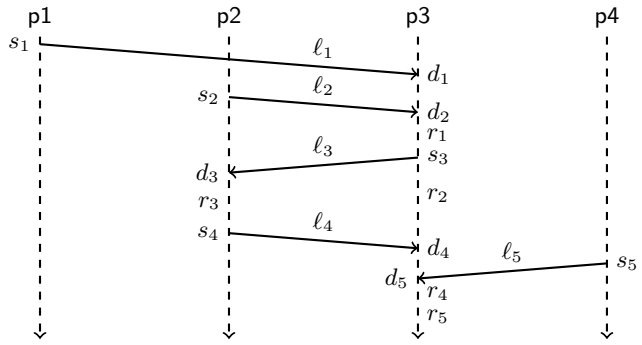

Fig. 3: Processes ($\mathsf{p}i$, $i = 1, \ldots, 4$) are represented as vertical dashed arrows (time flows from top to bottom). Message sending and delivery is represented by solid arrows labeled with a message tag (ℓ_i), from a sending event (s_i) to a delivery event (d_i), $i = 1, \ldots, 5$. Receive events are denoted by r_i, $i = 1, \ldots, 5$. Note that all events associated to a message ℓ_i have the same subscript i.

Definition 2 (trace [7]). *A* trace *is a mapping from pids to sequences of terms of the form*

- $\mathsf{spawn}(p)$*, where* p *is the pid of the spawned process;*
- exit*, which denotes process termination;*
- $\mathsf{send}(\ell, p)$*, where* ℓ *is the tag of the message sent (initially stored in the network) and* p *is the pid of the target process;*
- $\mathsf{deliver}(\ell)$*, where* ℓ *is the tag of the delivered message (i.e., moved from the network to the mailbox of the target process);*
- $\mathsf{rec}(\ell)$*, where* ℓ *is the tag of the message consumed from the local mailbox.*

We note that deliver *events are attributed to the target of the message. Given a trace* $\mathcal{T}$*, we let* $\mathcal{T}(p)$ *denote the sequence of actions associated to process* p *in* $\mathcal{T}$*. Also,* $\mathcal{T}[p \mapsto as]$ *denotes that* $\mathcal{T}$ *is an arbitrary trace such that* $\mathcal{T}(p) = as$*; we use this notation either as a condition on* $\mathcal{T}$ *or as a modification of* $\mathcal{T}$*.*

Example 2. Let us consider the following trace:

$$
\begin{array}{l}
[\ \mathsf{p1} \mapsto \mathsf{spawn(p3)}, \mathsf{spawn(p2)}, \mathsf{spawn(p4)}, \mathsf{send}(\ell_1, \mathsf{p_3}), \mathsf{exit};\\
\ \ \mathsf{p2} \mapsto \mathsf{send}(\ell_2, \mathsf{p_3}), \mathsf{deliver}(\ell_3), \mathsf{rec}(\ell_3), \mathsf{send}(\ell_4, \mathsf{p_3}), \mathsf{exit};\\
\ \ \mathsf{p3} \mapsto \mathsf{deliver}(\ell_1), \mathsf{deliver}(\ell_2), \mathsf{rec}(\ell_1), \mathsf{send}(\ell_3, \mathsf{p_2}), \mathsf{rec}(\ell_2), \qquad (1)\\
\qquad\ \ \mathsf{deliver}(\ell_4), \mathsf{deliver}(\ell_5), \mathsf{rec}(\ell_4), \mathsf{rec}(\ell_5), \mathsf{exit};\\
\ \ \mathsf{p4} \mapsto \mathsf{send}(\ell_5, \mathsf{p_3}), \mathsf{exit} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad]
\end{array}
$$

The associated execution can be informally represented using a simple message-passing diagram, as shown in Figure 3, where we have skipped spawn actions for clarity.

Observe that we do not need to fix a particular (global) interleaving for all the actions in the trace. Only the order within each process matters; i.e., a trace

represents a *partial order* on the possible interleavings (analogously to the SYN-sequences of [13]).

We also consider a simplification of the trace, called *log* in [11,12], where process exit and message delivery actions are skipped and message sending is represented just by $\mathsf{send}(\ell)$, without the pid of the target process.

