The Shared-Memory Interferences of Erlang/OTP Built-Ins

Stavros Aronis
Department of Information Technology
Uppsala University
Uppsala, Sweden
stavros.aronis@it.uu.se

Konstantinos Sagonas
Department of Information Technology
Uppsala University
Uppsala, Sweden
kostis@it.uu.se

Abstract

Erlang is a concurrent functional language based on the actor model of concurrency. In the purest form of this model, actors are realized by processes that do not share memory and communicate with each other exclusively via message passing. Erlang comes quite close to this model, as message passing is the primary form of interprocess communication and each process has its own memory area that is managed by the process itself. For this reason, Erlang is often referred to as implementing "shared nothing" concurrency. Although this is a convenient abstraction, in reality Erlang's main implementation, the Erlang/OTP system, comes with a large number of built-in operations that access memory which is shared by processes. In this paper, we categorize these built-ins, and characterize the interferences between them that can result in observable differences of program behaviour when these built-ins are used in a concurrent setting. The paper is complemented by a publicly available suite of more than one hundred small Erlang programs that demonstrate the racing behaviour of these built-ins.

CCS Concepts • **Software and its engineering → Concurrent programming languages**; *Software testing and debugging*;

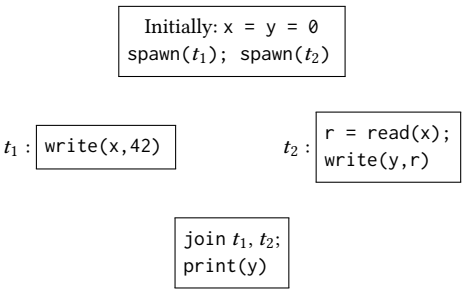
Keywords Actors, BEAM, Concuerror, Erlang, Scheduling nondeterminism

ACM Reference Format:
Stavros Aronis and Konstantinos Sagonas. 2017. The Shared-Memory Interferences of Erlang/OTP Built-Ins. In *Proceedings of 16th ACM SIGPLAN International Workshop on Erlang, Oxford, UK, September 8, 2017 (Erlang'17),* 12 pages.
https://doi.org/10.1145/3123569.3123573

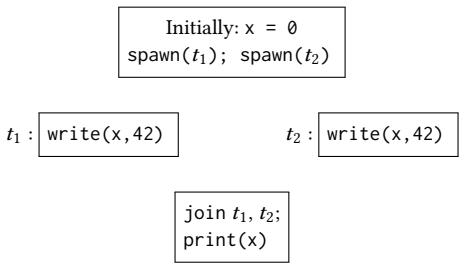
1 Introduction

For a language famous for its "shared nothing" approach to concurrency [3], Erlang's main implementation, the Erlang/OTP system[1], comes with a large number of built-in operations that depend on and affect shared memory. This is not surprising, as it is impossible to write any interesting concurrent program without some interaction between processes. Even if this interaction consists of sending a message from one process to another, at the virtual machine (VM) level this means that the send operation needs to write to some memory that is not local to the process that executes the send, namely to the recipient's mailbox. So, even in pure message-passing concurrent Erlang programs, *some* shared memory accesses do take place. Operations accessing shared memory are described in various places in Erlang/OTP's documentation, but the interferences between them have so far escaped any form of complete specification, either formal or informal. Nevertheless, such operations are widely used in most Erlang programs and their interference can result in non-deterministic and often unexpected and unintended —if not erroneous— behaviours in a multi-threaded setting.

In low-level languages that implement concurrency via shared memory, e.g., C/pthreads, conflicts between instructions that lead to races are well-known to programmers. The most common definition specifies that two memory accesses *conflict* if they access the same memory location and at least one of them is a `write`. So, the following program, consisting of a *main* thread and two threads t_1 and t_2 executing concurrently, has two different outcomes based on whether the `read` takes place before the `write` to `x` or not.



Even though two `write` operations on the same global variable conflict with each other, sometimes their interference is not observable. Such is the case for the program below, which will print 42 independently of which `write` executes first.



This example also shows that sometimes the observability of conflicts between operations are conditional on *values* that these operations contain. In this paper, we are interested to characterize *observable* interferences of Erlang/OTP built-ins; not interference between racing operations that cannot be observed in programs.

[1] By "Erlang/OTP" we refer to the latest released version of the Erlang/OTP implementation at the time of writing this paper, which is version 19.3 (`git SHA: a748caf`).

Erlang'17, September 8, 2017, Oxford, UK

ACM ISBN 978-1-4503-5179-9/17/09…$15.00
https://doi.org/10.1145/3123569.3123573

The examples we presented have Erlang counterparts. Read and write operations to shared data exist in lookups and inserts to `public` ETS tables. The following Erlang program has two different outcomes (`[]` and `[{key,42}]`) depending on whether the `lookup` operation is executed before or after the `insert` operation.

```
register(main,self()),
ets:new(tab,[public,named_table]),
spawn(p1), spawn(p2)
```

```
p1 : ets:insert(tab,{key,42})
```

```
p2 : L = ets:lookup(tab,key),
main ! L
```

```
receive M -> erlang:display(M) end
```

Asynchronous message passing is another source of non-determinism in Erlang. The following program will print different results depending on which send operation wins the race of delivering its message first on *main*'s mailbox.

```
register(main,self()),
spawn(p1), spawn(p2)
```

```
p1 : main ! 17
```

```
p2 : main ! 42
```

```
receive M -> erlang:display(M) end
```

However, in Erlang, the above races (i.e., accessing shared memory and message passing) are not the only reasons for concurrent execution leading to differences in observable behaviour. Besides calls to built-in operations, there are also asynchronous *events*, like a process exiting, that can cause programs to have different results. Consider the following program:

```
spawn(p1), spawn(p2)
```

```
p1 : register(name,self()),
...
```

```
p2 : register(name,self()),
...
```

```
whereis(name)
```

Here, two processes, $p1$ and $p2$, race in registering their process identifier under the same name, then do some other operations (not shown) and exit. If the chronologically second `register` takes place while the other process is still alive, the process executing the second `register` will crash. Actually, the `whereis` call in this program does not have just two outcomes but three ($p1$, $p2$, or the atom `undefined` if it is scheduled either before the registration attempts or right after one or both processes that have successfully registered exit). Its observable behaviour is not influenced only by built-in operations that are present in the program's code but also by events (processes exiting) that are implicit here. Our aim in this paper is to categorize the shared-memory interferences of Erlang/OTP built-ins and events that occur in an Erlang node and result in observable differences in behaviour of Erlang programs.

