Computing Race Variants in Message-Passing Concurrent Programming with Selective Receives*

Germán Vidal[0000−0002−1857−6951]

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

Abstract. Message-passing concurrency is a popular computation model that underlies several programming languages like, e.g., Erlang, Akka, and (to some extent) Go and Rust. In particular, we consider a message-passing concurrent language with dynamic process spawning and *selective* receives, i.e., where messages can only be consumed by the target process when they match a specific constraint (e.g., the case of Erlang). In this work, we introduce a notion of *trace* that can be seen as an abstraction of a class of *causally equivalent* executions (i.e., which produce the same outcome). We then show that execution traces can be used to identify message *races*. We provide constructive definitions to compute message races as well as to produce so-called *race variants*, which can then be used to drive new executions which are not causally equivalent to the previous ones. This is an essential ingredient of state-space exploration techniques for program verification.

Published as *Vidal, G. (2022). Computing Race Variants in Message-Passing Concurrent Programming with Selective Receives. In: Mousavi, M.R., Philippou, A. (eds) Formal Techniques for Distributed Objects, Components, and Systems. FORTE 2022. Lecture Notes in Computer Science, vol 13273. Springer, Cham.*

The final authenticated publication is available online at
https://doi.org/10.1007/978-3-031-08679-3_12

1 Introduction

Software verification and debugging are recognized as essential tasks in the field of software development. Not surprisingly, a recent study [27] points out that 26% of developer time is spent reproducing and fixing code bugs (which adds up to $61 billion annually). The study also identifies *reproducibility* of bugs as the biggest challenge to fix bugs faster. The situation is especially difficult for

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

concurrent and distributed applications because of nondeterminism. In this context, traditional testing techniques often provide only a poor guarantee regarding software correctness.

As an alternative, *state-space exploration* techniques constitute a well established approach to the verification of concurrent software that basically consists in exploring the reachable states of a program, checking whether a given property holds (like some type of deadlock, a runtime error, etc). This is the case, e.g., of *model checking* [7], where properties have been traditionally verified using a *model* of the program. More recently, several *dynamic* approaches to state-space exploration have been introduced, which work directly with the implementation of a program. *Stateless model checking* [12] and *reachability testing* [26,22] are examples of this approach. In turn, reproducibility of bugs has been tackled by so-called *record-and-replay* debuggers. In this case, a program is first instrumented so that its execution produces a *log* as a side-effect. If a problem occurs during the execution of the program, one can use the generated log to play it back in the debugger and try to locate the source of the misbehavior.

In this work, we focus on an asynchronous *message-passing* concurrent programming language like, e.g., Erlang [9], Akka [2] and, to some extent, Go [13] and Rust [25]. A running application consists of a number of processes, each with an associated (private) *mailbox*. Here, processes can only interact through (asynchronous) message sending and receiving, i.e., we do not consider shared-memory operations. Typically, there is some degree of nondeterminism in concurrent executions that may affect the outcome of a computation. For instance, when two processes send messages to another process, these messages may sometimes arrive in any order. These so-called *message races* play a key role in the execution of message-passing concurrent programs, and exploring all feasible combinations of message sending and receiving is an essential component of state-space exploration techniques.

In particular, we consider a language with so-called *selective* receives, where a process does not necessarily consume the messages in its mailbox in the same order they were delivered, since receive statements may impose additional constraints. For instance, a receive statement in Erlang has the form

$$\texttt{receive}\ p_1\ \texttt{[when}\ g_1\texttt{]} \rightarrow t_1\texttt{;}\ \ldots\texttt{;}\ p_n\ \texttt{[when}\ g_n\texttt{]} \rightarrow t_n\ \texttt{end}$$

In order to evaluate this statement, a process should look for the *oldest* message in its mailbox that matches a pattern p_i and the corresponding (optional) guard g_i holds (if any);[1] in this case, the process continues with the evaluation of expression t_i. When no message matches any pattern, the execution of the process is *blocked* until a matching message reaches its mailbox.

Considering a message-passing concurrent language with selective receives is relevant in order to deal with a language like Erlang. Unfortunately, current approaches either do not consider selective receives—the case of reachability testing [21] which, in contrast, considers different ports for receive statements—or have

[1] If the message matches several patterns, the first one is considered.

not formally defined the semantics of the language and its associated *happened-before* relation—the case of Concuerror [5], which implements a stateless model checker for Erlang that follows the approach in [1,3,4].

In this paper, we introduce a notion of *trace* that is tailored to message-passing execution with dynamic process spawning and selective receives. Our traces can be seen as an extension of the *logs* of [19,20], which were introduced in the context of causal-consistent *replay* (reversible) debugging in Erlang. In particular, the key extension consists in adding some additional information to the events of a trace, namely the identifier of the target process of a message and its actual value for *send* events, and the actual constraints of a receive statement for *receive* events. In this way, we can identify not only the communications performed in a program execution but also its message races (see the discussion in the next section).

In contrast to other notions of trace that represent a particular interleaving, our traces (analogously to the *logs* of [19,20] and the *SYN-sequences* of [21]) record the sequence of actions performed by each process in an execution, ignoring the concrete scheduling of all processes' actions. These traces can easily be obtained by instrumenting the source code so that each process keeps a record of its own actions. The traces can be seen as an abstraction of a class of executions which are *causally equivalent*. Roughly speaking, two executions are causally equivalent when the executed actions and the final outcome are the same but the particular scheduling might differ. We then introduce constructive definitions for computing message races and *race variants* from a given trace. Here, race variants are denoted by a (possibly partial) trace which can then be used to drive a new program execution (as in the replay debugger CauDEr [10]). Moreover, we prove that any execution that follows (and possibly goes beyond) the computed race variant cannot give rise to an execution which is causally equivalent to the previous one, an essential property of state-space exploration techniques.

The paper is organized as follows. After some motivation in Section 2, we formalize the notions of *interleaving* and *trace* in Section 3, where we also provide a declarative definition of message race and prove a number of properties. Then, Section 4 provides constructive definitions for computing message races and race variants, and proves that race variants indeed give rise to executions which are not causally equivalent to the previous one. Finally, Section 5 presents some related work and concludes.

2 Message Races and Selective Receives

In this section, we informally introduce the considered setting and motivate our definition of *trace*. As mentioned before, we consider a message-passing (asynchronous) concurrent language with selective receives. Essentially, concurrency follows the actor model: at runtime, an application can be seen as a collection of processes that interact through message sending and receiving. Each process has an associated identifier, called *pid* (which stands for process identifier), that is

```
proc1() -> P2 = spawn(proc2()),
           P3 = spawn(proc3(P2)),
           send({val,1},P2).
proc2() -> receive
              {val,M} when M>0 -> {ok,M};
              error -> error
           end.
proc3(P2) -> send({val,0},P2),
             send({val,2},P2).
```

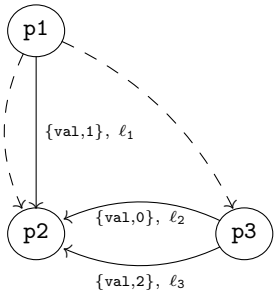

Fig. 1: A simple message-passing program

unique in the execution.[2] Furthermore, we assume that processes can be spawned *dynamically* at runtime.

As in other techniques where message races are computed, e.g., *dynamic partial order reduction* (DPOR) [11,1] for *stateless model checking* [12], we distinguish *local* evaluations from *global* (or *visible*) actions. Examples of local evaluations are, e.g., a function call or the evaluation of a case expression. In turn, *global* actions include the spawning of a new process as well as any event related with message passing. In particular, we consider that sent messages are eventually stored in the *mailbox* of the target process. Then, the target process can *consume* these messages using a *receive* statement, which we assume is selective, i.e., it may impose some additional constraints on the receiving messages.