Definition 3 (log). *A* log *is a mapping from pids to sequences of terms of the form* $\mathsf{spawn}(p)$*,* $\mathsf{send}(\ell)$*, and* $\mathsf{rec}(\ell)$*, where* p *is a pid and* ℓ *is a message tag. We use the same notation conventions as for traces. Moreover, given a trace* $\mathcal{T}$*, we let* $log(\mathcal{T})$ *be the log,* $\mathcal{W}$*, obtained from* $\mathcal{T}$ *by removing message delivery and exit actions, as well as by replacing every action of the form* $\mathsf{send}(\ell, p)$ *by* $\mathsf{send}(\ell)$*.*

For instance, the log obtained from the trace in Example 2 above is as follows:

$$\begin{array}{ll} [\ \mathsf{p1} \mapsto \mathsf{spawn}(\mathsf{p3}), \mathsf{spawn}(\mathsf{p2}), \mathsf{spawn}(\mathsf{p4}), \mathsf{send}(\ell_1); \\ \ \ \mathsf{p2} \mapsto \mathsf{send}(\ell_2), \mathsf{rec}(\ell_3), \mathsf{send}(\ell_4); \\ \ \ \mathsf{p3} \mapsto \mathsf{rec}(\ell_1), \mathsf{send}(\ell_3), \mathsf{rec}(\ell_2), \mathsf{rec}(\ell_4), \mathsf{rec}(\ell_5); \\ \ \ \mathsf{p4} \mapsto \mathsf{send}(\ell_5) &] \end{array} \qquad (2)$$

Despite the simplification, the resulting log suffices to replay a given execution [12, Theorem 4.22] or a *causally equivalent* one.[5] Therefore, in the following, we distinguish *logs*, which are useful to replay a given execution, and *traces*, which can be used, e.g., to identify message races (as in [7]).

Consider, for instance, the execution of Figure 3. Here, we might have a race for p3 between messages ℓ_1 and ℓ_2 (assuming both messages match the constraints of the receive statement r_1). If we swap the delivery of these messages, we can have a new execution which is not causally equivalent to the previous one and, thus, may give rise to a different outcome. A similar situation occurs with messages ℓ_4 and ℓ_5. See [7] for more details on the computation of message races. A typical state-space exploration method would follow these steps:

- First, one considers a random execution of the program and its associated trace.
- The trace is then analyzed and its message races are identified (if any).
- For each message race, we construct a (partial) log that can be used to drive the execution of the program to an execution point where a different choice is made. Then, execution continues nondeterministically, eventually producing a trace of the entire execution. We call this operation *prefix-based tracing*.

[5] We say that two actions are *causally* related when one action cannot happen without the other, e.g., message sending and receiving, spawning a process and any action of this process, etc. Causality is often defined as the transitive closure of the above relation. When two actions are not causally related, we say that they are *independent*. Two executions are *causally equivalent* if they only differ in the order of independent actions. Equivalently, two executions are causally equivalent if they have the same log [12]. Actually, logs can be seen as a representation of so-called *Mazurkiewicz traces* [14]. We refer the interested reader to [12] for more details.

– The process starts again with the new executions, and so forth. Typically, some backtracking algorithm is used in order to avoid considering the same execution (or a causally equivalent one) once and again.

For instance, given the execution of Figure 3 and the associated trace in (1), we have a race for $\mathsf{p3}$ between messages ℓ_1 and ℓ_2. Here, the following partial log could be used to drive the execution to a different choice, where message ℓ_2 is delivered to process $\mathsf{p3}$ before message ℓ_1:

$$\begin{array}{l}[\ \mathsf{p1} \mapsto \mathsf{spawn}(\mathsf{p3}), \mathsf{spawn}(\mathsf{p2}), \mathsf{spawn}(\mathsf{p4}), \mathsf{send}(\ell_1);\\ \quad \mathsf{p2} \mapsto \mathsf{send}(\ell_2);\\ \quad \mathsf{p3} \mapsto \mathsf{rec}(\ell_2);\\ \quad \mathsf{p4} \mapsto \mathsf{send}(\ell_5) \qquad\qquad\qquad\qquad\qquad]\end{array} \tag{3}$$

In the instrumented semantics, a *logged* system is now denoted by a triple $\mathcal{W}; \Gamma; \Pi$, where $\mathcal{W}$ is a (possibly partial) log. We will simply speak of *systems* when no confusion can arise between logged and non-logged systems. Furthermore, we also need some auxiliary functions. In prefix-based tracing, some steps might be driven by a log while others might not (e.g., when all the actions of a process have been already *consumed*). In order to deal with these two situations in a uniform way, we introduce the following function next:

$$\mathsf{next}(p, \mathcal{W}) = \begin{cases} (p', \mathcal{W}) & \text{if } \mathcal{W}(p) = [\,] \text{ and } p' \text{ is a fresh identifier} \\ (p', \mathcal{W}[p \mapsto as]) & \text{if } \mathcal{W}(p) = \mathsf{spawn}(p') {+} as \\ (\ell, \mathcal{W}[p \mapsto as]) & \text{if } \mathcal{W}(p) = \mathsf{send}(\ell) {+} as \\ (\ell, \mathcal{W}[p \mapsto as]) & \text{if } \mathcal{W}(p) = \mathsf{rec}(\ell) {+} as \end{cases}$$

Essentially, $\mathsf{next}(p, \mathcal{W})$ either *consumes* the first action of $\mathcal{W}(p)$ and returns the corresponding pid p' (if the first action is $\mathsf{spawn}(p')$) or a message tag ℓ (if the first action is $\mathsf{send}(\ell)$ or $\mathsf{rec}(\ell)$), or returns fresh identifiers when $\mathcal{W}(p)$ is empty. It also returns the log resulting from removing the consumed action (if any). Here, we consider that pids and tags belong to the same domain for simplicity; otherwise, one would need two different functions, $\mathsf{next\_pid}$ and $\mathsf{next\_tag}$, depending on the particular case.