More specifically, in this paper we present:

- A logical decomposition of the Erlang/OTP runtime system into components that support a particular functionality, and access, in a racing manner, particular parts of shared memory. We assign labels to such components, and broadly describe the functionality and data used within them.
- A decomposition of the message passing and fault tolerance mechanisms into asynchronous *events*, that are comparable with built-in operations for describing interferences.
- Some indicative examples that show ways in which built-in operations and events interfere in an observable way.
- A characterization of all built-in operations and events, based on the components they involve. The characterization assigns at least one common component label to each member of any pair of built-in operations or events that can interfere in an observable way.
- A relatively complete account of all pairs of built-in operations and events that interfere, accompanied by a publicly available collection of small Erlang programs that result in different outcomes, depending on how specific pairs of operations and events are ordered during their execution.

2 Built-In Operations and Events

Erlang programs make heavy use of operations that are considered to be provided by the language's implementation; any primitive operation, such as an arithmetic calculation or construction and decomposition of composite values (lists, tuples, maps, etc.) can be seen as a *built-in operation* (also simply called a *built-in*). As Erlang is a functional programming language in its core, a large number of built-ins is intended for purely functional computations. Such built-ins operate on data that is always process-local and therefore do not cause interference between concurrently executing processes.

Most Erlang programs, however, are designed to involve more than one process. Any kind of cooperation between concurrent processes is achieved via the use of other built-in operations that access some shared memory, either data shared directly between processes (e.g., stored in ETS tables or in other system-wide data structures). A straightforward way to explain the behaviour of a concurrent Erlang program is to determine the relative order in which interfering built-in operations are executed. In this paper, we expose the interference between built-ins that operate on shared data through a publicly available set of small programs that have different results based on the order of execution of such built-ins.

We also introduce a number of *events*, that are comparable with built-in operations, and are considered to either be executed by processes (e.g., in the case of process termination), or by other independent entities (e.g., conceptual "message queues" in the case of message delivery, as explained in the next paragraph). Events can also be placed in an ordering together with built-in operations that involve shared data. The reason for inclusion of events is that asynchronous message passing and mechanisms that support fault tolerance add complex ways in which processes interact, which do not always directly correspond to the execution of a built-in operation by a process. For example, as we saw, the termination of execution of a process is directly associated with a number of implicit effects that can affect the state of an Erlang VM (e.g., unregistering a process name, sending signals to linked processes, etc.) and therefore other processes in it.

The limited guarantees of asynchronous message passing make an even stronger case for using events for some observable behaviours in a node. As stated in the Academic and Historical FAQ of Erlang [7]:

> *"If there is a live process and [another process sends] it message A and then message B, it's guaranteed that if message B arrived, message A arrived before it."*

An intuitive way to think about this behaviour is to consider a send operation as placing a message in a queue that is unique for each sender-recipient pair. Messages in each such queue will be delivered to the mailbox of the recipient in order. However, there are no ordering guarantees for messages from different queues.

We explain the need of message passing events via one more example. Consider the following program.

```
register(main,self()),
register(p1,spawn(p1)),
spawn(p2)
```

$p1$:
```
receive M -> main ! M end
```

$p2$:
```
main ! first,
p1 ! second
```

```
receive M -> erlang:display(M) end
```

This program corresponds directly to the following passage of the FAQ of the language [7]:

> *"On the other hand, imagine processes P, Q and R. P sends message A to Q, and then message B to R. There is no guarantee that A arrives before B."*

Let's change the names to the ones used in the example:

> *"On the other hand, imagine processes p2, main and p1. p2 sends message* `first` *to main, and then message* `second` *to p1. There is no guarantee that* `first` *arrives before* `second`*."*

Simply sending two messages to two different processes does not directly result in differences in observable behaviour. To create a situation where the difference is observable, our example goes a step further: process $p1$ forwards message `second` to *main*, thereby creating the potential for a message delivery race in *main*'s mailbox.

Based only on the message passing guarantees that Erlang's documentation provides, the above program has two possible outcomes: `first` and `second`. On the other hand, at any point in time, the Erlang/OTP system makes —and in fact needs to make— some concrete decisions about implementation choices that Erlang's documentation allows. For example, it is currently the case that when both the sender and the recipient are on the same Erlang node, a message is placed in the recipient's mailbox (if it is alive) before the sender's call to the send operation returns. So, when this particular program runs on a single node, the `second` send does not even begin executing before the `first` send has delivered its message. So, *currently*, only one of the two possible behaviours is observable when running on a single Erlang node, but both behaviours are possible when each process is running on different distributed Erlang nodes.

But the key word in the previous sentence is "currently". As far as we know, there is nothing in Erlang/OTP's documentation that guarantees that the send operation returns only after the message has been delivered (or after the VM has discovered that the message cannot be delivered). In contrast, the documentation explicitly refers to the send operation as being *asynchronous*. Events are suitable for capturing such asynchronous behaviour of built-ins. For example, we will employ a msg_deliver event to disconnect the point in time when the send built-in gets called with the time when the message is actually delivered to the recipient's mailbox. In a nutshell, events are a mechanism that will allow us to consider observable behaviours that come from VM actions that are not present in the code or from effects of built-ins that do not happen in an order that agrees with the order that calls to these built-ins are executed (as e.g., the two sends from $p2$ in the last example).

3 Erlang/OTP Components

In this section, we broadly describe an Erlang/OTP system as a collection of components. We also introduce a number of **labels** that we will use to annotate all built-in operations and events that interfere with each other (the complete set of labels appears in Table 17 in Appendix A). For easier reference, we list all labels introduced in each subsection alongside its title.

3.1 Processes and Nodes (Labels: process, registry, leader)

Erlang programs are composed of a number of *processes*, which run on runtime systems called *nodes*. On each node any number of processes may be created ("spawned") dynamically. Each process on a node has a unique *process identifier (PID)*, which is a value of a dedicated datatype and can be used to e.g., send messages to it. Processes can communicate using just PIDs, even across nodes.

Each Erlang process evaluates a function that is specified when the process is spawned. The process continues execution until it has fully evaluated this function. Whenever a process terminates ("exits") an *exit reason* is given: if the termination is normal then the reason is the atom `'normal'`; if any other value is given as a reason, the termination is considered abnormal. A process may exit abnormally if e.g., an exception is raised and not caught by the code that this process runs during its execution. After termination, the process is considered to be "dead"[2]. As mentioned, a process exiting is treated as as an event, and the exact steps that this event comprises are explained in detail in Section 3.5, after all other components have been presented.

Process Table The use of processes already requires operations that access shared memory, such as a mechanism to assign unique PIDs, an operation that checks whether a particular PID corresponds to a process that has exited, an operation that reports how many processes exist on a node, etc. We will use the label **process** for such operations. A per-node data structure, called *process table*, is the shared data structure that is accessed by these operations.

Process Registry Processes that need to be easily discoverable by other processes (e.g., because they handle some node-wide capability, such as connecting to other nodes) can use another shared memory data structure, the *process registry*, to associate their PID with an atom, which can be viewed as a "name" for the process.

