# Declarative Debugging of Concurrent Erlang Programs

## (Extended version)

**Rafael Caballero · Enrique Martin-Martin · Adrián Riesco · Salvador Tamarit**

July 22, 2016

**Abstract** Erlang is a concurrent language with features such as actor model concurrency, no shared memory, message passing communication, high scalability, and availability. However, the development of concurrent programs is a complex and error prone task. In this paper we present a declarative debugging approach for concurrent Erlang programs. Our debugger asks questions about the validity of transitions between the different points of the program that involve message passing. The answers, which represent the intended behavior of the program, are compared with the transitions obtained in an actual execution of the program. The differences allow us to detect program errors and to point out the pieces of source code responsible for the bugs. In order to represent the computations we present a semantic calculus for concurrent Core Erlang programs. The debugger uses the proof trees in this calculus as the debugging trees used for selecting the questions asked to the user. The relation between the debugging trees and the semantic calculus allows us to establish the soundness of the approach. The theoretical ideas have been implemented in a debugger prototype.

**Keywords** Concurrency, Declarative Debugging, Erlang, Semantics

## 1 Introduction

Concurrent programming has become increasingly prevalent in the last years, because it allows programs to exploit the great amount of parallelism in current hardware. However, debugging concurrent programs is a complex task [32] that poses additional challenges with respect to sequential programming. In particular, the programmer must be very careful about the information that is shared between processes, the execution order, and even how the information is accessed.

One of the most successful concurrent languages of the last few years is Erlang [5], a programming language that combines functional programming (e.g. higher-order functions and single assignments), with features required in the development of scalable commercial applications like garbage collection, built-in concurrency based on the actor model [2], and hot-swapping of modules. The language is used as the base of many fault-tolerant, reliable software systems. The development of this kind of systems is a complicated process where tools such as discrepancy analyzers [31], test-case generators [37,36, 6], and debuggers play an important rôle.

In this paper, we focus on the problem of debugging Erlang concurrent programs.The standard distribution of the language already includes a useful trace-debugger including different types of breakpoints, stack tracing, graphical watch of processes and messages, and other features. However, debugging a program is still a difficult, time-consuming task, and for this reason alternative or complementary debugging tools are convenient.

The problem when tracing Erlang concurrent computations is to define a suitable notion of *computation step*. In the case of Erlang sequential programs there is a typical notion of computation step: function calls. Using for instance the Erlang trace debugger, the user can examine a function call and decide whether the returned result was expected for the given input values. If the result is unexpected the call is considered *invalid*. There are two possible explanations behind invalid calls: either

Rafael Caballero · Enrique Martin-Martin · Adrián Riesco
Universidad Complutense de Madrid, Madrid, Spain
rafa@ucm.es - emartinm@ucm.es - ariesco@ucm.es

Salvador Tamarit
Universidad Politécnica de Madrid, Madrid, Spain
stamarit@babel.ls.fi.upm.es

the called function is erroneously defined or it contains a call to another function that produces an unexpected result. In previous works [10,11] we have shown how this idea can be automatized to detect erroneous function definitions in sequential Erlang programs. Once an erroneous function has been located, the bodies of its defining rules can be further examined in order to locate a more specific error [12].

However, in a concurrent computation the situation is not so simple, and function calls are no longer the computation steps that the user needs to examine in order to find the bug. Maybe the function has returned the expected value but it has sent and incorrect message to some process, or it has 'forgotten' to create a new process. Moreover, waiting until a function returns a value is sometimes unfeasible since concurrent functions are often defined as non-terminating loops. A usual structure of a concurrent program, using Erlang terminology, is:

1. Wait for input messages using a `receive` statement, until a message matching any of the possibilities enumerated in the `receive` statement is accepted.
2. Process the message.
3. Go back to step 1.

On the other hand, trying to depict and trace the whole bunch of running processes with their respective interactions can be an overwhelming task.

In this paper, we propose to use concurrent transitions as suitable computation steps for debugging, distinguishing the following possibilities:

– In absence of message consumption (that is, if no `receive` statement is found), we consider the computation of a function result as a computation step. This is important, because most of the errors in concurrent programs still occur in the sequential part [38]. However, in order to adapt to the concurrent setting, the result contains not only the final value but also the messages sent and the processes created during the transition.
– In the concurrent fragments of the computation, the computation step is the transition from one `receive` statement (the *origin* of the transition) to either a final value or the next `receive` statement (the *destination*). This transition consumes one message and also considers as part of the result the messages sent and the processes created.