Our second function, $\mathsf{admissible}$, is used to check if delivering a message is consistent with the current system. Note that message delivery is in principle a nondeterministic operation in the standard semantics (Figure 2) when we have messages in different queues of Γ addressed to the same target process. On the other hand, once messages are delivered, the order is fixed and the receive statements will consume them in a deterministic manner. Therefore, we should ensure that message deliveries follow the corresponding log. For this purpose, we introduce the auxiliary function $\mathsf{admissible}$. Given a log $\mathcal{W}$, if $\mathcal{W}(p)$ is not empty, we have $\mathsf{admissible}(p, \mathcal{W}[p \mapsto as], q, \ell) = \mathit{true}$ if $\mathsf{rec}(\ell_1), \ldots, \mathsf{rec}(\ell_n)$, $n > 0$, are the receive actions in as, q contains messages $\ell_1, \ldots, \ell_i$, $0 \leq i < n$, and $\ell = \ell_{i+1}$. When $\log(p)$ is empty or contains no rec actions, function $\mathsf{admissible}$ simply returns *true* in order to proceed nondeterministically as in the standard semantics. Otherwise, it compares the list of messages to be received by p and

$$(\textit{Exit})\quad \frac{\textit{final}(ls)}{\mathcal{W};\Gamma;\langle p, ls, q\rangle \mid \Pi \rightsquigarrow_{p:\mathsf{exit}} \mathcal{W};\Gamma;\Pi}$$

$$(\textit{Local})\quad \frac{ls \xrightarrow{\iota} ls'}{\mathcal{W};\Gamma;\langle p, ls, q\rangle \mid \Pi \rightsquigarrow_{\epsilon} \mathcal{W},\Gamma;\langle p, ls', q\rangle \mid \Pi}$$

$$(\textit{Self})\quad \frac{ls \xrightarrow{\mathsf{self}(\kappa)} ls'}{\mathcal{W};\Gamma;\langle p, ls, q\rangle \mid \Pi \rightsquigarrow_{\epsilon} \mathcal{W};\Gamma;\langle p, ls'\{\kappa \mapsto p\}, q\rangle \mid \Pi}$$

$$(\textit{Spawn})\quad \frac{ls \xrightarrow{\mathsf{spawn}(\kappa, ls_0)} ls' \text{ and } \mathsf{next}(p,\mathcal{W}) = (p',\mathcal{W}')}{\mathcal{W};\Gamma;\langle p, ls, q\rangle \mid \Pi \rightsquigarrow_{p:\mathsf{spawn}(p')} \mathcal{W}';\Gamma;\langle p, ls'\{\kappa \mapsto p'\}, q\rangle \mid \langle p', ls_0, [\,]\rangle \mid \Pi}$$

$$(\textit{Send})\quad \frac{ls \xrightarrow{\mathsf{send}(p',v)} ls' \text{ and } \mathsf{next}(p,\mathcal{W}) = (\ell,\mathcal{W}')}{\mathcal{W};\Gamma[(p,p') \mapsto qs];\langle p, ls, q\rangle \mid \Pi \rightsquigarrow_{p:\mathsf{send}(\ell,p')} \mathcal{W}';\Gamma[(p,p') \mapsto qs{+}\{v,\ell\}];\langle p, ls', q\rangle \mid \Pi}$$

$$(\textit{Deliver})\quad \frac{\mathsf{admissible}(p,\mathcal{W},q,\ell) = \textit{true}}{\mathcal{W};\Gamma[(p',p) \mapsto \{v,\ell\}{+}vs];\langle p, ls, q\rangle \mid \Pi \rightsquigarrow_{p:\mathsf{deliver}(\ell)} \mathcal{W};\Gamma[(p',p) \mapsto vs];\langle p, ls, q{+}\{v,\ell\}\rangle \mid \Pi}$$

$$(\textit{Receive})\quad \frac{ls \xrightarrow{\mathsf{rec}(\kappa, cs)} ls' \quad \mathsf{matchrec}(ls', \kappa, cs, q) = (ls'', q', \ell) \text{ and } \mathsf{next}(p,\mathcal{W}) = (\ell,\mathcal{W}')}{\mathcal{W};\Gamma;\langle p, ls, q\rangle \mid \Pi \rightsquigarrow_{p:\mathsf{rec}(\ell)} \mathcal{W}';\Gamma;\langle p, ls'', q'\rangle \mid \Pi}$$

Fig. 4: Prefix-based tracing semantics

the list of messages already in p's mailbox in order to determine if ℓ is indeed the next message that must be delivered in order to follow the order of message receptions given by the log.