[2] In the Erlang VM, PIDs of processes that have exited can be reused for subsequently created new processes. However, the range of PIDs is big enough for their recycling to not cause any confusion to programs in practice. Also, there exists an (adjustable) limit to the number of processes that can simultaneously be alive in a node, which is by default high enough to also be irrelevant in practice; i.e., we will assume that the VM never runs out of PIDs and new processes can always be spawned.

Such names can then be used in the place of PIDs for some operations. Operations that register or unregister such names or lookup a name in the registry can interfere, as e.g., a process cannot have more than one names and two processes cannot be registered under the same name. We will use the label **registry** for such operations.

Group Leader Every process is also member of a process group, designated by a *group leader*. I/O operations to standard streams (i.e., stdout and stderr) from processes in the group are designed to be handled by the group leader. When a new process is spawned, it inherits the same group leader as its parent. We will use the label **leader** for operations that manipulate leader information.

Node Each node maintains information about all other nodes that it can communicate with. Node discovery is usually implicit, whenever an operation involving processes on a different node is attempted, such as sending a message. Any process can stop the node it is running on by executing a `halt` operation. Such an operation interferes with all I/O operations from the node, with messages and signals to processes in other nodes, and with monitoring operations by other nodes. Since this paper focuses on shared-memory (i.e., intra-node) interferences, we will not consider effects that are related to the existence of other nodes.

3.2 Messages and Signals (Labels: mailbox, signal, link, timer)

The message and signal passing capabilities involve sending and receiving messages and signals, "trapping" signals, and setting conditions where messages or signals are automatically sent.

Message and Signal Sending and Delivery Processes can send messages to each other freely, using PIDs or registered names. If the PID of a dead process is used, sending fails silently; if a name that does not correspond to a registered process is used, an exception is raised (essentially, a registry lookup operation precedes the sending, if needed).

Exit signals (or simply *signals*) are a key ingredient of Erlang's fault tolerance mechanisms. They are automatically sent from an exiting process to processes that are linked to it (explanation of links follows), but can also be sent explicitly. Exit signals include the exit reason of the exiting process and, just like those, are characterized as normal if the reason is the atom `'normal'`, and abnormal otherwise. A process can choose to *trap* exit signals delivered to it and treat them as messages; untrapped, abnormal exit signals will cause the recipient process to also exit abnormally with the same reason as the emitting process. As an exception, exit signals with the atom `'kill'` as reason are never trapped and will always cause the recipient to exit with the atom `'killed'` as exit reason.

Since message delivery is asynchronous, a call to any operation that sends a message or signal, such as the `send` or `exit` built-in operations, does not correspond to the delivery of that message or signal. We will instead use:(i) two events, named msg_deliver and sig_deliver, to denote the delivery of a message and a signal, respectively; (ii) the label **mailbox** to annotate operations that influence the mailbox of a process; (iii) the label **signal** to annotate operations that involve signal handling and operations whose interruption by the arrival of an untrapped, abnormal exit signal (that will force the process to exit) can be observable.

Message Receiving and Timeouts Erlang processes retrieve messages from their mailbox using `receive` statements that are built into the syntax of the language and support pattern matching. When a process executes such a statement, a set of patterns is compared against each message in the mailbox for a match. The oldest message that matches some pattern mentioned in the `receive` is then removed and used. As we saw in the introduction, the order in which two messages from different senders arrive can affect the result of a `receive` statement.

A `receive` statement may also have an `after` clause, indicating code that should be executed after a specified timeout has expired if no message matching any of the patterns has arrived. If no `after` clause is present or if the timeout value is the atom `'infinity'`, the process blocks until a matching message arrives. Executing a `receive` statement will also correspond to an event (as `receive` is part of the syntax of Erlang and does not look like other built-in operations) named receive, in order to be able to capture the interference of receive timeouts with built-in operations and events that place messages in a process' mailbox.

Links and Monitors Erlang processes can set *links* between them. If a link exists between two processes, when one of them exits, the exiting one will send an exit signal with the reason of its exit to the other. If a link (that is bidirectional in the sense that it can cause either process to exit, if the other exits abnormally) is not suitable, the *monitor* mechanism can be used instead. When a process exits, a special message is sent to every other process that has a monitor on it. Multiple monitors can exist from a process to another; each is identified by a unique value of the *reference* datatype (explained further in Section 3.4). Links and monitors can be canceled. It is guaranteed that no signals or messages will be delivered to the other end after canceling a particular link or monitor. Nevertheless, a link can be re-established. We will use the **link** label to annotate operations that are related to links. As we will see in Section 5.2, a label for monitors is not necessary.

Timers Finally, Erlang's implementation includes *timers*, which are a dedicated subsystem to send 'delayed' messages. Timers, like monitors, are also identified by values of type reference. For most intents and purposes, a timer is equivalent to a process whose purpose is to send a specified message at some specified time. Additionally, a process can read the time remaining in a timer, reset it or cancel it. There are notably no guarantees that a message will be in the recipient's mailbox when the timer expires[3]. We will therefore use an event, named timer_send, to denote the expiration of a timer (which has effects similar to a send operation). The label **timer** is used for all operations that involve timers.

3.3 ETS (Label: ets)

The Erlang Term Storage is a subsystem of the Erlang/OTP implementation that *"provides the ability to store very large quantities of data in an Erlang runtime system, and to have constant access time to the data"* [12]. This purpose is achieved through *ETS tables*, which correspond directly to key-value database tables, storing entries that are Erlang tuples. Such tables are always owned by a particular process and are by default destroyed when that process exits. ETS tables support a simple access rights mechanism, which controls

[3]The following quote comes from the "Erlang Run-Time System (ERTS) Reference Manual Version 8.3" [8], in the description of the `erlang:cancel_timer/2` operation:

> *"Even if the timer had expired, it does not tell you if the time-out message has arrived at its destination yet."*

Table 1. Built-ins that generate **unique** identifiers/values.

Built-ins…	…that generate
`erlang:spawn*/*`	PIDs
`erts_internal:open_port/2`	Port IDs
`ets:new/2`	ETS table IDs
`erlang:make_ref/0` `erlang:monitor/2` `erlang:send_after/*` `erlang:spawn*/*` `erlang:start_timer/*`	References
`erlang:unique_integer/*`	Unique Integers

whether other processes are allowed to read or write to them. When this happens, ETS tables are memory which is shared between processes. ETS tables have a dedicated identifier allocation mechanism, for "table identifiers" (TIDs) as well as a "naming" mechanism (via the option `named_table`) that is separate from the process registry. Processes can access a table using either a TID or a table's name.

Use of ETS tables can give rise to race conditions that are analogous to those existing in a shared memory system. These include typical read/write races but also races related to deallocation and changes in a table's ownership (and therefore access rights).

Operations on ETS tables are annotated with the label **ets**.

3.4 Other Components (Labels: unique)

In this subsection we group features of an Erlang/OTP system whose interference we consider to be beyond the scope of this paper. We give brief arguments about our decision for each case individually.