Thus, in our debugging setting we plan to examine the transitions of each process, giving the user the opportunity to indicate whether the transition result is expected or not. A transition ends in what we call a *medium-sized normal form* (*mnf*), which can be either a value or a `receive` statement.

Formally, our debugger is based on a general technique known as *declarative debugging* [42]. Also known as *declarative diagnosis* and *algorithmic debugging*, this technique abstracts the execution details to focus on the validity of the results. It has been widely employed in the logic [42], functional [34], and object-oriented [27] programming languages. Declarative debugging is a two-step scheme: it first computes a debugging tree representing a wrong computation, and then traverses this tree by asking questions to the *oracle* (normally the user, but other sources as unit tests can be used [44]) until the bug is identified. In our case, the debugging trees correspond to proofs in a semantic calculus for concurrent Core Erlang programs, the intermediate language that Erlang uses to codify all the programs in a uniform representation. The calculus non-trivially extends our previous work for sequential programs in [10] with rules for process creation and message sending/receiving, elements that will play an important rôle in the questions asked by the debugger, and also to deal with *configurations* of processes instead of simple expressions.

In our setting, each debugging tree node corresponds to a transition of a process, possibly involving message passing and the creation of new processes. A tree node is considered *valid* if the transition produced the expected result, messages, and new processes; and *invalid* otherwise. The debugger navigates the tree by asking questions to the user about the validity of some nodes until a buggy node—an invalid node with only valid children—is found, being its associated piece of code the source of the error.

The main contributions of this paper are:

1. A new semantic calculus for Core Erlang concurrent programs, including features like message passing and process creation. This calculus uses message outboxes instead of the traditional approach of using message inboxes. This mitigates the lack of expressiveness of single-node semantics regarding the order of messages, as pointed out in [43], and simplifies the debugging task.
2. Since we obtain our debugging trees by using this calculus, we can prove the soundness and completeness of our approach.
3. A prototype of the declarative debugger for concurrent Erlang programs, called EDD (Erlang Declarative Debugger). This prototype, written on top of our declarative debugger for sequential Erlang, is written also in Erlang, and allows us to assess the applicability of our approach. The prototype asks questions to the user about the validity of the tran-

sitions relevant for locating the error. Moreover, it also generates graphical representations showing different perspectives of the computation steps.
4. We can detect errors in non-terminating executions, such as deadlocks and livelocks. Since our calculus can infer this kind of situations, a non-terminating scenario is a possible initial symptom.

The rest of the paper is organized as follows: Section 2 explains the basics of the Erlang language and describes a running example used throughout the paper. Section 3 presents several alternatives (including EDD) to debug the running example, showing their strong and weak points. Section 4 explains our debugger in detail. Section 5 presents the calculus we have tailored for concurrent Core Erlang programs. Section 6 presents the transformations applied to the proof trees obtained from the semantic calculus to apply declarative debugging, as well as the associated soundness and completeness results and a description of the errors that can be found. Section 7 presents related work regarding debugging concurrent systems and semantics for Erlang. Finally, Section 8 shows the conclusions and points out the lines of the future work.

## 2 Erlang

Erlang [5] is a concurrent language following the functional paradigm (including features such as higher-order functions, λ-abstractions, and single assignment) with dynamic typing and strict evaluation. Regarding concurrency, it follows the actor model [2]. It allows to model problems where different processes communicate through asynchronous messages, and gives support to fault-tolerant and soft real-time applications, as well as to non-stop applications thanks to the so-called *hot swapping*.

Erlang supports basic data types such as atoms (identifiers started by a lowercase letter like `ok`), Boolean values (`true` and `false`), or float numbers (`3`, `3.14`, `1.12E-10`). Unlike other functional languages, it is not possible to define new algebraic data types in Erlang programs, and all the data structures must be represented with *tuples* and *lists*. Tuples are sequences of values between curly braces (e.g. `{msg,hello}`), while lists are represented using a Prolog-like syntax, either as a list of elements (e.g. `[1,hello]`), or using the head-tail notation (e.g. `[1 | [hello | []]]`).

Variables (identifiers started by an uppercase letter) are bound using pattern matching, either directly (e.g. `X=fact(3)`, `{X,Y}=getPoint()`), by parameter passing in functions, or by branching constructions (in `case`, `receive`, or `try-catch` statements).