The instrumented semantics is defined by means of the labeled transition system shown in Figure 4. Now, each transition is labeled with an *event* of the form $p : a$ where p is the pid of a process and a is the action performed by this process. Let us briefly explain the transition rules:

- The first three rules, *Exit*, *Local* and *Self* are similar to their counterpart in the standard semantics (Figure 2), since the log plays no role in these cases. The only relevant difference is that we label the transition with the corresponding action, $p : \mathsf{exit}$, in the first rule, and ϵ (a null event) in the other two rules.
- Rules *Spawn* and *Send* proceed in a similar way: when $\mathcal{W}(p)$ is not empty, the pid of the new process (rule *Spawn*) or the message tag (rule *Send*) are taken from the log. Otherwise, fresh identifiers are used, as in the standard semantics of Figure 2. The transitions are labeled with the events $p : \mathsf{spawn}(p')$ and $p : \mathsf{send}(\ell, p')$, respectively.
- Rule *Deliver* ensures that messages are delivered according to the order in $\mathcal{W}(p)$. Observe that, given a process p, the order of message deliveries is now

deterministic when $\mathcal{W}(p)$ is not empty and includes at least one rec action. Here, the transition is labeled with the event $p{:}\mathsf{deliver}(\ell)$.
- Finally, rule *Receive* is similar to its counterpart in Figure 2, with only a subtle difference: now, function matchrec also returns the tag of the selected message, since it is required for the label of the transition, $p{:}\mathsf{rec}(\ell)$. We note that function next is only used to consume an action from the log (when $\mathcal{W}(p)$ is not empty) but it imposes no actual restriction on the transition, since once the messages are in the process queue, message reception becomes deterministic. This is why function admissible checks the log in order to deliver messages in the right order.

Given a sequence of events $es = (p_1 : a_1, p_2 : a_2, \ldots, p_n : a_n)$, we let $\mathsf{actions}(p, es)$ denote the sequence of actions $a'_1, a'_2, \ldots, a'_m$ such that $p{:}a'_1, p{:}a'_2, \ldots, p{:}a'_m$ are all the events of process p in es and in the same order. Then, given a derivation $\delta = (\alpha_0 \rightsquigarrow_{e_1} \alpha_1 \rightsquigarrow_{e_2} \cdots \rightsquigarrow_{e_n} \alpha_{n+1})$, $n > 0$, the associated trace, in symbols $\mathsf{trace}(\delta)$, is a trace $\mathcal{T}$ such that $\mathcal{T}(p_i) = \mathsf{actions}(p_i, es)$ for each pid p_i occurring in $es = (e_1, \ldots, e_n)$.

Following [12], we say that two derivations are *causally equivalent* if their logs are the same (cf. Theorem 3.6 in [12]).[6] Now, we focus on two scenarios for prefix-based tracing: "pure tracing" and "pure replay". In the following, we say that a logged system is *initial* if it has the form $\mathcal{E}; \mathcal{E}; \langle p, ls, [\,] \rangle$. By abuse of notation, we let $\mathcal{E}$ denote both a log where pid p is mapped to an empty sequence and a network where the queue of (p, p) is empty. Similarly to the previous section, we assume that all derivations start with an initial logged system.

The following result states that prefix-based tracing is indeed a conservative extension of the standard semantics:

Theorem 1 (pure tracing). *Let $\alpha \hookrightarrow \ldots \hookrightarrow \alpha'$ be a derivation with the standard semantics (Fig. 2). Then, there is a derivation $\delta = (\mathcal{E}; \alpha \rightsquigarrow_{e_1} \ldots \rightsquigarrow_{e_n} \mathcal{E}; \alpha')$ with the prefix-based semantics of Figure 4, where $\mathsf{trace}(\delta)$ is its associated trace.*

Proof. The proof is straightforward since function next always returns a fresh pid/tag and function admissible always returns true when the log is empty. Therefore, the only difference between the rules in Figure 2 and those in Figure 4 when the log is empty is that the transitions are labeled with the corresponding event, so that a trace can be obtained. □

Let us now consider pure replay. In the following, we assume that all logs are *consistent*, i.e., they have been obtained from the trace of a derivation. Moreover, we say that a derivation *consumes* a log when it only performs a transition for process p if $\mathcal{W}(p)$ is not an empty sequence. In other words, it performs a replay of the execution represented by the log, and no more. The next result states that, given the log of a derivation, prefix-based tracing with this log produces a derivation which is causally equivalent to the original one.