Ports Operations on files as well as interaction with other programs managed by the operating system are managed by *ports*. Ports generally behave like processes (e.g., support sending and accepting messages and signals, can have a registered name, etc.). Like ETS tables, they are also owned by processes and are designed to be controlled and communicate with just their owner. Erlang has a dedicated datatype for *port identifiers*.

Since the controlling process is expected to be the only one interacting with the port in a sequential way, we will not consider interfering operations on ports in the rest of this paper.

References and Other "Unique Values" *References* are values of a dedicated datatype and are generated by certain built-in operations. For all practical purposes, a reference value is guaranteed to be unique on a particular node.

Built-ins that generate references, process, port or ETS table identifiers, or unique integers, interfere, since they may use shared data to ensure the uniqueness of the returned values. However, values returned by such built-ins are in general unpredictable; e.g., the result of a comparison between two PIDs or two references can be different for reasons that cannot always be explained by the scheduling of the operations generating those values. We present all such operations in Table 1 and will not give tests where interference between such operations could also be observed due to scheduling. We will, however, use the label **unique**, to remind of this effect.

Monitor and timer creation operations are included since their results include references. The variants of the `spawn` operation that establish a monitor together with spawning a process also generate references.

Date and Time There are a number of built-in operations that retrieve date and time information. In the past, such mechanisms were also used to obtain unique, increasing integers, but as of Erlang/OTP 18 their uses have been clarified [9]. It is straightforward to see that programs can have different results if processes execute such operations in different relative orders (and e.g., compare the returned values). Such effects are nevertheless not due to sharing memory but due to dependence on a clock. We list such built-ins in Table 2 and do not discuss them further in the rest of this paper.

Table 2. Date and time built-ins.

`erlang:date/0`	`erlang:time/0`
`erlang:localtime/0`	`erlang:timestamp/0`
`erlang:monotonic_time/{0,1}`	`erlang:time_offset/{0,1}`
`erlang:now/0`	`erlang:universaltime/0`
`erlang:system_time/{0,1}`	

Debugging A number of Erlang/OTP built-ins are intended for debugging purposes and their interference with other built-ins is not generally interesting for reasoning about programs. We list all built-ins that we consider as debugging support in Table 3, together with a minimal explanation about their intended uses. As the interference of these operations can be unnecessarily broad, we will ignore them in the rest of this paper.

Table 3. Debugging built-ins.

BIF	Purpose
`erlang:is_process_alive/1`	checks liveness of a process
`erlang:process_display/2`	prints process info
`erlang:process_flag/3`	enables debugging of another process
`erlang:process_info/{1,2}`	shows info about a process
`erlang:processes/0`	returns all live processes
`erlang:registered/0`	returns all registered processes
`erlang:resume_process/1`	resumes execution of a process
`erlang:statistics/1`	returns info about node
`erlang:suspend_process/2`	pauses execution of a process
`erlang:system_flag/2`	sets node-wide settings
`erlang:system_info/1`	returns node-wide info
`erlang:system_monitor/{0,1,2}`	enables advanced node monitoring
`erlang:system_profile/{0,2}`	enables profiling for the node
`ets:all/0`	returns all ETS tables
`ets:info/{1,2}`	returns info about an ETS table
`ets:slot/2`	arbitrary access to an ETS table

Process Dictionary Each Erlang process a stateful dictionary data structure, which is intended to be local to that process. It is however possible for another process to inspect it via the debugging operations `erlang:process_info/{1,2}`. Given that the last is a debugging operation, we consider operations that involve the dictionary (shown in Table 4) as process local and do not further consider their interactions.

Table 4. Process dictionary related built-ins.

`erlang:erase/0`	`erlang:get/0`	`erlang:get_keys/0`	`erlang:put/2`
`erlang:erase/1`	`erlang:get/1`	`erlang:get_keys/1`	

Primitive Printing There exist a number of built-ins (shown in Table 5) used for primitive printing. These built-ins have obvious side-effects that can differ based on their scheduling, but do not, per se, operate on shared memory. We will therefore exclude them.

Table 5. Primitive printing built-ins.

`erlang:display/1 erlang:display_string/1 erlang:display_nl/0`

OS Interaction Finally, Erlang/OTP provides a number of built-in operations that are handled directly by the underlying operating system. Most of these operations are only reading data (e.g., OS version or system time), but it is also possible to use them to set environment variables (and thus introduce races if e.g., those are then read by another process). Again, such interference is beyond the scope of this paper, so we will ignore those operations too.

3.5 Summary of Process Exiting

When a process p is exiting with reason r the following changes occur at the VM level:

- the process status is marked as `'exiting'` and new messages are not delivered to p
- process p is no longer considered "alive"
- if there exist pending timers for p's PID, they are canceled
- ownership of ETS tables owned by the process is transferred to designated "heir processes"; tables without an heir are deleted
- if p had a registered name this name is unregistered
- link signals are sent to every linked process
- "monitor down" messages are sent for every monitor

The order that these changes take place is unspecified in the Erlang/OTP documentation and it is unclear what guarantees are provided by the implementation. It is for example possible that process execution is interrupted during exiting and only some of the ETS tables owned by it have been passed on or deleted. Similarly, if a process had a registered name, it is not guaranteed that when a monitor signal is sent the name will have been unregistered, even though the current implementation satisfies this constraint. To keep our presentation simple, we chose to treat exiting as an event by itself, named exit, and consider all these changes to happen atomically. Nevertheless, due to the asynchronous description of message/signal send and delivery, some of the possibly non-deterministic/unspecified behaviour is maintained (e.g., it is not certain when each one of the linked processes will actually receive an exit signal).

4 Litmus Tests Suite

We now move on to the description of the litmus tests suite. We begin by giving the design goals that impose a structure to the individual tests, continue with the overall structure of the repository and finish by providing criteria for the inclusion of tests in the suite.

4.1 Litmus Tests Design and Structure

The most common purpose of a test suite is to ensure that the behaviour of a program remains consistent. This is achieved either by ensuring that the result of the program is always some expected value, or at the very least that a particular bad behaviour (e.g., the program crashing) is not possible as a result of running the tests. Litmus tests [2] can follow a slightly different approach, also describing allowed behaviours that a particular program can have. Values specified as *allowed* results need not be returned at every run of the test; there should however exist executions that indeed return those values. Conversely, *forbidden* values must never be returned.

We are interested in creating test programs that can have *several* different allowed values, under different interleavings of their processes. Each test therefore includes the following parts:

- A function, which describes the program executed by the top (*main*) process of the test. This top process has full freedom to execute any code, including spawning additional processes, but it should ultimately return some value. The most common structure of its code is to start with a (possibly empty) setup code block, then spawn two processes that will execute the two operations that interfere with each other, and finish with some code that has different results based on which operation won the race. Note that this structure corresponds to the four boxes we have been using in the examples we have presented so far.
- All the possible allowed "result values" that can be returned by the *main* process.
- Metadata describing the operations that are interacting in the test to produce different results.