Erlang processes have no shared memory and run in the *Erlang Virtual Machine* (EVM). As they are lightweight processes it is possible to have thousands of them running in the same EVM at the same time [3].

The predefined Erlang function `spawn` creates a new process executing a function with some arguments and returns its *process identifier* (PID). Process communication is performed by means of asynchronous message passing. The content of a message can be any Erlang value, and are sent using the `!` (pronounced *bang*) operator. For example, to send the message `{msg,hello}` to the process `PID` we write `PID ! {msg,hello}`. Each process can read a message that has been sent to it using the `receive...end` construction. This instruction can contain several *clauses* with a pattern, an optional guard, and a *body*. The behavior is the following: it takes the first (oldest) message and pattern-matches it against each clause in order. If a successful match occurs and the guard is fulfilled, it retrieves the message, binds the variables of the pattern, and executes the body. Otherwise, it tries with the next older message. If no message matches the pattern and fulfilles the guard, then the process is suspended until it receives a new message.

Throughout the rest of the paper we use the running example of Figure 2 which represents a simplified version of the connection process in the Transmission Control Protocol (TCP) [39].

This method, known as *three-way handshaking*, is described in the left-hand side of Figure 1. The connection starts when the client sends a SYN packet and an initial sequence number $x$ to a port open in the server. If the port is not open the server sends back a packet with the RST bit on, indicating that the connection has been rejected (this message is not displayed in the figure). If the port is open, the server sends a SYN-ACK packet back to the client, including an acknowledgment number set to one more than the received sequence number $(x + 1)$, and with its own sequence number $y$. Finally, the client replies with an ACK back to the server, with the sequence number set to the received acknowledgement value $(x+1)$, and the new acknowledment number to one more than the received sequence number $(y+1)$. At this point the transmission of data packets can start.

The `main` function in Figure 2 creates one server process and two client processes using the **`spawn`** calls of lines 2–4. Each **`spawn`** call has 3 arguments: the module that contains the code of the process (the macro `?MODULE` is expanded to the current module name), the function that the process will execute (`server_fun` and `client_fun`, respectively), and a list of arguments passed to that function. In this example the processes receive as first parameter the identifier of the process to

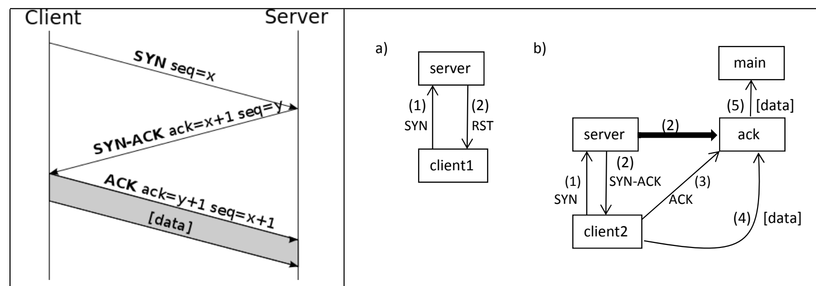


**Fig. 1** TCP Three-way handshake

```
1 main() ->
2   Server_PID = spawn(?MODULE, server_fun, [self(), 50, 500]),
3   spawn(?MODULE, client_fun, [Server_PID, 57, 100, client1]),
4   spawn(?MODULE, client_fun, [Server_PID, 50, 200, client2]),
5   receive
6     {data,D} ->
7       D
8   end.
9
10 server_fun(Main_PID, Port, Seq) ->
11   receive
12     {Client_PID, {syn, Port, SeqCl}} ->
13       Ack_PID = spawn(?MODULE, ack, [Main_PID, Port, SeqCl+1, Seq+1, Client_PID]),
14       Client_PID ! {Ack_PID, {syn_ack, SeqCl+1, Seq}},
15       server_fun(Main_PID, Port, Seq+1) ;
16     {Client_PID, {syn, _, _}} ->
17       Client_PID ! rst
18       % server_fun(Main_PID,Port,Seq+1)
19   end.
20
21 ack(Main_PID, Port, Ack, Seq, Client_PID) ->
22   receive
23     {Client_PID, {ack, Port, Seq, Ack}} ->
24       receive
25         {Client_PID, {data, Port, D} ->
26           Main_PID ! {data, D};
27         _ ->
28           Main_PID ! {data, error_data}
29       end;
30     _ ->
31       Main_PID ! {data, error_ack}
32   end.
33
34 client_fun(Server_PID, Port, Seq, Data) ->
35   Server_PID ! {self(), {syn, Port, Seq}},
36   syn_ack(Port, Data, Seq+1).
37
38 syn_ack(Port, Data, Ack) ->
39   receive
40     rst ->
41       {port_rejected, Port} ;
42     {Ack_PID, {syn_ack, Ack, Seq}} ->
43       Ack_PID ! {self(), {Ack, Port, Seq+1, ack}},  % swap  Ack <--> ack
44       Ack_PID ! {self(), {data, Port, Data}},
45       {Seq+1, Ack, Port, Data}
46   end.
```