[6] To be precise, the semantics in [12] does not consider process mailboxes nor message deliveries. Nevertheless, these actions are not observable in logs, and hence the property carry over easily to our case.

Theorem 2 (pure replay). *Let $\mathcal{W}$ be a nonempty log and let $\delta = (\mathcal{W}; \alpha \rightsquigarrow_{e_1} \ldots \rightsquigarrow_{e_n} \mathcal{E}; \alpha')$ be a derivation with the rules of Figure 4 that consumes log $\mathcal{W}$. Then, $log(\mathsf{trace}(\delta)) = \mathcal{W}$.*

Proof. (Sketch) The claim follows easily by induction on the length of the considered derivation. Since the base case is trivial, let us consider the inductive case. Here, we make a case distinction on the applied rule to system $\mathcal{W}; \alpha$:

- If we perform a step with rules *Exit*, *Local*, *Self* or *Deliver*, the claim follows trivially by induction since they have no impact on $log(\mathsf{trace}(\delta))$.
- Consider now a step with rule *Spawn* applied to a process p, and assume that the log has the form $\mathcal{W}[p \mapsto \mathsf{spawn}(p')+as]$ and the step is labeled with the event $p{:}\mathsf{spawn}(p')$. Hence, $\mathsf{trace}(\delta)$ associates an action $\mathsf{spawn}(p')$ to process p and so does $log(\mathsf{trace}(\delta))$. Then, the claim follows by applying the inductive hypothesis on the derived system $\mathcal{W}[p \mapsto as]; \alpha''$. A similar reasoning can be made with rule *Receive*.
- Finally, we consider rule *Send* applied to process p, and assume that the log has the form $\mathcal{W}[p \mapsto \mathsf{send}(\ell)+as]$. Here, $\mathsf{trace}(\delta)$ associates an action $\mathsf{send}(\ell, p')$ to process p and, thus, $log(\mathsf{trace}(\delta))$ will add $\mathsf{send}(\ell)$ to the sequence of actions of process p. Then, the claim follows by applying the inductive hypothesis on the derived system $\mathcal{W}[p \mapsto as]; \alpha''$. □

In the next section, we introduce an implementation of prefix-based tracing by means of a program instrumentation.

4 A Program Instrumentation for Prefix-Based Tracing

Now, we focus on the design of a program instrumentation to perform prefix-based tracing in Erlang. In a nutshell, our program instrumentation proceeds as follows:

- First, we introduce a new process, called the *scheduler* (a server), that will be run as part of the source program.
- The scheduler ensures that the actions of a given log are followed in the same order, and that the corresponding trace is eventually computed. It also includes a data structure that corresponds to the network Γ introduced in the previous section. In the instrumented program, all messages will be sent via the scheduler.
- Finally, the sentences that correspond to the concurrent actions spawn, send and rec are instrumented in order to interact with the scheduler. The remaining code will stay untouched.

The scheduler uses several data structures called *dictionaries*, a typical key-value data structure which is commonly used in Erlang applications. Here, we consider the following standard operations on dictionaries:

- $fetch(k, dict)$, which returns the value val associated to key k in $dict$. We write $dict[k]$ as a shorthand for $fetch(k, dict)$.

– $store(k, val, dict)$, which updates the dictionary by adding (or updating, if the key exists) a new pair with key k and value val. In this case, we write $dict[k] := val$ as a shorthand for $store(k, val, dict)$.

In particular, we consider the following dictionaries:

– Pids, which maps the pid of each process to a (unique) reference, i.e., Pids$[p]$ denotes the reference of pid p. While pids are relative to a particular execution (i.e., the pid of the same process may change from one execution to the next one), the corresponding reference in a log or trace is permanent. This mapping is used to dynamically keep the association between pids and references in each execution. For instance, an example value for Pids is $[\{\langle 0.80.0\rangle, \mathsf{p1}\}, \langle 0.83.0\rangle, \mathsf{p2}\}]$, where $\langle 0.80.0\rangle, \langle 0.83.0\rangle$ are Erlang pids and p1, p2 are the corresponding references.
– LT, which is used to associate each process reference with a tuple of the form $\{ls, as\}$, where ls is a (possibly empty) list with the events of a log and as is a (possibly empty) list with the (reversed) trace of the execution so far. The log is used to drive the next steps, while the second component is used to store the execution trace so far. The list storing the trace is reversed for efficiency reasons (since it is faster to add elements to the head of the list). E.g., the initial value of LT for the partial log displayed in (3) is as follows:

$$[\{\mathsf{p1},\ \{[\mathsf{spawn(p3)}, \mathsf{spawn(p2)}, \mathsf{spawn(p4)}, \mathsf{send}(\ell_1)], [\,]\}\},\\ \{\mathsf{p2},\ \{[\mathsf{send}(\ell_2)], [\,]\}\}, \{\mathsf{p3}, \{[\mathsf{rec}(\ell_2)], [\,]\}\}, \{\mathsf{p4}, \{[\mathsf{send}(\ell_5)], [\,]\}\}]$$