Example 1. Let us consider the simple code shown in Figure 1 (we use a pseudocode that resembles the syntax of Erlang [9]). Assume that the initial process (the one that starts the execution) has pid `p1` and that it begins with a call to function `proc1`, which first spawns two new processes with pids `p2` and `p3` that will evaluate the calls `proc2()` and `proc3(P2)`, respectively. A call to `spawn` returns the pid of the new process, so variables `P2` and `P3` are bound to pids `p2` and `p3`, respectively. The evaluation of `proc1()` ends by sending the message `{val,1}` to process `p2`. Process spawning is denoted by a dashed arrow in the diagram, while message sending is denoted by a solid arrow. Messages are *tagged* with a unique identifier (e.g., ℓ_1).

Process `p3` sends two messages, `{val,0}` and `{val,2}`, to process `p2`. Process `p2` initially blocks waiting for the arrival of a message that matches either the pattern `{val,M}`, i.e., a tuple whose first component is the constant `val` and the second component (denoted by variable `M`) is an integer greater than zero, or the constant `error`. Note that message $\{\texttt{val}, 0\}$ does not match the constraints of the receive statement since the integer value is not greater than zero. Thus, only messages ℓ_1 and ℓ_3 can be consumed by the receive statement of process `p2`.

[2] In the following, we often say "process p" to mean "process with pid p".

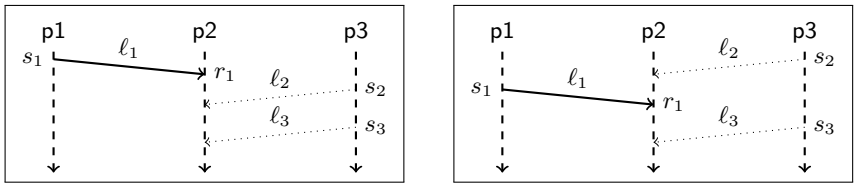

Fig. 2: Alternative interleavings for the execution of the program of Figure 1. We have three processes, identified by pids p1, p2 and p3. Solid arrows denote the connection between messages sent and received (similarly to the synchronization pairs of [21]), while dotted arrows represent messages sent but not yet received. Time, represented by dashed lines, flows from top to bottom.

In principle, one could represent a program execution by means of a concrete *interleaving* of its concurrent actions, i.e., a sequence of *events* of the form *pid*: *action*. E.g., we could have the following interleaving for the program of Figure 1:

```
(1) p1:spawn(proc2())          (4) p2:receive({val,1})
(2) p1:spawn(proc3(p2))        (5) p3:send({val,0},p2)
(3) p1:send({val,1},p2)        (6) p3:send({val,2},p2)
```

This interleaving is graphically depicted in Figure 2a, where process spawning is omitted for clarity.

In this work, though, we opt for a different representation, which is similar to the notion of *log* (in the context of replay debugging [19,20]) and that of *SYN-sequence* (in reachability testing [21]). In contrast to interleavings (as in, e.g., stateless model checking [12] and DPOR techniques [11,1]), the advantage of using logs is that they represent a *partial order* for the concurrent actions, so that DPOR techniques are no longer needed. To be more precise, a log (as defined in [19]) maps each process to a sequence of the following actions:

- process spawning, denoted by $\mathsf{spawn}(p)$, where p is the *pid* of the new process;
- message sending, denoted by $\mathsf{send}(\ell)$, where ℓ is a message tag;
- and message reception, denoted by $\mathsf{rec}(\ell)$, where ℓ is a message tag.

In contrast to the SYN-sequences of [21], *synchronization pairs* (connecting message sending and receiving) are not explicitly considered but can easily be inferred from send/receive actions with the same message tag. Furthermore, logs include spawn actions because runtime processes are not statically fixed, which is not considered by SYN-sequences. Logs are used by the reversible debugger CauDEr [10] as part of an approach to *record-and-replay* debugging in Erlang (a popular approach to deal with the problem of reproducibility of bugs).

In practice, logs can be obtained by using an instrumented semantics (as in [19,20]) or by instrumenting the program so that its execution (in the standard environment) produces a log as a side-effect (along the lines of the technique presented in [15]). It is worthwhile to note that no centralised monitoring is

required; every process only needs to register its own actions independently. For instance, a log associated with the execution shown in Figure 1 is as follows:

$$[\mathsf{p1} \mapsto \mathsf{spawn}(\mathsf{p2}), \mathsf{spawn}(\mathsf{p3}), \mathsf{send}(\ell_1);\quad \mathsf{p2} \mapsto \mathsf{rec}(\ell_1);\quad \mathsf{p3} \mapsto \mathsf{send}(\ell_2), \mathsf{send}(\ell_3)]$$

Unfortunately, this log does not contain enough information for computing *message races*. A first, obvious problem is that *send* events do not include the pid of the target process. Hence, even if there is a (potential) race between messages ℓ_1 and ℓ_2 to reach process $\mathsf{p2}$, this cannot be determined from the log. Trivially, one could solve this problem by adding the pid of the target process to every send event, as follows (we omit spawn actions since they are not relevant for the discussion):

$$[\mathsf{p1} \mapsto \ldots, \mathsf{send}(\ell_1, \mathsf{p2});\quad \mathsf{p2} \mapsto \mathsf{rec}(\ell_1);\quad \mathsf{p3} \mapsto \mathsf{send}(\ell_2, \mathsf{p2}), \mathsf{send}(\ell_3, \mathsf{p2})] \qquad (*)$$

Now, in principle, one could say that messages ℓ_1 and ℓ_2 race for process $\mathsf{p2}$ since the target is the same ($\mathsf{p2}$) and there are no dependencies among $\mathsf{send}(\ell_1, \mathsf{p2})$ and $\mathsf{send}(\ell_2, \mathsf{p2})$ (s_1 and s_2 in Figure 2a).

However, when we consider *selective* receives, the log (*) above can be ambiguous. In particular, this log represents both interleavings represented by the diagrams of Figure 2a and Figure 2b (where message ℓ_2 reaches first process $\mathsf{p2}$ but its associated value, $\{\mathtt{val}, \mathtt{0}\}$, does not match the constraints of the the receive statement since the guard `M>0` does not hold for `M=0`). However, the diagram in Figure 2a points out to a (potential) message race between messages ℓ_1 and ℓ_2, while the diagram in Figure 2b suggests a (potential) message race between messages ℓ_1 and ℓ_3 instead.

In order to distinguish the executions shown in Figures 2a and 2b one could add a new action, $\mathsf{deliver}(\ell)$, to explicitly account for the delivery of a message with tag ℓ. In this way, we would know the order in which messages are stored in the process' mailbox, which uniquely determines the order in which they can be consumed by receive statements. E.g., the actions of process $\mathsf{p2}$ in the execution of Figure 2a would be

$$[\ \ldots\ \mathsf{p2} \mapsto \mathsf{deliver}(\ell_1), \mathsf{rec}(\ell_1), \mathsf{deliver}(\ell_2), \mathsf{deliver}(\ell_3)\ \ldots\ ] \qquad (**)$$

while those of Figure 2b would be as follows:

$$[\ \ldots\ \mathsf{p2} \mapsto \mathsf{deliver}(\ell_2), \mathsf{deliver}(\ell_1), \mathsf{rec}(\ell_1), \mathsf{deliver}(\ell_3)\ \ldots\ ] \qquad (***)$$

This approach is explored in [14], where a *lightweight* (but approximate) technique to computing message races is proposed. Unfortunately, making explicit message delivery does not suffice to allow one to compute message races in general. In particular, while it would allow us to distinguish the situation of Figure 2a from that of Figure 2b, we could not still determine whether there is an actual race between messages ℓ_1 and ℓ_2 or between messages ℓ_1 and ℓ_3. For instance, for the program of Figure 1, only the message race between ℓ_1 and ℓ_3 is feasible, as explained above. For this purpose, we need to also include the actual values of messages as well as the constraints of receive statements.