We do not strictly enforce a structure on the *main* function, as any such imposed structure (e.g., specific number of interfering processes, support for options for synchronization of their execution, etc.) will have to be shared among tests which may not require it. Additionally, such shared structure corresponds to code that should probably be placed in a separate file, and not be directly visible in the test; ensuring that such code does not inadvertently affect the particular operations whose interference we want to test is an unnecessary burden.

Using these pieces, we can define the following:

- A number of tests, each checking that a particular specified allowed value is indeed returned in some executions, similar to the behaviour of allowed values in litmus tests.
- A test checking that the returned value is always one of the specified values; it is forbidden for the test to have other results.

As an example, we show in Figure 1 the test for the interaction between two `erlang:register/2` built-ins using the same name. The pair of interfering operations is specified with the attributes `operation_1` and `operation_2`. The main function is `test/0` and it is spawning the two children processes that can interfere. Rather than using the `erlang:whereis/1` built-in, as we did in our example in the introduction, we just check if the second process terminates normally or crashes, using a monitor. The only way for this process to crash is to fail in its `register` operation because the name is already in use.

The `litmus.hrl` header file, whose contents are shown in Figure 2, is defining the tests we described earlier: one test for each of the two allowed results, each containing an assertion that is expected to fail in some execution (showing that the result is indeed possible) and a third test, containing an assertion that is not expected to fail in any execution, ensuring that any result of the test is included in the list of all expected results.

```
%%% @author Stavros Aronis <aronisstav@gmail.com>

-module(register_register_name).

-operation_1({erlang,register,2}).
-operation_2({erlang,register,2}).

-define(RESULT_1, true).
-define(RESULT_2, false).

-include("../../headers/litmus.hrl").

p1() ->
  register(name, self()).

p2() ->
  register(name, self()).

test() ->
  Fun1 = fun() -> p1() end,
  _P1  = spawn(Fun1),
  Fun2 = fun() -> p2() end,
  {P2, M} = spawn_monitor(Fun2),
  receive
    {'DOWN', M, process, P2, Tag} ->
      Tag =/= normal
  end.
```

Figure 1. The `registry/register_register_name.erl` test.

```
-export([test/0]).

-export([possible_1/0, possible_2/0, exhaustive/0]).

-include_lib("stdlib/include/assert.hrl").

possible_1() ->
  ?assertNotEqual(?RESULT_1, test()).

possible_2() ->
  ?assertNotEqual(?RESULT_2, test()).

exhaustive() ->
  ?assert(lists:member(test(), [?RESULT_1, ?RESULT_2])).
```

Figure 2. The `litmus.hrl` header.

4.2 Repository

The collection of litmus tests is hosted at:

https://github.com/aronisstav/erlang-concurrency-litmus-tests

For each of the labels introduced in Section 3 (except **unique**) there exists a sub-directory under the `litmus` directory, containing tests relevant to that particular component. In Section 5 we present an overview of the included tests.

As the contents of the repository may evolve, we have included the Git tag `Erlang17` (SHA: 95e7cd8) to refer to the version used in this paper. However, we recommend looking at the most recent version of the repository's `master` branch.

4.3 Criteria for Inclusion

Given the partitioning of an Erlang/OTP VM into components presented in Section 3, it is fairly easy to determine when built-in operations or events interfere. However, there exist cases where the same pair of operations can interfere in different ways, even under the same label. In the interest of exhaustiveness, we try to include all such possible ways in the repository.

Different Argument Combinations The test shown in Figure 1 exposes the interference between two register operations when using the same name for different processes. It is documented that a register operation also interferes when the process given as argument already has a name. Therefore, the test suite also contains a second test where the two register operations have the same *Pid* (process identifier, and not name) as target. In Table 6 (Section 5) there exist indeed two tests for a pair of `erlang:register/2` operations, describing exactly this situation.

Different Contexts It can also be the case that two operations interfere in different ways depending on the context. As an example, the interference between an `erlang:register/2` and an `erlang:unregister/1` with the same name is slightly different, depending on whether the registered name is already registered at the time when the operations can interact or not (the names of two suitable tests are included in the same entry of Table 6).

Notice that the same pair of built-in operations or events can interfere in ways that logically belong to different labels: as an example, two sig_deliver events with two different abnormal reasons (that are not `'kill'`) and the same recipient process, interfere differently if the recipient is trapping exits (a **mailbox** interference, as the signals will be placed in the mailbox in different orders) and in a different way if it is not (a **signal** interference, as the recipient will exit with different abnormal reasons). We remind that the intention of our characterization is to assign at least one common component label to each member of any pair of built-in operations or events that can interfere in an observable way. To find all the ways in which a particular pair of built-in operations or events interfere, one should lookup each pair element in Tables 18 and 19 and then inspect each of the shared labels at the table indicated in Table 17.

Validation In order to verify the validity of the tests we used Concuerror [5, 11]. We give details about the validation in Section 6.

5 Interference Tables

We now present the interaction of built-in operations and events within each component, as they were presented in Section 3. We show a number of tables, that have built-in operations and events in their rows and columns and relevant tests in their intersections. (In the electronic version of the paper, the names of the tests are clickable links to the repository.) To stay consistent while avoiding duplication, rows have all the built-in operations and events related to a label, while columns contain either(i) only the built-ins and events that have relevant interferences, or (ii) the same contents as rows but in reverse order, creating a "triangular" table. For some built-in operations (such as e.g., `spawn`) we show only one of the many available forms and use the symbol "`*`" (as part of a name or arity) to denote the existence of other variants, which can be found in Appendix A. As mentioned in Section 4.2, tests in the repository are organized in directories, based on their label and name.

Table 6. Names of tests showing interferences between built-ins and events labeled as **registry**.

	erlang:whereis/1	erlang:unregister/1	erlang:send/2	erlang:register/2	erlang:monitor/2	timer_send	exit
exit	exit_whereis	exit_unregister	exit_send	exit_register_taken	exit_monitor		
timer_send		timer_send_unregister		register_timer_send			
erlang:monitor/2		monitor_unregister		monitor_register			
erlang:register/2	register_whereis	register_unregister_free register_unregister_taken	register_send	register_register_name register_register_pid			
erlang:send/2		send_unregister					
erlang:unregister/1	unregister_whereis	unregister_unregister					
erlang:whereis/1							

Table 7. Names of tests showing interferences between built-ins and events labeled as **leader**.

	erlang:spawn*/*	erlang:group_leader/2	erlang:group_leader/0
erlang:group_leader/0		group_leader_get_set	
erlang:group_leader/2	group_leader_spawn	group_leader_set_set	
erlang:spawn*/*			

Table 8. Names of tests showing interferences between built-ins and events labeled as **process**.

	exit
exit	
erlang:group_leader/2	exit_group_leader_leaded exit_group_leader_leader
erlang:link/1	exit_link_no_trap exit_link_trap
erlang:monitor/2	exit_monitor
erlang:register/2	exit_register_exiting
ets:give_away/3	exit_ets_give_away

5.1 Processes and Nodes

Process Table Names of tests showing the interference of operations with the **process** label appear in Table 8. All these operations depend on the PID/port they are given as argument corresponding to a live process/open port and have different behaviour if that is not the case.