– MBox, which represents the network Γ, also called global mailbox. The key of this dictionary is the pid of the target process, and the value is another dictionary in which the keys are pids (those of the sender processes) and the values are lists of (tagged) messages. For instance, the value of MBox after sending the first two messages of the execution shown in Figure 3 could be as follows:

$$\{\langle 0.84.0\rangle,\ \{\langle 0.80.0\rangle, [\{\ell_1, v_1\}]\},\\ \{\langle 0.83.0\rangle, [\{\ell_2, v_2\}]\}\}$$

where $\langle 0.80.0\rangle, \langle 0.83.0\rangle, \langle 0.84.0\rangle$ are the pids of p1, p2, p3, respectively, v_1 and v_2 are the message values and ℓ_1 and ℓ_2 are their respective tags.

Let us now describe the instrumentation of the source code. First, every expression of the form spawn$(mod, fun, args)$ is replaced by a call to a new function spawn_inst with the same arguments. The implementation of this function is essentially as follows:

```
spawn_inst(M, F, A) →
    Pid = self(),
    SpawnPid = spawn(fun() →
                 sched ! {Pid, spawn, self()},
                 apply(M, F, A)
    end),
    receive ack → ok end,
    SpawnPid.
```

where spawn takes an anonymous function as argument (so that the new process will evaluate the body of the anonymous function) and Erlang's predefined function *apply* is used to compute the application of a function to some arguments.

Intuitively speaking, the new function (1) sends the message $\{P1, \mathsf{spawn}, P2\}$ to the scheduler (here denoted by sched), where $P1$ is the pid of the current process and $P2$ is the pid of the spawned process, and (2) inserts a receive expression to make this communication *synchronous*. The reason for (2) is that every message of the form $\{P1, \mathsf{spawn}, P2\}$ must add $P2$ to the data structure Pids, either with a new reference or with the one in the current log. We require this operation to be completed before either the spawned process or the one performing the spawn can proceed with any other action. Otherwise, the scheduler could run into an inconsistent state.

The instrumentation of message sending is much simpler. We just perform the following rewriting:

$$e_1 \,!\, e_2 \qquad \Rightarrow \qquad \mathsf{sched} \,!\, \{\mathsf{self}(), \mathsf{send}, e_1, e_2\}$$

where sched is the pid of the scheduler and self() is a predefined function that returns the pid of the current process. Finally, the instrumentation of a receive expression rewrites the code as follows:

$$\begin{array}{l} \mathsf{receive}\ p_1 \to e_1; \ldots; p_n \to e_n\ \mathsf{end} \\ \quad \Rightarrow \mathsf{receive}\ \{L_1, p_1\} \to \mathsf{sched} \,!\, \{\mathsf{self}(), \mathsf{rec}, L_1\}, e_1; \ldots; \\ \qquad\qquad \{L_n, p_n\} \to \mathsf{sched} \,!\, \{\mathsf{self}(), \mathsf{rec}, L_n\}, e_n\ \mathsf{end} \end{array}$$

where $L_1, \ldots, L_n$ are fresh variables that are used to gather the tag of the received message and send it to the scheduler.

The main algorithm of the scheduler can be found in Algorithm 1. First, we have an initialization where the pid of the main process is associated with the reference p1 in Pids, the initial logs are assigned to LT, and the mailbox is initially empty. As is common in server processes, the scheduler is basically an infinite loop with a receive statement to process the requests. Here, we consider three requests, which correspond to the messages sent from the instrumented source code. Let us briefly explain the actions associated to each message:

- If the message received has the form $\{p, \mathsf{spawn}, p'\}$, where p, p' are pids, we look for the tuple associated to process Pids[p] in LT. If the log is empty, we can proceed nondeterministically and just need to keep a trace of the execution step. Here, we obtain a fresh reference, r', add the pair $\{p', r'\}$ to Pids, and update the trace in LT with the new action spawn(r'). If the log is not empty, we proceed in a similar way but the reference is given in the log entry. Finally, we have to acknowledge the reception of this message since this communication is synchronous (as explained above).
- If the message received has the form $\{p, \mathsf{send}, p', v\}$, we again distinguish the case where the process log is empty. In this case, we obtain a fresh reference ℓ (the message tag) and update LT with the new action send(ℓ). Finally, we use the auxiliary function *process_new_msg* to check the log of the target process, p', and then it proceeds as follows:

Algorithm 1 Scheduler

```
Initialization
  Pids := [{self(), p1}];  LT := /* prefix logs */;   MBox := { };
repeat
 receive
  {p, spawn, p'} →
      case LT[Pids[p]] of
        {[ ], as} →   /* trace mode */
                      r' := new_unique_ref();
                      update_pids(p', r', Pids);
                      LT[Pids[p]] := {[ ], [spawn(r')|as]};
        {[spawn(r')|ls], as} →   /* replay mode */
                      update_pids(p', r', Pids);
                      LT[Pids[p]] := {ls, [spawn(r')|as]};
      p ! ack;
      try_deliver(p);
  {p, send, p', v} →
      case LT[Pids[p]] of
        {[ ], as} →   /* trace mode */
                      ℓ := new_unique_ref();
                      LT[Pids[p]] := {[ ], [send(ℓ)|as]};
                      process_new_msg({p, p', ℓ, v}, MBox, LT);
        {[send(ℓ)|ls], as} →   /* replay mode */
                      LT[Pids[p]] := {ls, [send(ℓ)|as]};
                      process_msg({p, p', ℓ, v}, MBox, LT);
      try_deliver(p);
  {p, rec, ℓ} →
      case LT[Pids[p]] of
        {[ ], as} →   /* trace mode */
                      LT[Pids[p]] := {[ ], [rec(ℓ)|as]}
        {[rec(ℓ)|ls], as} →   /* replay mode */
                      LT[Pids[p]] := {ls, [rec(ℓ)|as]}
      try_deliver(p)
until true
```

- If the log of $\mathsf{Pids}[p']$ is empty, we add the action $\mathsf{deliver}(\ell)$ to the trace of $\mathsf{Pids}[p']$ and then send the message to the target process: $p'!\{\ell, v\}$, i.e., we apply an *instant-delivery* strategy, where messages are delivered as soon as possible (this is the usual action in the Erlang runtime environment).

- If the log is not empty, we do not know when this message should be received. Hence, we add a new (tagged) message $\{\ell, v\}$ from p to p' to the mailbox MBox, and add an action $\mathsf{deliver}(\ell)$ at the end of the current log. Note that computed logs (as in [11,12]) should not contain deliver actions. This one is artificially added to *force* the delivery of message ℓ as soon as possible (see function *try_deliver* below).

The pseudocode of function *process_new_msg* can be found below:

```
process_new_msg({p, p', ℓ, v}, MBox, LT) →
    case LT[Pids[p']] of
        {[ ], as'} → p' ! {ℓ, v},
                     LT[Pids[p']] := {[ ], [deliver(ℓ)|as']};
        {as, as'} → add_message(p, p', {ℓ, v}, MBox),
                    LT[Pids[p']] := {as+deliver(ℓ), as'}
    end
```

If the log is not empty, we proceed in a similar way but the message tag is given by the log and we call the auxiliary function *process_msg* instead. This function checks the log of the target process, $\mathsf{Pids}[p']$, and then proceeds as follows:

- If the next action in the log is $\mathsf{rec}(\ell)$, we add the action $\mathsf{deliver}(\ell)$ to the trace of $\mathsf{Pids}[p']$ and send the message to the target process: $p' \,!\, \{\ell, v\}$.
- If the first action is not $\mathsf{rec}(\ell)$, we add a new (tagged) message $\{\ell, v\}$ from p to p' to the mailbox MBox. Finally, if the log of $\mathsf{Pids}[p']$ contains an action $\mathsf{rec}(\ell)$, we are done; otherwise, an action of the form $\mathsf{deliver}(\ell)$ is added to the end of the log of process $\mathsf{Pids}[p']$, as before.

The pseudocode of function *process_msg* can be found below:

```
process_msg({p, p', ℓ, v}, MBox, LT) →
    case LT[Pids[p']] of
        {[rec(ℓ)|as], as'} → p' ! {ℓ, v},
                             LT[Pids[p']] := {as, [deliver(ℓ)|as']};
        {as, as'} → add_message(p, p', {ℓ, v}, MBox),
                    if not(member(rec(ℓ), as))
                    then LT[Pids[p']] := {as+deliver(ℓ), as'}
    end
```

– Finally, when the received message has the form $\{p, \mathsf{rec}, \ell\}$, we just update the trace with the new action $\mathsf{rec}(\ell)$ and, if the log was not empty, we remove the first action $\mathsf{rec}(\ell)$ from the log.