In the next section, we propose an appropriate definition of *trace* (an extended log) that includes enough information for computing message races.

3 Execution Traces

In this section, we formalize an appropriate notion of trace that is adequate to compute message races in a message-passing concurrent language with dynamic process spawning and selective receives. Here, we do not consider a specific programming language but formalize our developments in the context of a generic language that includes the basic actions spawn, send, and receive.

As mentioned in the previous section, we consider that each process is uniquely identified by a pid. A message takes a *value* v (from a given domain) and is uniquely identified by a tag ℓ.[3] We further require the domains of pids, values, and tags to be disjoint. We also consider a generic domain of constraints and a decidable function match so that, for all value v and constraint cs, $\mathsf{match}(v, cs)$ returns *true* if the value matches the constraint cs and *false* otherwise. In Erlang, for instance, a constraint is associated with the clauses of a receive statement, i.e., it has the form $(p_1\ \texttt{[when}\ g_1\texttt{]} \to t_1\texttt{;}\ \ldots\texttt{;} p_n\ \texttt{[when}\ g_n\texttt{]} \to t_n)$, and function match determines if a value v matches some pattern p_i and the associated guard g_i (if any) evaluates to true.

In this work, events have the form $p\!:\!a$, where p is a pid and a is one of the following actions:

- $\mathsf{spawn}(p')$, which denotes the spawning of a new process with pid p';
- $\mathsf{send}(\ell, v, p')$, which denotes the sending of a message with tag ℓ and value v to process p';
- $\mathsf{rec}(\ell, cs)$, which denotes the *reception* of a message with tag ℓ by a receive statement with a constraint cs.[4]

In the following, a (finite) sequence is denoted as follows: $e_1, e_2, \ldots, e_n$, $n \geq 0$, where n is the length of the sequence. We often use set notation for sequences and let $e \in S$ denote that event e occurs in sequence S. Here, ϵ denotes an empty sequence, while $S;S'$ denotes the concatenation of sequences S and S'; by abuse of notation, we use the same operator when a sequence has only a single element, i.e., $e_1;(e_2, \ldots, e_n)$ and $(e_1, \ldots, e_{n-1});e_n$ both denote the sequence $e_1, \ldots, e_n$. Furthermore, given a sequence of events

$$S = (p_1\!:\!a_1, p_2\!:\!a_2, \ldots, p_n\!:\!a_n)$$

we let $\mathsf{actions}(p, S)$ 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 and only the events of process p in S and in the same order. Given a sequence $S = (e_1, \ldots, e_n)$, we also say that e_i *precedes* e_j, in symbols $e_i \prec_S e_j$, if $i < j$.

[3] Message tags were introduced in [18] to uniquely identify messages, since we might have several messages with the same value and would be indistinguishable otherwise.

[4] Note that *receive* actions represent the *consumption* of messages by receive statements rather than their delivery to the process' mailbox. Observe that the order of message delivery and message reception might be different (see, e.g., messages ℓ_1 and ℓ_2 in Figure 2b).

Now, we formalize the notions of *interleaving* and *trace*. Intuitively speaking, an interleaving is a sequence of events that represents a *linearization* of the actions of a concurrent execution, while a trace is a mapping from processes to sequences of actions (so a trace only denotes a *partial relation* on events).

Definition 1 (interleaving). *A sequence of events $S = (p_1 : a_1, \ldots, p_n : a_n)$ is an* interleaving *with initial pid p_1 if the following conditions hold:*

1. *Each event $(p_j : a_j) \in S$ is either preceded by an event $(p_i : \mathsf{spawn}(p_j)) \in S$ with $p_i \neq p_j$, $1 \leq i < j \leq n$, or $p_j = p_1$.*
2. *Each event $(p_j : \mathsf{rec}(\ell, cs)) \in S$ is preceded by an event $(p_i : \mathsf{send}(\ell, v, p_j)) \in S$, $1 \leq i < j \leq n$, such that $\mathsf{match}(v, cs) = true$.*
3. *For each pair of events $p_i : \mathsf{send}(\ell, v, p_j)$, $p_j : \mathsf{rec}(\ell, cs) \in S$, we have that, for all $p_i : \mathsf{send}(\ell', v', p_j) \in S$ that precedes $p_i : \mathsf{send}(\ell, v, p_j)$, in symbols*

   $$p_i : \mathsf{send}(\ell', v', p_j) \prec_S p_i : \mathsf{send}(\ell, v, p_j)$$

   either $\mathsf{match}(v', cs) = false$ or there is an event $p_j : \mathsf{rec}(\ell', cs') \in S$ such that $p_j : \mathsf{rec}(\ell', cs') \prec_S p_j : \mathsf{rec}(\ell, cs)$.
4. *Finally, for all event $p_i : \mathsf{spawn}(p_j)$, p_j only occurs as the argument of* spawn *in this event, and for all event $p_i : \mathsf{send}(\ell, v, p_j)$, ℓ only occurs as the argument of* send *in this event (uniqueness of pids and tags).*

The first two conditions in the definition of interleaving are very intuitive: all the actions of a process (except for those of the initial process, p_1) must happen after its spawning, and each reception of a message ℓ must be preceded by a sending of message ℓ and, moreover, the message value should match the receive constraint. The third condition is a bit more involved but can be explained as follows: the messages sent between two given processes should be delivered in the same order they were sent. Thus, if a process p_j receives a message ℓ from process p_i, all previous messages sent from p_i to p_j (if any) should have been already received or their value should not match the constraint of the receive statement. The last condition simply ensures that pids and tags are unique in an interleaving, as mentioned before.

Example 2. Consider the program of Example 1. A possible interleaving is

$$\mathsf{p1} : \mathsf{spawn}(\mathsf{p2}),\ \mathsf{p1} : \mathsf{spawn}(\mathsf{p3}),\ \mathsf{p1} : \mathsf{send}(\ell_1, v_1, \mathsf{p2}),\ \mathsf{p2} : \mathsf{rec}(\ell_1, cs_1),$$
$$\mathsf{p3} : \mathsf{send}(\ell_2, v_2, \mathsf{p2}),\ \mathsf{p3} : \mathsf{send}(\ell_3, v_3, \mathsf{p2})$$

which can be graphically represented by the diagram of Figure 2a.

An interleaving induces a *happened-before* relation [17] on events as follows:

Definition 2 (happened-before, independence). *Let $S = (e_1, \ldots, e_n)$ be an interleaving. We say that $e_i = (p_i : a_i)$ happened before $e_j = (p_j : a_j)$, $i < j$, in symbols $e_i \rightsquigarrow_S e_j$, if one of the following conditions hold:*

1. *$p_i = p_j$ (i.e., the actions of a given process cannot be swapped);*

2. $a_i = \mathsf{spawn}(p_j)$ *(i.e., a process cannot perform an action before it is spawned);*
3. $a_i = \mathsf{send}(\ell, v, p_j)$ *and* $a_j = \mathsf{rec}(\ell, cs)$ *(i.e., a message cannot be received before it is sent).*

If $e_i \rightsquigarrow_S e_j$ *and* $e_j \rightsquigarrow_S e_k$*, then* $e_i \rightsquigarrow_S e_k$ *(transitivity). If neither* $e_i \rightsquigarrow_S e_j$ *nor* $e_j \rightsquigarrow_S e_i$*, we say that the two events are independent.*

Given an interleaving S, the associated happened-before relation $\rightsquigarrow_S$ is clearly a (strict) partial order since the following properties hold:

- No event may happen before itself (irreflexivity) since $e_i \rightsquigarrow_S e_j$ requires $i < j$ by definition.[5]
- If $e_i \rightsquigarrow_S e_j$ we have $i < j$ and, thus, $e_j \rightsquigarrow_S e_i$ is not possible (asymmetry).
- Finally, the relation $\rightsquigarrow_S$ is transitive by definition.