**Fig. 2** TCP Three-way handshake in Erlang

which they must report: the server reports to the current process—`self()`—and the clients to the server process. The second parameter is the port to be used. The third parameter is an arbitrary number of sequence. Client processes receive an additional parameter which here is simply an atom (`client1` and `client2`, respectively), but in general represents the data that each client will try to transmit. The `main` function is expected to return the first data obtained after a successful connection, which in this example should be `client2` because the first client tries to connect the server by the closed port 57. The expected behavior of the processes is explained in the right-hand side of Figure 1. In the case of the first client—part a) of the figure—it is expected that the server replies with RST to SYN (line 17) because a closed port has been chosen. The second client—part b)—sends SYN to the server by the open port 50. Then the server sends SYN-ACK back (line 14) and creates a new process `ack` that will recieve the `ACK` message and, if the handshake succeeds, the data from the client. The client sends this last ACK message and the data (lines 43–44) to this process, which itself reports to the `main` process (line 26). However, the program contains two errors. The first one is indicated by the comment (`%`) in line 18: the server should never end because more clients may request new connections, but there is no recursive call to `server` when a closed port is tried. Due to this missing recursive call, the server ends after attending the first client, and the second client remains waiting for SYN-ACK in line 42. Thus, the second client cannot establish the connection and send its data, and hence the user observes that the evaluation of the expression `tcp:main()` never ends because it is waiting for the data. The second error, which will be noticeable after correcting the first one, is that the two parameters `ack` and `Ack` should be interchanged in line 43.

## 3 Debugging Erlang

In this section we explore different alternatives that an Erlang programmer could use to detect the cause of the bugs above, and we present our approach from the point of view of the programmer. In all the cases the starting point is the same: the programmer expects a terminating program that returns `"client2"`, however, the program does not terminate and it does not show anything in the console.

### 3.1 Debugging by printing

The simplest debugging technique is often the first one to be tried: instrumenting the code with `io:format` expressions to print helpful information. This approach allows to generate a trace of the program reduced only to a subset of interesting events. Since the program is not terminating, it seems convenient to print the start and end of every function, as well as their parameters and the process where they are executed. For example in the `server_fun` function we insert the following two expressions:

```
server_fun(Main_PID, Port, Seq) ->
  io:format("START: server_fun(~p,~p,~p) in ~p~n",
    [Main_PID,Port,Seq,self()]),
  receive
    ...
  end,
  io:format("END: server_fun(~p,~p,~p) in ~p~n",
    [Main_PID,Port,Seq,self()]).
```

After instrumenting all the functions, the execution of the program starting from `main()` prints:

```
START: main()
START: server_fun(<0.33.0>,50,500) in <0.40.0>
START: client_fun(<0.40.0>,57,100,client1)
         in <0.41.0>
START: client_fun(<0.40.0>,50,200,client2)
         in <0.42.0>
START: syn_ack(57,client1,101) in <0.41.0>
START: syn_ack(50,client2,201) in <0.42.0>
END: server_fun(<0.33.0>,50,500) in <0.40.0>
END: syn_ack(57,client1,101) in <0.41.0>
END: client_fun(<0.40.0>,57,100,client1)
       in <0.41.0>
```

The most interesting message is the following:

```
END: server_fun(<0.33.0>,50,500) in <0.40.0>
```

The function `server_fun` finishes, but this behavior is unexpected because the server should continuously listen for connections. Inspecting the code of this function the programmer could fix the bug in line 18 by inserting the recursive call.