Each of the above cases ends with a call *try_deliver*(p), which is basically used to deliver messages that could not be delivered before (because it would have violated the order of some log). For this purpose, this function checks the next action in the log of process $\mathsf{Pids}[p]$. If it has either the form $\mathsf{rec}(\ell)$ or $\mathsf{deliver}(\ell)$, and the message tagged with ℓ is the oldest one in one of the queues of MBox with target p, then we send the message to p, remove it from MBox and add $\mathsf{deliver}(\ell)$ to the trace of process $\mathsf{Pids}[p]$. Furthermore, in case the element of the log was $\mathsf{deliver}(\ell)$, we recursively call *try_deliver*(p) to see if there are more messages that can be delivered. In any other case, the function does nothing.

The implementation of the program instrumentation to perform prefix-based tracing is publicly available from `https://github.com/mistupv/cauder`.

5 Concluding Remaks

In this work, we have formalized the notion of prefix-based tracing, an essential component of state-space exploration methods, in the context of a message-passing concurrent language that can be seen as a subset of Erlang. We have proved that prefix-based tracing indeed subsumes traditional tracing and replay. Furthermore, we have implemented this operation by means of a program instrumentation which is parametric on the given input log.

We consider several interesting avenues for future work. On the one hand, we plan to extend prefix-based tracing to also consider several built-in's of the Erlang language that involve shared-memory concurrency. This extension will significantly extend the class of considered programs. On the other hand, an experimental evaluation will be carried over to determine the overhead introduced by the program instrumentation.

Acknowledgements. The authors would like to thank Ivan Lanese for his useful remarks on a preliminary version of this paper. We would also like to thank the anonymous reviewers and the participants of LOPSTR 2021 for their suggestions to improve this work.

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: A foundation for optimal dynamic partial order reduction. J. ACM **64**(4), 25:1–25:49 (2017). https://doi.org/10.1145/3073408
2. Christakis, M., Gotovos, A., Sagonas, K.: Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In: Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013). pp. 154–163. IEEE Computer Society (2013). https://doi.org/10.1109/ICST.2013.50
3. Erlang website. URL: `https://www.erlang.org/` (2021)
4. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005). pp. 110–121. ACM (2005). https://doi.org/10.1145/1040305.1040315
5. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: POPL. pp. 174–186 (1997). https://doi.org/10.1145/263699.263717
6. González-Abril, J.J., Vidal, G.: Causal-Consistent Reversible Debugging: Improving CauDEr. In: Morales, J.F., Orchard, D.A. (eds.) Proceedings of the 23rd International Symposium on Practical Aspects of Declarative Languages (PADL 2021). Lecture Notes in Computer Science, vol. 12548, pp. 145–160. Springer (2021). https://doi.org/10.1007/978-3-030-67438-0_9
7. González-Abril, J.J., Vidal, G.: A lightweight approach to computing message races with an application to causal-consistent reversible debugging (2021), `http://arxiv.org/abs/2112.12869`
8. Hwang, G., Tai, K., Huang, T.: Reachability testing: an approach to testing concurrent software. Int. J. Softw. Eng. Knowl. Eng. **5**(4), 493–510 (1995). https://doi.org/10.1142/S0218194095000241

9. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: A Causal-Consistent Reversible Debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) Proceedings of the 14th International Symposium on Functional and Logic Programming (FLOPS'18). Lecture Notes in Computer Science, vol. 10818, pp. 247–263. Springer (2018). https://doi.org/10.1007/978-3-319-90686-7_16
10. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018). https://doi.org/10.1016/j.jlamp.2018.06.004
11. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay debugging for message passing programs. In: Pérez, J.A., Yoshida, N. (eds.) Proceedings of the 39th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019). Lecture Notes in Computer Science, vol. 11535, pp. 167–184. Springer (2019). https://doi.org/10.1007/978-3-030-21759-4_10
12. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay reversible semantics for message passing concurrent programs. Fundam. Informaticae **178**(3), 229–266 (2021). https://doi.org/10.3233/FI-2021-2005
13. Lei, Y., Carver, R.H.: Reachability testing of concurrent programs. IEEE Trans. Software Eng. **32**(6), 382–403 (2006). https://doi.org/10.1109/TSE.2006.56
14. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, 1986. Lecture Notes in Computer Science, vol. 255, pp. 279–324. Springer (1987). https://doi.org/10.1007/3-540-17906-2_30
15. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M., López-García, P. (eds.) Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016). Lecture Notes in Computer Science, vol. 10184, pp. 259–274. Springer (2017). https://doi.org/10.1007/978-3-319-63139-4_15
16. Svensson, H., Fredlund, L.A., Earle, C.B.: A unified semantics for future Erlang. In: 9th ACM SIGPLAN workshop on Erlang. pp. 23–32. ACM (2010). https://doi.org/10.1145/1863509.1863514