In the following, we say that two interleavings are *causally equivalent* if they have the same events and only differ in the swapping of a number of independent events. Formally,

Definition 3 (causal equivalence). *Let* S_1 *and* S_2 *be interleavings with the same initial pid. We say that* S_1 *and* S_2 *are* causally equivalent*, in symbols* $S_1 \approx S_2$*, if* S_2 *can be obtained from* S_1 *by a finite number of swaps of consecutive independent events.*

We note that our notion of causal equivalence is similar to that of *trace equivalence* in [23] and that of causally equivalent *derivations* in [19,20].

The causal equivalence relation on interleavings is an *equivalence relation* since it is trivially reflexive ($S \approx S$ holds for all interleavings), symmetric ($S_1 \approx S_2$ implies $S_2 \approx S_1$ by considering the same swaps in the reverse order), and transitive ($S_1 \approx S_2$ and $S_2 \approx S_3$ implies $S_1 \approx S_3$ by considering first the swaps that produce S_2 from S_1 and, then, those that transform S_2 into S_3).

It is worthwhile to note that not all independent events can be swapped if we want to produce a valid interleaving. Let us illustrate this point with an example:

Example 3. Consider the interleaving shown in Example 2 that is graphically represented in the diagram of Figure 2a. Here, we might perform a number of swaps of independent consecutive events so that we end up with the following causally equivalent interleaving:

$$\mathsf{p1}:\mathsf{spawn}(\mathsf{p2}),\ \mathsf{p1}:\mathsf{spawn}(\mathsf{p3}),\ \mathsf{p3}:\mathsf{send}(\ell_2, v_2, \mathsf{p2}),\ \mathsf{p1}:\mathsf{send}(\ell_1, v_1, \mathsf{p2}),\ \mathsf{p2}:\mathsf{rec}(\ell_1, cs_1),\ \mathsf{p3}:\mathsf{send}(\ell_3, v_3, \mathsf{p2})$$

which corresponds to the diagram of Figure 2b. In this case, we were able to swap the events $\mathsf{p1}:\mathsf{send}(\ell_1, v_1, \mathsf{p2})$ and $\mathsf{p3}:\mathsf{send}(\ell_2, v_2, \mathsf{p2})$ because they are independent and, moreover, the resulting interleaving does not violate condition (3) in Definition 1 since $\mathsf{match}(v_2, cs_1) = \mathit{false}$.

[5] Note that repeated events in an interleaving are not allowed by Definition 1.

In contrast, we could not swap $\mathsf{p1} : \mathsf{send}(\ell_1, v_1, \mathsf{p2})$ and $\mathsf{p3} : \mathsf{send}(\ell_3, v_2, \mathsf{p2})$ since the resulting sequence of events

$$\mathsf{p1}{:}\mathsf{spawn}(\mathsf{p2}),\ \mathsf{p1}{:}\mathsf{spawn}(\mathsf{p3}),\ \mathsf{p3}{:}\mathsf{send}(\ell_2, v_2, \mathsf{p2}),\ \mathsf{p3}{:}\mathsf{send}(\ell_3, v_3, \mathsf{p2}),\ \mathsf{p1}{:}\mathsf{send}(\ell_1, v_1, \mathsf{p2}),\ \mathsf{p2}{:}\mathsf{rec}(\ell_1, cs_1)$$

would not be an interleaving because it would violate condition (3) in Definition 1; namely, we have $\mathsf{match}(v_3, cs_1) = \mathit{true}$ and, thus, event $\mathsf{p2} : \mathsf{rec}(\ell_1, cs_1)$ would not be correct in this position (message ℓ_3 should be received instead).

In general, we can easily prove that the swap of two independent events in an interleaving always produces a valid interleaving (according to Definition 1) except when the considered events are both send with the same source and target pids. In this last case, it depends on the particular interleaving, as illustrated in the previous example.

A straightforward property is the following: causally equivalent interleavings induce the same happened-before relation, and vice versa.

Lemma 1. *Let S, S' be interleavings with the same initial pid. Then, we have $S \approx S'$ iff $\rightsquigarrow_S = \rightsquigarrow_{S'}$.*

While interleavings might be closer to an actual execution, it is often more convenient to have a higher-level representation, one where all causally equivalent interleavings have the same representation. For this purpose, we introduce the notion of *trace* as a mapping from pids to sequences of actions. Here, the key idea is to keep the actions of each process separated.

First, we introduce some notation. Let τ be a mapping from pids to sequences of actions, which we denote by a finite mapping of the form

$$[p_1 \mapsto A_1; \ldots; p_n \mapsto A_n]$$

Given an interleaving S, we let

$$tr(S) = [p_1 \mapsto \mathsf{actions}(p_1, S); \ldots; p_n \mapsto \mathsf{actions}(p_n, S)]$$

where $p_1, \ldots, p_n$ are the pids in S. We also let $\tau(p)$ denote the sequence of actions associated with process p in τ. Also, $\tau[p \mapsto A]$ denotes that τ is an arbitrary mapping such that $\tau(p) = A$; we use this notation either as a condition on τ or as a modification of τ. We also say that $(p : a) \in \tau$ if $a \in \tau(p)$. Moreover, we say that $p_1 : a_1$ *precedes* $p_2 : a_2$ in τ, in symbols $(p_1 : a_1) \prec_\tau (p_2 : a_2)$, if $p_1 = p_2$, $\tau(p_1) = A$, and a_1 precedes a_2 in A; otherwise, the (partial) relation is not defined.

Definition 4 (trace). *A* trace τ *with initial pid p_0 is a mapping from pids to sequences of actions if $tr(S) = \tau$ for some interleaving S with initial pid p_0.*

One could give a more direct definition of trace by mimicking the conditions of an interleaving, but the above, indirect definition is simpler.

A trace represents a so-called *Mazurkiewicz trace* [23], i.e., it represents a *partial order relation* (using the terminology of model checking [12]), where all linearizations of this partial order represent causally equivalent interleavings. In particular, given a trace τ, we let $\mathsf{sched}(\tau)$ denote the set of all *causally equivalent* linearizations of the events in τ, which is formalized as follows:

Definition 5. *Let τ be a trace with initial pid p_0. We say that an interleaving S with initial pid p_0 is a linearization of τ, in symbols $S \in \mathsf{sched}(\tau)$, if $tr(S) = \tau$.*

The following property is a trivial consequence of our definition of function sched:

Lemma 2. *Let S, S' be interleavings with the same initial pid and such that $\mathsf{actions}(p, S) = \mathsf{actions}(p, S')$ for all pid p in S, S'. Then, $tr(S) = tr(S')$.*

Proof. The proof is a direct consequence of the definition of function tr, since only the relative actions of each process are recorded in a trace.