Process Registry Names of tests showing the interference of operations with the **registry** label appear in Table 6. The captured behaviour is a lookup or modification in the process registry. The monitor operation is included as it supports monitoring registered names. As lookups for timers that send to a named process happen when the timer is triggered, the timer_send event is included in the tables, just like send. Notice that, unlike send operations, timer_send events have no interference with a registered process exits, since the events fail silently in such cases.

Group Leader Names of tests showing the interference of operations with the **leader** label appear in Table 7.

5.2 Messages and Signals

In this section, we describe a number of different labels, each intended to capture interaction between built-ins and events that involve different aspects of the message and signal handling.

Operations and Events that Send Messages and Signals As explained in Section 3.2, the sending and the delivery of a message or signal are more suitably viewed as two separate events in an Erlang/OTP node. Any send operation can then be considered to not interfere with other send operations to the same target, since it is the order of the respective deliver events that determine the target's behaviour (i.e., mailbox contents or exiting behaviour). The operations that send messages or signals can, nevertheless, interfere with other operations and events due to other reasons (e.g., accesses to the process registry). Additionally, some operations, such as the `erlang:link/1` built-in when the target does not exist and the process is trapping exits result in immediate placement of messages. In Table 9 we list all operations whose sending can be seen as a separate event from the delivery. The exit event is included due to the signals and messages that can be triggered by links and monitors. Link and monitor operations have also been included for the cases where the non-existence of a live target can not be easily checked and the message or signal can not be attributed to an exit event.

Table 9. Built-ins and events that send messages and signals.

erlang:exit/2	erlang:send/2	erlang:send_after/{3,4}
erlang:link/1	timer_send	erlang:start_timer/{3,4}
erlang:monitor/2	exit	ets:give_away/3

Any operation or event that can change the contents of a mailbox is marked with the **mailbox** label. The interactions between pairs of such operations and events are shown in Table 10. They involve the placement of messages in the mailbox and the difference in the behaviour of a `receive` with a timeout based on the delivery of a message or trapped signal.

Interference from Signal Delivery The delivery of an abnormal exit signal can cause a process that is not trapping exit signals to terminate abnormally. If that process was about to perform some other operation, then the ordering of the signal arrival and the execution of that operation can be important. An obvious such operation is enabling or disabling trapping exit signals, but also any other operation that is not merely "reading" some shared information. We mark such operations with the **signal** label and show their interference with signal delivery in Table 12.

Table 10. Names of tests showing interferences between built-ins and events labeled as **mailbox**.

	msg_deliver	sig_deliver
msg_deliver	msg_msg msg_msg_indirect	msg_sig
receive	msg_receive	sig_receive
sig_deliver	msg_sig	sig_sig sig_sig_indirect
`erlang:demonitor/2`	msg_demonitor msg_demonitor_info msg_demonitor_info_flush	
`erlang:link/1`	msg_link	sig_link
`erlang:monitor/2`	msg_monitor	sig_monitor

The `erlang:unlink/1` built-in has two different tests, depending on whether the delivered signal is coming from the process that is unlinked (and should be ignored) or another link is canceled (and therefore an abnormal exit signal should not be propagated).

Links The operations that manipulate links interfere when they change the status of a link or when determining whether a signal to a linked process will be emitted or delivered. We show interference of operations via the **link** label in Table 13. The operation that establishes a link is also in a race with the exit of the target process, as shown in the table for the **registry** label (Table 6). Since it is not possible to observe the difference between an `erlang:unlink/1` operation interfering with an exit signal emission (connected to the exit event, unlinking implies that the signal is not emitted) and the delivery of the signal (connected to the sig_deliver event, unlinking implies that the signal is discarded upon delivery), we do not add a separate test, deeming the relevant test in the **signal** table (Table 12) sufficient.

Monitors Perhaps surprisingly, the operations that set and cancel monitors on processes or ports do not interfere directly with each other. The reason behind this is that a demonitor operation can only be executed by the process that set up the monitor, and this will always happen in program order. Also, just like in the previous paragraph, the difference between a monitor canceled before it is emitted or before it is delivered is not possible to detect, therefore the tests in the **mailbox** table (Table 10) are enough.

Timers The "normal" sequence of built-in operations and events that involve a particular timer is the following:1) a call to one of the operations that setup the timer is performed; 2) the timer expires, marked by a timer_send event; 3) the timer's message arrives, marked by a msg_deliver event. Notice that just like send operations, operations that setup timers do not have interferences (even the registry lookup happens at timer expiration). We omit the interference between two `erlang:read_timer/2` operations as their results are based again on the clock and not on how the operations are scheduled to access shared memory. As mentioned earlier, if a target process exits, the timer is automatically canceled[4]. The interference of operations that involve timers is shown in Table 11.

[4]There also exists the possibility of asynchronous timer read or cancellation, which we do not cover in this paper but can be sufficiently covered by appropriate events, denoting when such asynchronous requests are handled and resulting in messages being later delivered to the caller.

5.3 ETS

In Table 14 we split built-in operations and events that affect ETS tables into categories, based on(i) whether they affect the table itself or just its entries, and (ii) whether they only read entries or can also modify them. These three categories are:(1) `ets-global`, which includes operations that modify parameters of the table itself; note that the exit event of a process is included here, because it implies transfer of ownership or deletion of tables owned by the exiting process (as explained in Section 3.5) (2) `ets-write`, which includes operations that modify one or more entries of the table by e.g., inserting new values or updating/deleting existing ones, and (3) `ets-read`, which includes operations that just read entries.

Since there can be interference between nearly every `ets-global` operation when paired any another `ets-global`, `ets-write` or `ets-read` operation, and the same can be said for every `ets-write` operation when paired with any other `ets-write` or `ets-read` operation, instead of presenting an exhaustive pairing of all operations, we describe broadly their interference in categories in Table 15 and give some highlighting examples about interesting pairings in the next paragraphs. For the four cells in Table 15 that contain just the word "Yes" we intend to add tests in the test suite in a possibly less pedantic way, without aiming to be fully exhaustive but exercising all interesting pairs of operations and covering all the ETS built-ins. We have nevertheless deemed inclusion of those tests in the text of this paper superficial.

Interference Involving* `ets-global` *Operations The interference involving `ets-global` operations are the following:(i) A call to `ets:new/2` can interfere with other calls that try to create a table with the same name or rename a table using a taken name. This, of course, is the case only if said call to `ets:new/2` is trying to create a named table. Conversely, interference exists also with operations that free a name, by renaming or destroying a table. Such interference between pairs of `ets-global` operations are shown in Table 16. (ii) Any operation in the `ets-write` or `ets-read` categories can interfere with an operation that changes the name (`ets:rename/2`), the owner (`ets:give_away/3`)[5] or the existence of a table (`ets:new/2`, `ets:delete/1` or an exit event).

Interference Involving* `ets-write` *Operations For practically every pair of operations in the `ets-write` category there exist interference, which can be detected by the final contents of the table. Pairs of pure deletion operations (i.e., delete operations that do not also return deleted values) are the only ones that are exempt from this pattern, as they usually commute: regardless of the order of two deletions, the table will end up missing all entries that the two operations deleted.

Operations in the `ets-write` category are also interfering with operations in the `ets-read` category, if they involve (or could involve) the same entry or entries.

Interference Involving* `ets-read` *Operations Pairs of operations in the `ets-read` category never interfere with each other.

[5]As described in Section 3.3, Erlang supports access restrictions to ETS tables, based on parameters set at table creation and the owner of the table. If any of the operations on an ETS table are executed with improper access rights they fail. Changing the owner of a table can therefore interfere with any operation on the entries of the table.

Table 11. Names of tests showing interferences between built-ins and events labeled as **timer**.

	erlang:read_timer/2	erlang:cancel_timer/2
exit	exit_read_timer	exit_cancel_timer
timer_send	timer_send_read_timer	timer_send_cancel_timer
erlang:cancel_timer/2	cancel_timer_read_timer	cancel_timer_cancel_timer
erlang:read_timer/2		cancel_timer_read_timer

Table 12. Names of tests showing interferences between built-ins and events labeled as **signal**.

	sig_deliver
exit	sig_exit
sig_deliver	sig_sig
erlang:cancel_timer/1	sig_cancel_timer
erlang:exit/2	sig_exit_2
erlang:group_leader/2	sig_group_leader
erlang:link/1	sig_link
erlang:process_flag/2	sig_trap_exit
erlang:register/2	sig_register
erlang:send/2	sig_send
erlang:send_after/4	sig_send_after
erlang:spawn*/*	sig_spawn
erlang:start_timer/4	sig_start_timer
erlang:unlink/1	sig_unlink_linked sig_unlink_other
erlang:unregister/1	sig_unregister

Table 13. Names of tests showing interferences between built-ins and events labeled as **link**.

	erlang:unlink/1	erlang:link/1
erlang:link/1	link_unlink_linked link_unlink_unlinked	
erlang:unlink/1		

6 Validation of the Tests

In order to verify the validity of each test we used Concuerror [5, 11], a stateless model checking tool for finding concurrency-related errors in Erlang programs or verifying their absence.

Concuerror uses a stateless search algorithm to systematically explore the execution of a test program under conceptually all process interleaving. To achieve this, the tool performs a code rewrite that inserts *preemption points* (i.e., points where a context switch is allowed to occur) in the code under execution. This instrumentation allows Concuerror to take control of the scheduler when the program is run, without having to modify the Erlang VM in any way. Concuerror's instrumentor automatically inserts preemption points before any call to a built-in operations, unless the operation has been identified as safe (i.e., operating only on data that is not shared among processes)[6]. Additionally the tool supports instrumentation of process exiting, `receive` statements and message passing mechanisms in a way compatible with the events mechanism presented in this paper.

[6]In order to control the results of the built-in operations listed in Table 1 between different executions, Concuerror simulates part of these built-ins.

The tool normally uses dynamic partial order reduction techniques [1] to limit the number of interleavings it explores but these can be suppressed to run a test in a truly exhaustive way, regarding process interleaving and delivery of messages and signals. All tests in the repository were run using this mode of the tool, and the behaviour of each litmus test was verified to be the expected one.

7 Related Work

Formal Erlang Specification Even though there is no canonical specification for Erlang, prior work exists on providing a formal semantics [13], including the majority of built-in operations discussed in this paper. Ideas such as the separation of sending and delivering operations are presented there as well, but the overall focus is on the formal specification of a model of the language rather than the description of the interactions between built-in operations that are available in the implementation. The specification of Svensson et al. [13] also intentionally diverges from the actual implementation (e.g., providing only monitors and not links), in the interest of better design, simplification and reducing interferences.

Erlang Testing Tools The interferences between Erlang built-in operations is naturally of interest for a number of different testing and verification tools for Erlang programs.

As mentioned earlier, Concuerror [5, 11] is a tool for finding concurrency-related errors in Erlang programs or verifying their absence. In order to be effective, Concuerror inserts preemption points only at process actions that interact with (i.e., inspect or update) some concurrency-related primitive that accesses VM-level data structures that are shared by processes. These are exactly the operations we are describing in this paper. Moreover, the conditions under which these operations race, as exposed in the tests, are crucial for techniques such as optimal-DPOR [1] that reduce the state space searched by Concuerror. Using tests like the ones included in the test suite we describe here, we were able to identify several cases where the tool was missing dependencies between Erlang's built-ins and subsequently patched all such cases.

McErlang [4, 10] is a model checking tool for Erlang programs, built, in part, following the aforementioned formal specification [13]. Interferences between operations are arising as a result of their modeling and are not identified separately. Just as Concuerror, McErlang uses a source-to-source transformation to instrument programs. However, since the tool does not take into account which operations interfere under what conditions, it can explore significantly many more interleavings of Erlang programs than it would need to do if it used the fine-grained interferences we present in this paper.

PULSE [6], is a random testing tool for Erlang programs. It differs from Concuerror and McErlang in that it cannot provide guarantees about the absence of concurrency errors, but it is also similar to them in that it also instruments programs in order to

Table 14. ETS built-ins split into three categories.

Category	Built-ins and events				
ets-global	ets:delete/1	ets:give_away/3	ets:new/2	ets:rename/2	exit
ets-write	ets:delete/2	ets:delete_object/2	ets:insert_new/2	ets:select_delete/2	ets:update_counter/{3,4}
	ets:delete_all_objects/1	ets:insert/2	ets:safe_fixtable/2	ets:take/2	ets:update_element/3
ets-read	ets:first/1	ets:lookup_element/3	ets:member/2	ets:select_count/2	
	ets:last/1	ets:match/{1,2,3}	ets:next/2	ets:select/{1,2,3}	
	ets:lookup/2	ets:match_object/{1,2,3}	ets:prev/2	ets:select_reverse/{1,2,3}	

Table 15. Interferences between the categories of built-ins and events labeled as **ets**.

	ets-read	ets-write	ets-global
ets-global	Yes	Yes	Yes (Table 16)
ets-write	Yes	Yes	
ets-read	No		

Table 16. Names of tests showing interferences between built-ins and events labeled as ets-global.

	ets:rename/2	ets:new/2
ets:new/2	new_rename	new_new
ets:rename/2	rename_rename	

explore different schedules. PULSE does not handle all built-in operations automatically, requiring manual annotations from the user.