The next result states that all the interleavings in $\mathsf{sched}(\tau)$ are indeed causally equivalent:

Theorem 1. *Let τ be a trace. Then, $S, S' \in \mathsf{sched}(\tau)$ implies $S \approx S'$.*

Proof. Let us consider two different interleavings $S, S' \in \mathsf{sched}(\tau)$. By the definition of function tr and Definition 5, S and S' have the same events and the same initial pid. Also, both interleavings have the same relative order for the actions of each process. Moreover, by definition of interleaving (Definition 1), we know that all events $p\!:\!a$ of a process (but the initial one) must be preceded by an event $p' : \mathsf{spawn}(p)$, and that all receive events $p : \mathsf{rec}(\ell, cs)$ must be preceded by a corresponding send event $p' : \mathsf{send}(\ell, v, p)$. Therefore, the happened-before relation induced from S and S' must be the same and, thus, $S \approx S'$ by Lemma 1. □

Trivially, all interleavings in $\mathsf{sched}(\tau)$ induce the same happened-before relation (since they are causally equivalent). We also say that τ induces the same happened-before relation (i.e., $\rightsquigarrow_S$ for any $S \in \mathsf{sched}(\tau)$) and denote it with $\rightsquigarrow_\tau$.

The following result is also relevant to conclude that a trace represents *all and only* the causally equivalent interleavings.

Theorem 2. *Let τ be a trace and $S \in \mathsf{sched}(\tau)$ an interleaving. Let S' be an interleaving with $S' \notin \mathsf{sched}(\tau)$. Then, $S \not\approx S'$.*

Proof. Assume that S and S' have the same events and that $\mathsf{actions}(p, S) = \mathsf{actions}(p, S')$ for all pid p in S, S' (otherwise, the claim follows trivially). Let us proceed by contradiction. Assume that $S' \notin \mathsf{sched}(\tau)$ and $S \approx S'$. Since $\mathsf{actions}(p, S) = \mathsf{actions}(p, S')$ for all pid p in S, S', we have $tr(S) = tr(S')$ by Lemma 2. Thus, $S' \in \mathsf{sched}(\tau)$, which contradicts our assumption. □

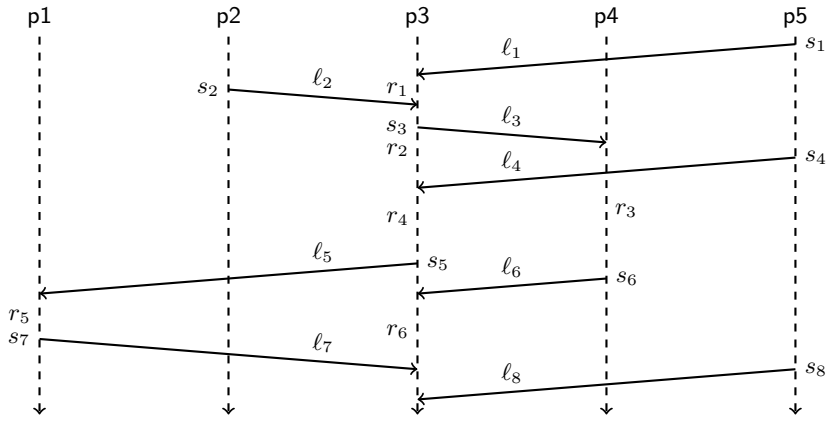

Fig. 3: Message-passing diagram. Processes (pi, $i = 1, \ldots, 5$) are represented as vertical dashed arrows, where time flows from top to bottom. Message sending is represented by solid arrows labeled with a tag (ℓ_i), $i = 1, \ldots, 8$. Note that all events associated with a message ℓ_i have the same subscript i.

Example 4. Consider the following trace τ, where we abbreviate $\mathsf{send}(\ell_i, v_i, p_i)$ as s_i and $\mathsf{rec}(\ell_i, cs_i)$ as r_i:

$$[\ \mathsf{p1} \mapsto \mathsf{spawn(p3)}, \mathsf{spawn(p2)}, \mathsf{spawn(p4)}, \mathsf{spawn(p5)}, r_5, s_7; \quad \mathsf{p2} \mapsto s_2;$$
$$\mathsf{p3} \mapsto r_1, s_3, r_2, r_4, s_5, r_6; \quad \mathsf{p4} \mapsto r_3; \quad \mathsf{p5} \mapsto s_1, s_4, s_8 \ ]$$

A possible execution following this trace is graphically depicted in Figure 3, where spawn actions have been omitted for clarity. Moreover, we assume that arrowheads represent the point in time where messages are delivered to the target process. Despite the simplicity of traces, we can extract some interesting conclusions. For example, if we assume that τ is the trace of a terminating execution, we might conclude that messages ℓ_7 and ℓ_8 are *orphan* messages (i.e., messages that are sent but never received) since there are no corresponding events r_7 and r_8 in τ. A possible interleaving in $\mathsf{sched}(\tau)$ that follows the diagram in Figure 3 is as follows:

$$\mathsf{p_1} : \mathsf{spawn}(p_3),\ \mathsf{p_1} : \mathsf{spawn}(p_2),\ \mathsf{p_1} : \mathsf{spawn}(p_4),\ \mathsf{p_1} : \mathsf{spawn}(p_5),$$
$$\mathsf{p_5} : s_1,\ \mathsf{p_3} : r_1,\ \mathsf{p_2} : s_2,\ \mathsf{p_3} : s_3,\ \mathsf{p_3} : r_2,\ \mathsf{p_5} : s_4,\ \mathsf{p_3} : r_4,$$
$$\mathsf{p_4} : r_3,\ \mathsf{p_3} : s_5,\ \mathsf{p_4} : s_6,\ \mathsf{p_1} : r_5,\ \mathsf{p_3} : r_6,\ \mathsf{p_1} : s_7,\ \mathsf{p_5} : s_8$$

By swapping, e.g., events $\mathsf{p_2} : s_2$ and $\mathsf{p_3} : s_3$ we get another interleaving in $\mathsf{sched}(\tau)$, and so forth. Note that $\mathsf{p_2} : s_2$ and $\mathsf{p_3} : s_3$ are independent since $\mathsf{p_3} : s_3 \prec_\tau \mathsf{p_3} : r_2$.

In the following, we assume that the actions of a process are uniquely determined by the order of its receive events (equivalently, by the order in which messages are delivered to this process). To be more precise, given a sequence of messages delivered to a given process, the actions of this process are deterministic except

for the choice of fresh identifiers for the pids of spawned processes and the tags of sent messages, which has no impact on the outcome of the execution. Therefore, if we have two executions of a program where each process receives the same messages and in the same order, and perform the same number of steps, then the computations will be the same (identical, if we assume that the same process identifiers and message tags are chosen).

The following notion of *subtrace* is essential to characterize message races:

Definition 6 (subtrace). *Given traces τ, τ' with the same initial pid, we say that τ' is a* subtrace *of τ, in symbols $\tau' \ll \tau$, iff for all pid p in τ, τ' we have that the sequence $\tau'(p)$ is a prefix of the sequence $\tau(p)$.*

Intuitively speaking, we can obtain a subtrace by deleting the final actions of some processes. However, note that actions cannot be arbitrarily removed since the resulting mapping must still be a trace (i.e., all linearizations must still be interleavings according to Definition 1). For instance, this prevents us from deleting the sending of a message whose corresponding receive is not deleted.

Let us conclude this section with a *declarative* notion of message race:

Definition 7 (message race). *Let τ be a trace with $\tau(p) = a_1, \ldots, a_n$ and $a_i = \mathsf{rec}(\ell, cs)$, $1 \leq i \leq n$. There exists a message race between ℓ and ℓ' in τ iff there is a subtrace $\tau' \ll \tau$ such that $\tau'(p) = a_1, \ldots, a_{i-1}$ and $\tau'[p \mapsto a_1, \ldots, a_{i-1}, \mathsf{rec}(\ell', cs)]$ is a trace.*

Informally speaking, we have a message race whenever we have an execution which is a prefix of the original one up to the point where a different message is received.

Consider, e.g., the trace in Example 4. If we assume that the value of message ℓ_4 matches the constraints cs_2 of receive $r_2 = \mathsf{rec}(\ell_2, cs_2)$, i.e., $\mathsf{match}(v_4, cs_2) = \mathit{true}$, we have a race between ℓ_2 and ℓ_4 since we have the following subtrace τ'

$$[\ \mathsf{p1} \mapsto \mathsf{spawn}(\mathsf{p3}), \mathsf{spawn}(\mathsf{p2}), \mathsf{spawn}(\mathsf{p4}), \mathsf{spawn}(\mathsf{p5}), \cancel{r_5}, \cancel{s_1};\quad \mathsf{p2} \mapsto s_2;$$
$$\mathsf{p3} \mapsto r_1, s_3, \cancel{r_2}, \cancel{r_4}, \cancel{s_5}, \cancel{r_6};\quad \mathsf{p4} \mapsto r_3;\quad \mathsf{p5} \mapsto s_1, s_4, s_8\ ]$$

and $\tau'[\mathsf{p3} \mapsto r_1, s_3, \mathsf{rec}(\ell_4, cs_2)]$ is a trace according to Definition 4.

4 Computing Message Races and Race Variants

In this section, we introduce constructive definitions for computing message races and *race variants* from a given execution trace. Intuitively speaking, once we identify a message race, a race variant is a *partial* trace that can be used to drive the execution of a program so that a new interleaving which is not causally equivalent to the previous one is obtained. Computing message races and race variants are essential ingredients of a systematic state-space exploration tool.

First, we introduce the notion of *race set* that, given a trace τ and a message ℓ that has been received in τ, computes all the messages that race with ℓ in τ for the same receive (if any). It is worthwhile to note that race sets are defined on traces, i.e., message races do not depend on a particular interleaving but on the class of causally equivalent interleavings represented by a trace.

Definition 8 (race set). *Let τ be a trace with $e_r = (p{:}\mathsf{rec}(\ell, cs)) \in \tau$. Consider a message $\ell' \neq \ell$ with $e'_s = (p'{:}\mathsf{send}(\ell', v', p)) \in \tau$ such that $\mathsf{match}(v', cs) = true$. We say that messages ℓ and ℓ' race for e_r in τ if*

- *e_r does not happen before e'_s, i.e., $e_r \not\rightsquigarrow_\tau e'_s$;*
- *for all event $e''_s = (p' : \mathsf{send}(\ell'', v'', p)) \in \tau$ such that $e''_s \prec_\tau e'_s$ either $\mathsf{match}(v'', cs) = false$ or there exists an event $(p : \mathsf{rec}(\ell'', cs'')) \in \tau$ with $(p{:}\mathsf{rec}(\ell'', cs'')) \prec_\tau (p{:}\mathsf{rec}(\ell, cs))$.*

We let $\mathsf{race\_set}_\tau(\ell)$ denote the set all messages that race with ℓ in τ.

Intuitively speaking, the definition above requires the following conditions for messages ℓ and ℓ' to race for a receive statement e_r:

1. The target of both messages must be the same (p) and their values should match the constraint cs in e_r (note that we already know that the value of message ℓ matches the constraint of e_r since τ is a trace).
2. The original receive event, e_r, cannot happen before the sending event e'_s of message ℓ'. Otherwise, we had a dependency and removing e_r would prevent e'_s to happen (in a well-formed trace).
3. Finally, we should check that there are no other messages sent by the same process (and to the same target) that match the constraint cs and have not been received before e_r (since, in this case, the first of such messages would race with ℓ instead).

Given a trace τ and a receive event $p : \mathsf{rec}(\ell, cs) \in \tau$, a naive algorithm for computing the associated race set, $\mathsf{race\_set}_\tau(\ell)$, can proceed as follows:

- First, we identify the set of events of the form $p' : \mathsf{send}(\ell', v', p)$ in τ with $\ell' \neq \ell$, i.e., all *send* events where the target process is p and the message tag is different from ℓ.
- Now, we remove from this set each *send* event $p' : \mathsf{send}(\ell', v', p)$ where $p : \mathsf{rec}(\ell, cs) \rightsquigarrow_\tau p'{:}\mathsf{send}(\ell', v', p)$.
- We also remove the events $p'{:}\mathsf{send}(\ell', v', p)$ where $\mathsf{match}(v', cs) = false$.
- Finally, for each subset of *send* events from the same process, we select (at most) one of them as follows. We check for each *send* event (starting from the oldest one) whether there is a corresponding *receive* event in p which precedes $p : \mathsf{rec}(\ell, cs)$. The message tag of the first *send* event without a corresponding *receive* (if any) belongs to the race set, and the remaining ones (from the same process) can be discarded.

Example 5. Consider again the trace τ from Example 4. Let us focus on the second receive event of process p3, denoted by r_2. Here, we have five (other) messages with the same target (p3): ℓ_1, ℓ_4, ℓ_6, ℓ_7 and ℓ_8. Let us further assume that the values of all messages match the constraint of r_2 except for message ℓ_4. Let us analyze each message separately:

- Message ℓ_1 is excluded from the message race since there exists a corresponding receive event, r_1, and $r_1 \prec_\tau r_2$. Hence, $\ell_1 \notin \mathsf{race\_set}_\tau(\ell_2)$.

- As for message ℓ_4, we trivially have $r_2 \not\rightsquigarrow_\tau s_4$. Moreover, there is a previous event, s_1, from p5 to p3 but it has already been received (r_1). However, we assumed that the value of message ℓ_4 does not match the constraints of r_2 and, thus, $\ell_4 \notin \mathsf{race\_set}_\tau(\ell_2)$.
- Consider now message ℓ_6. At first sight, it may seem that there is a dependency between r_2 and the sending event s_6 (since message ℓ_2 was delivered before s_3). However, this is not the case since event s_3 happened before r_2 and, thus, $r_3 \rightsquigarrow_\tau s_6$ but $r_2 \not\rightsquigarrow_\tau r_3$. Moreover, there are no previous send events in p4 and, thus, $\ell_6 \in \mathsf{race\_set}_\tau(\ell_2)$.
- Regarding message ℓ_7, we have $r_2 \rightsquigarrow_\tau s_7$ since $r_2 \rightsquigarrow_\tau s_5$, $s_5 \rightsquigarrow_\tau r_5$ and $r_5 \rightsquigarrow_\tau s_7$. Therefore, messages ℓ_2 and ℓ_7 cannot race for r_2 and $\ell_7 \notin \mathsf{race\_set}_\tau(\ell_2)$.
- Finally, consider message ℓ_8. Trivially, we have $r_2 \not\rightsquigarrow_\tau s_8$. Now, we should check that all previous sent messages (ℓ_1 and ℓ_4) have been previously received or do not match the constraints of e_r, which is the case. Therefore, $\ell_8 \in \mathsf{race\_set}_\tau(\ell_2)$.

Hence, we have $\mathsf{race\_set}_\tau(\ell_2) = \{\ell_6, \ell_8\}$.