The tests presented in this paper can serve as a minimal sanity check suite for any of the aforementioned tools.

Language Specifications via Tests There has been extensive work in providing a full specification to the Javascript language via the ECMAScript standard specification and the accompanying conformance suite [14]. The test suite included in this work can be seen as a similar attempt, at a much smaller scale, to describe the expected behaviours of Erlang's concurrency built-in operations (including all their non-deterministic behaviours due to interference).

Litmus Tests in Other Settings Litmus tests [2] were already mentioned in Section 4 as an inspiration for the structure of tests in our suite. In that work, tests are expressed in a high level language and they are compiled into C before they can run. The goal is to test whether particular behaviours are observable on actual processors.

8 Concluding Remarks

In this paper and, more importantly, in the publicly available test suite that accompanies it, we have shown that there are many different ways that Erlang/OTP built-ins race with each other in accessing shared memory and, as a consequence, result in observable differences in concurrent programs that depend on the interleaving of these built-ins. Although many, if not most, of these interferences are probably well-known to seasoned Erlang programmers, we are not aware of any other document or study that tried to characterize and categorize them in the level of detail that we have done in this paper.

Table 17. Summary of labels.

Label	Table	Meaning
ets	14, 15	access/modification of one or more ETS tables
leader	7	access/modification of process leader info
link	13	access/modification of links
mailbox	10	access/modification of a process' mailbox
process	8	check for whether a process is live
registry	6	access/modification of process registry info
signal	12	handling of an exit signal
timer	11	access/modification of a timer
unique	1	generation of a unique value

We believe that we have provided a relatively complete account of all shared-memory interferences between the current Erlang/OTP built-ins and runtime system events. But even if we have missed some, it is relatively trivial for others to extend the existing test suite with more programs that cover interferences that may be missing from our list, or add new ones if new built-ins accessing shared memory get added to the Erlang/OTP in one of its future releases.

A Reference Tables of Labels for Erlang Built-Ins and Events

A.1 Summary of Labels

In Table 17 we give a summarized meaning to each label introduced in Section 3, explaining why built-in operations or events are annotated by it. We also give a link to the table describing the built-in operations that have been annotated with that label.

A.2 Labels of Built-In Operations

Table 18 contains a list of the built-ins in the current implementation of Erlang/OTP, annotated with all relevant labels that have appeared in this paper. We have excluded ETS operations, which all have the **ets** label, as well as all built-ins that belong to the categories that we explicitly excluded from our presentation (purely functional, node, ports, os, date/time, debug, dictionary and printing operations).

A.3 Labels of Events

Just like built-in operations, each event also has a set of labels, shown in Table 19.

Acknowledgments

This work was carried out within the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center), partly supported by the Swedish Research Council.

Table 18. Labels of built-in operations.

BIF	Labels
`erlang:cancel_timer/2` *	**signal**, **timer**
`erlang:demonitor/2` ★	**mailbox**
`erlang:exit/2`	**signal**
`erlang:group_leader/0`	**leader**
`erlang:group_leader/2`	**leader**, **process**, **signal**
`erlang:link/1`	**mailbox**, **link**, **process**, **signal**
`erlang:make_ref/0`	**unique**
`erlang:monitor/2`	**mailbox**,**process**,**registry**,**unique**
`erlang:process_flag/2`	**signal**
`erlang:read_timer/2` †	**timer**
`erlang:register/2`	**process**, **registry**, **signal**
`erlang:send/2` ‡	**registry**, **signal**
`erlang:send_after/4` ⁋	**signal**, **unique**
`erlang:spawn*/*` ⊕	**leader**, **signal**, **unique**
`erlang:start_timer/4` ⊗	**signal**, **unique**
`erlang:unique_integer/1` ◇	**unique**
`erlang:unlink/1`	**link**, **signal**
`erlang:unregister/1`	**registry**, **signal**
`erlang:whereis/1`	**registry**
`erts_internal:open_port/2`	**unique**
`ets:give_away/3`	**process**
`ets:new/2`	**unique**

* With an empty list as second argument, also covers `cancel_timer/1`.
★ With an empty list as second argument, also covers `demonitor/1`.
† With an empty list as second argument, also covers `read_timer/1`.
‡ The built-in operation `send/3` is relevant only for remote sends.
⁋ With an empty list as last argument, also covers `send_after/3`.
⊕ Covering all variants of `spawn`.
⊗ With an empty list as last argument, also covers `send_after/3`.
◇ With an empty list as argument, also covers `unique_integer/0`.

Table 19. Labels of events.

Event	Labels
exit	**ets**, **process**, **registry**, **signal**, **timer**
msg_deliver	**mailbox**
receive	**mailbox**
sig_deliver	**mailbox**, **signal**
timer_send	**registry**, **timer**

References

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/2535838.2535845

[2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests against Hardware *(Lecture Notes in Computer Science)*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.), Vol. 6605. Springer, Berlin, Heidelberg, 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

[3] Joe Armstrong. 2010. Erlang. *Comm. of the ACM* 53, 9 (2010), 68–75. https://doi.org/10.1145/1810891.1810910

[4] Clara Benac Earle and Lars-Åke Fredlund. 2009. Recent Improvements to the McErlang Model Checker. In *Proceedings of the 8th ACM SIGPLAN Workshop on Erlang (Erlang '09)*. ACM, New York, NY, USA, 93–100. https://doi.org/10.1145/1596600.1596613

[5] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. 2013. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST '13)*. IEEE Computer Society, Washington, DC, USA, 154–163. https://doi.org/10.1109/ICST.2013.50

[6] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. Finding Race Conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/1596550.1596574

[7] Erlang FAQ 2017. Academic and Historical Questions. http://erlang.org/faq/academic.html#idp33226448. (2017). Accessed: 2017-05-27.

[8] Erlang Run-Time System Application (ERTS) Reference Manual 2017. erlang. http://erlang.org/doc/man/erlang.html. (2017). Accessed: 2017-05-27.

[9] Erlang Run-Time System Application (ERTS) User's Guide 2017. Time and Time Correction in Erlang. http://erlang.org/doc/apps/erts/time_correction.html. (2017). Accessed: 2017-05-27.

[10] Lars-Åke Fredlund and Hans Svensson. 2007. McErlang: A Model Checker for a Distributed Functional Programming Language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 125–136. https://doi.org/10.1145/1291151.1291171

[11] Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. 2011. Test-Driven Development of Concurrent Programs using Concuerror. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang (Erlang '11)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2034654.2034664

[12] STDLIB Reference Manual 2017. ets. http://erlang.org/doc/man/ets.html. (2017). Accessed: 2017-05-27.

[13] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. 2010. A Unified Semantics for Future Erlang. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang (Erlang '10)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/1863509.1863514

[14] Test262: ECMAScript Language Conformance Test Suite 2017. ECMA TC39. https://github.com/tc39/test262. (2017). Accessed: 2017-05-27.