As mentioned before, computing message races can be useful to identify alternative executions which are not causally equivalent to the current one. Ideally, we want to explore only one execution (interleaving) per equivalence class (trace). For this purpose, we introduce the notion of *race variant* which returns a (typically partial) trace, as follows:

Definition 9 (race variant). *Let $\tau[p \mapsto A;\mathsf{rec}(\ell, cs);A']$ be a trace with $\ell' \in \mathsf{race\_set}_\tau(\ell)$. The* race variant *of τ w.r.t. ℓ and ℓ', in symbols $\mathsf{variant}_\tau(\ell, \ell')$, is given by the (possibly partial) trace*

$$\mathsf{rdep}(A', \tau[p \mapsto A;\mathsf{rec}(\ell', cs)])$$

where the auxiliary function rdep *is inductively defined as follows:*

$$\mathsf{rdep}(A, \tau) = \begin{cases} \tau & \text{if } A = \epsilon \\ \mathsf{rdep}(A', \tau) & \text{if } A = \mathsf{rec}(\ell, cs);A' \\ \mathsf{rdep}(A';A'', \tau[p \mapsto \epsilon]) & \text{if } A = \mathsf{spawn}(p);A',\ \tau(p) = A'' \\ \mathsf{rdep}(A';A^*, \tau[p \mapsto A'']) & \text{if } A = \mathsf{send}(\ell, v, p);A',\ \tau(p) = A'';\mathsf{rec}(\ell, cs);A^* \\ \mathsf{rdep}(A', \tau) & \text{if } A = \mathsf{send}(\ell, v, p);A',\ \mathsf{rec}(\ell, cs) \notin \tau(p) \end{cases}$$

Intutively speaking, $\mathsf{variant}_\tau(\ell, \ell')$ removes the original receive action $\mathsf{rec}(\ell, cs)$ from τ as well as all the actions that depend on this one (according to the happened-before relation). Then, it adds $\mathsf{rec}(\ell', cs)$ in the position of the original receive.

Example 6. Consider again the execution trace τ from Example 4, together with the associated race set computed in Example 5: $\mathsf{race\_set}_\tau(\ell) = \{\ell_6, \ell_8\}$.

Let us consider ℓ_6. Here, the race variant $\mathsf{variant}_\tau(\ell_2, \ell_6)$ is computed from $\mathsf{rdep}((r_4, s_5, r_6),\ \tau[\mathsf{p3} \mapsto r_1, s_3, \mathsf{rec}(\ell_6, cs)])$ as follows:

$$\begin{array}{ll}
& \mathsf{rdep}((r_4, s_5, r_6),\ \tau[\mathsf{p3} \mapsto r_1, s_3, \mathsf{rec}(\ell_6, cs)]) \\
= & \mathsf{rdep}((s_5, r_6),\ \tau[\mathsf{p3} \mapsto r_1, s_3, \mathsf{rec}(\ell_6, cs)]) \\
= & \mathsf{rdep}((r_6, s_7),\ \tau[\mathsf{p1} \mapsto \mathsf{spawn}(\mathsf{p3}), \mathsf{spawn}(\mathsf{p2}), \mathsf{spawn}(\mathsf{p4}), \mathsf{spawn}(\mathsf{p5}); \\
& \qquad\qquad\ \mathsf{p3} \mapsto r_1, s_3, \mathsf{rec}(\ell_6, cs)]) \\
= & \mathsf{rdep}((s_7),\ \ \ \tau[\mathsf{p1} \mapsto \mathsf{spawn}(\mathsf{p3}), \mathsf{spawn}(\mathsf{p2}), \mathsf{spawn}(\mathsf{p4}), \mathsf{spawn}(\mathsf{p5}); \\
& \qquad\qquad\ \mathsf{p3} \mapsto r_1, s_3, \mathsf{rec}(\ell_6, cs)]) \\
= & \mathsf{rdep}(\epsilon,\ \ \ \ \ \tau[\mathsf{p1} \mapsto \mathsf{spawn}(\mathsf{p3}), \mathsf{spawn}(\mathsf{p2}), \mathsf{spawn}(\mathsf{p4}), \mathsf{spawn}(\mathsf{p5}); \\
& \qquad\qquad\ \mathsf{p3} \mapsto r_1, s_3, \mathsf{rec}(\ell_6, cs)])
\end{array}$$

Therefore, the computed race variant τ' is as follows:

$$\begin{array}{l}
[\ \mathsf{p1} \mapsto \mathsf{spawn}(\mathsf{p3}), \mathsf{spawn}(\mathsf{p2}), \mathsf{spawn}(\mathsf{p4}), \mathsf{spawn}(\mathsf{p5});\quad \mathsf{p2} \mapsto s_2; \\
\ \ \mathsf{p3} \mapsto r_1, s_3, \mathsf{rec}(\ell_6, cs);\quad \mathsf{p4} \mapsto r_3;\quad \mathsf{p5} \mapsto s_1, s_4, s_8\ \ \ ]
\end{array}$$

In the following, given traces τ, τ', if τ is a subtrace of τ', i.e., $\tau \ll \tau'$, we also say that τ' *extends* τ. Let us consider a trace τ and one of its race variants τ'. The next result states that there is no interleaving in $\mathsf{sched}(\tau'')$ that is causally equivalent to any interleaving of $\mathsf{sched}(\tau)$ for all traces τ'' that extend the race variant τ'. This is an easy but essential property to guarantee the optimality in the number of variants considered by a state-space exploration algorithm.

Theorem 3. *Let τ be a trace with $e_r = (p{:}\mathsf{rec}(\ell, cs)) \in \tau$ and $\ell' \in \mathsf{race\_set}_\tau(\ell)$. Let $\tau' = \mathsf{variant}_\tau(\ell, \ell')$ be a race variant. Then, for all trace τ'' that extends τ' and for all interleavings $S \in \mathsf{sched}(\tau)$ and $S'' \in \mathsf{sched}(\tau'')$, we have $S \not\approx S''$.*

Proof. Consider first a (possibly partial) trace τ_1 obtained from $\mathsf{rdep}(A', \tau[p \mapsto A;\mathsf{rec}(\ell, cs)])$, i.e., τ_1 is equal to the race variant except for the fact that we have not changed yet the considered receive event. Then, it is easy to see that τ_1 is a subtrace of τ, $\tau_1 \ll \tau$, since rdep just follows the happened-before relation in order to consistently remove all dependences of e_r. Note that τ_1 and τ' only differ in the receive event ($\mathsf{rec}(\ell, cs)$ in τ_1 and $\mathsf{rec}(\ell', cs)$ in τ'). Trivially, for all interleavings $S_1 \in \mathsf{sched}(\tau_1)$ and $S' \in \mathsf{sched}(\tau')$, we have $S \not\approx S'$ since the receive events $\mathsf{rec}(\ell, cs)$ and $\mathsf{rec}(\ell', cs)$ can only happen in one of the interleavings but not in both of them. Moreover, for all trace τ'' that extends τ', and for all interleavings $S'' \in \mathsf{sched}(\tau'')$ and $S \in \mathsf{sched}(\tau)$, we have $S'' \not\approx S$ since they will always differ in the receive events above. □

The definitions of message race and race variant can be used as the kernel of a state-space exploration technique that proceeds as follows:

1. First, a random execution of the program is considered, together with its associated trace.
2. This trace is used to compute message races (if any) as well as the corresponding race variants.

3. Then, each computed race variant is used to drive the execution of the program up to a given point, then continuing the execution nondeterministically according to the standard semantics. We gather the traces of these executions and the process starts again until all possible executions have been explored.

A formalization of such an algorithm can be found in the context of reachability testing [21] using SYN-sequences instead of traces.

On the other hand, the *prefix-based tracing* technique for Erlang introduced in [15] could be useful to instrument programs with a (possibly partial) trace so that their execution follows this trace and, then, continues nondeterministically, eventually producing a trace of the complete execution (point 3 above). The notion of trace in [15] is different to our notion of trace, though: message delivery is explicit and traces do not include message values nor receive constraints. Nevertheless, adapting their developments to our traces would not be difficult.

The definitions of message race and race variant could also be useful in the context of causal-consistent replay debugging [19,20]. First, we note that our traces could be straightforwardly used for replay debugging since they contain strictly more information than the logs of [19,20]. However, in contrast to the original logs, our traces would allow the replay debugger CauDEr [10] to also show the message races in a particular execution, and then let the user to replay any selected race variant, thus improving the functionality of the debugger. Some ongoing work along these lines can be found in [14]. However, the traces considered in [14] are similar to those in [15] (i.e., they have explicit events for message delivery and skip message values and receive constraints). As a consequence, the races considered in [14] are only *potential* races since there are no guarantees that message values in these races actually match the corresponding receive constraints. Nevertheless, an extension of CauDEr using our traces and the associated definitions of message race and race variant could be defined following a similar scheme.

5 Discussion and Future Work

The closest approach to our notion of trace are the *logs* of [19,20], which where introduced in the context of causal-consistent *replay* debugging for a message-passing concurrent language. In this work, we have extended the notion of log with enough information so that message races can be computed. Indeed, this work stemmed from the idea of improving causal-consistent replay debugging [19,20] with the computation of message races, since this information might be useful for the user in order to explore alternative execution paths. A first implementation in this direction is described in [14], although the traces are slightly different, as discussed above.

Another close approach is that of *reachability testing*, originally introduced in [16] in the context of multithreaded programs that perform read/write operations. This approach was then extended to message-passing programs in [26,22]

and later improved and generalized in [21].[6] The notion of *SYN-sequence* in reachability testing (and, to some extend, the *program executions* of [8]) share some similarities with our traces since both represent a partial order with the actions performed by a number of processes running concurrently (i.e., they basically denote a *Mazurkiewicz trace* [23]). Nevertheless, our traces are tailored to a language with selective receives by adding message values and receive constraints ([21], in contrast, considers different ports for receive statements). Moreover, to the best of our knowledge, these works have not considered a language where processes can be dynamically spawned, as we do.

Both reachability testing and our approach share some similarities with so-called *stateless model checking* [12]. The main difference, though, is that stateless model checking works with interleavings. Then, since many interleavings may boil down to the same Mazurkiewicz trace, *dynamic partial order reduction* (DPOR) techniques are introduced (see, e.g., [11,1]). Intuitively speaking, DPOR techniques aim at producing only one interleaving per Mazurkiewicz trace. Computing message races is more natural in our context thanks to the use of traces, since DPOR techniques are not needed. Concuerror [5] implements stateless model checking for Erlang [1,4], and has been recently extended to also consider *observational equivalence* [3], thus achieving a similar result as our technique regarding the computation of message races, despite the fact that the techniques are rather different (using traces vs using interleavings + DPOR).

Another, related approach is the detection of race conditions for Erlang programs presented in [6]. However, the author focuses on data races (that may occur when using some shared-memory built-in operators of the language) rather than message races. Moreover, the detection is based on a *static* analysis, while we consider a *dynamic* approach to computing message races.

To conclude, we have introduced appropriate notions of interleaving and trace that are useful to represent concurrent executions in a message-passing concurrent language with dynamic process spawning and selective receives. In particular, our notion of trace is essentially equivalent to a Mazurkiewicz trace, thus allowing us to represent all causally equivalent interleavings in a compact way. Despite the simplicity of traces, they contain enough information to analyze some common error symptoms (e.g., orphan messages) and to compute message races, which can then give rise to alternative executions (specified by so-called race variants, i.e., partial traces).

As for future work, we will consider the computation of message races from *incomplete* traces, since it is not uncommon that concurrent programs are executed in an endless loop and, thus, the associated traces are in principle infinite. We also plan to extend the traces with more events (like message *deliver* and process *exit*) so that they can be used to detect more types of error symptoms (like process deadlocks and lost or delayed messages).

Finally, another interesting line of research involves formalizing and implementing an extension of the causal-consistent replay debugger CauDEr [10] for

[6] [24] also deals with message-passing concurrent programs, but only *blocking* send and receive statements are considered.

Erlang in order to also show message races (our original motivation for this work). A preliminary approach along these lines can be found in [14], though the considered traces are slightly different, as mentioned above. In this context, we also plan to analyze efficiency issues and investigate the definition of efficient algorithms for computing race sets.

Acknowledgements. The author would like to thank Juan José González-Abril for his useful remarks on a preliminary version of this paper. I would also like to thank the anonymous reviewers 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. Akka website. URL: `https://akka.io/` (2021)
3. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Beyer, D., Huisman, M. (eds.) Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018). Lecture Notes in Computer Science, vol. 10806, pp. 229–248. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_14
4. Aronis, S., Sagonas, K.: The shared-memory interferences of erlang/otp built-ins. In: Chechina, N., Fritchie, S.L. (eds.) Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang. pp. 43–54. ACM (2017). https://doi.org/10.1145/3123569.3123573
5. 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
6. Christakis, M., Sagonas, K.: Static Detection of Race Conditions in Erlang. In: Carro, M., Peña, R. (eds.) Proc. of the International Symposium on Practical Aspects of Declarative Languages (PADL 2010). pp. 119–133. Springer (2010)
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986). https://doi.org/10.1145/5397.5399
8. Cypher, R., Leu, E.: Efficient race detection for message-passing programs with nonblocking sends and receives. In: Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing (SPDP 1995). pp. 534–541. IEEE (1995), `https://doi.org/10.1109/SPDP.1995.530730`
9. Erlang website. URL: `https://www.erlang.org/` (2021)
10. Fabbretti, G., González-Abril, J.J., Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr website. URL: `https://github.com/mistupv/cauder` (2021)
11. 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
12. Godefroid, P.: Model checking for programming languages using verisoft. In: POPL. pp. 174–186 (1997). https://doi.org/10.1145/263699.263717

13. Go website. URL: https://go.dev/ (2021)
14. González-Abril, J.J., Vidal, G.: A lightweight approach to computing message races with an application to causal-consistent reversible debugging. CoRR **abs/2112.12869** (2021), https://arxiv.org/abs/2112.12869
15. González-Abril, J.J., Vidal, G.: Prefix-based tracing in message-passing concurrency. In: Angelis, E.D., Vanhoof, W. (eds.) Proceedings of the 31st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2021). Lecture Notes in Computer Science, vol. 13290, pp. 157–175. Springer (2021), https://doi.org/10.1007/978-3-030-98869-2_9
16. Hwang, G., Tai, K., Huang, T.: Reachability testing: an approach to testing concurrent software. In: Proceedings of the First Asia-Pacific Software Engineering Conference (APSEC 1994). pp. 246–255. IEEE (1994), https://doi.org/10.1109/APSEC.1994.465255
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563
18. 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
19. 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
20. 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
21. 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
22. Lei, Y., Tai, K.: Efficient reachability testing of asynchronous message-passing programs. In: Proceedings of the 8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002). p. 35. IEEE Computer Society (2002), https://doi.org/10.1109/ICECCS.2002.1181496
23. 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
24. Netzer, R.H., Miller, B.P.: Optimal tracing and replay for debugging message-passing parallel programs. J. Supercomput. **8**(4), 371–388 (1995), https://doi.org/10.1007/BF01901615
25. Rust website. URL: https://www.rust-lang.org/ (2021)
26. Tai, K.: Reachability testing of asynchronous message-passing programs. In: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 1997). pp. 50–61. IEEE Computer Society (1997), https://doi.org/10.1109/PDSE.1997.596826
27. Undo Software: The Business Value of Optimizing CI Pipelines (2020), https://info.undo.io/ci-research-report