In order to fix the second bug we focus on the first `receive` expression of the function `ack`, since it is the place where the value `error_ack` is generated. Concretely, it is very interesting to discover what message is consumed in line 31. Since the information about function invocation does not seem very helpful for this bug, we remove all the `START` and `END` print expressions and modify the `ack` function:

```
...
M ->
  io:format( "ack(~p,~p,~p,~p,~p) in ~p~n",
    [Main_PID,Port,Ack,Seq,Client_PID,self()]),
  io:format( "MESSAGE: ~p~n", [M]),
  Main_PID ! {data,error_ack}
...
```

With these print expressions the program generates two lines:

```
ack(<0.33.0>,50,201,501,<0.42.0>) in <0.43.0>
MESSAGE: {<0.42.0>,{201,50,501,ack}}
```

From the actual arguments of the `ack` function we know that, at this point of the handshake, `Port` is 50, `Ack` is 201, and `Seq` is 501. Moreover, the received message seems legitimate because it contains the correct client PID `<0.42.0>` and a tuple of 4 elements `{201,50,501,ack}`, so it should have matched the pattern in the first branch of the `receive` expression (line 23). However, by carefully inspecting the message and the pattern in the code we discover that the `ack` constant is not correctly placed in the message. Since in this small program the only function that sends *ack* messages is `syn_ack`, a quick scrutiny of its code reveals the origin of the bug in line 43.

In summary, we need one execution of the program with suitable instrumented `io:format` expressions to resolve each bug. This debugging approach is simple and fast, but it presents some disadvantages. The main drawback is that we have to modify and restore our program, which is prone to leave useless fragments in the program. Moreover, the introduction of `io:format` expressions must be done with care as we can inadvertently change the behavior of the program. For example, inserting a `io:format` as the last expression of an Erlang clause would change the returned value. This is critical, as the programmer can end up debugging a different program. Furthermore, the `io:format` expressions can introduce delays in the execution that alter the interleavings between processes and therefore change the relative order of the messages. As a consequence, the instrumented program can obtain different values and even different bugs from the original program. Finally, although the traces generated are focused to some relevant events for the considered bug, programs with a high level of concurrency can produce very long traces. Finding a hint in traces with hundreds or thousands of lines can be unfeasible.

At a similar level of functionality to printed traces we can consider the Erlang *Event Trace* application.[1] This application allows programmers to select which events to trace during the execution of the program (messages, function calls, changes in the process status, etc.) and display them graphically as a sequence chart. The information in the trace is formatted in a nicer way, although the technique presents the same drawbacks as the traditional `io:format` debugging previously mentioned.

[1] http://erlang.org/doc/man/et.html

### 3.2 Built-in Erlang debugger

Another common approach for debugging a program is using the built-in Erlang debugger[2] through either its graphical interface or from the command line shell. This powerful debugger provides many useful options like conditional breakpoints, step-by-step execution inside one process, a list of all the processes along with their statuses, and the possibility of inspecting the pending messages of any process. The debugger supports debugging several modules at the same time, with the only limitation that they must be compiled with the `debug_info` flag. Figure 3 shows a debugging session of the TCP example where there are 4 active processes and we execute step-by-step the function `server_fun` in process `<0.189.0>`, which has two pending messages. Since the program we are debugging is not terminating, it seems convenient to execute step-by-step all the processes and see how variables and messages evolve. Thus, we launch the graphical debugger with the command `debugger:start()`, load the module, and add breakpoints in the first expression of the functions that execute the spawned processes: `server_fun`, `client_fun`, and `ack`. Then we run the goal `main()` and open a window for each generated process. Since the server starts with a `receive` expression, we advance the client processes first: we decide to first continue with `client1` and then to `client2`. At this point we can switch to the server, which has two pending messages, and execute the `receive` expression: it selects the second branch, sends a `rst` and terminates. The server should keep listening to new connections, so we have spotted the bug in the `server_fun` function.

Detecting the cause of the second bug is easy as `error_ack` messages are only generated in the `ack` function. Therefore we add a breakpoint at the beginning of this function and execute the program. When the execution stops in the breakpoint, we perform some steps and compare the message received and the pattern of a valid *ACK* message, detecting that the *ack* constant is incorrectly placed inside the message.

Similarly to debugging by printing, we need one execution of the program with suitable breakpoints to detect each bug. The built-in Erlang debugger is very comfortable, since we can watch the evolution of our concurrent program state in a step-by-step execution. However, this is also the main drawback. When we execute step-by-step one process and stop the others, we are implicitly choosing one particular scheduling. This scheduling can interleave the process and messages in a different order from the original execution, so the

[2] http://erlang.org/doc/apps/debugger/debugger_chapter.html

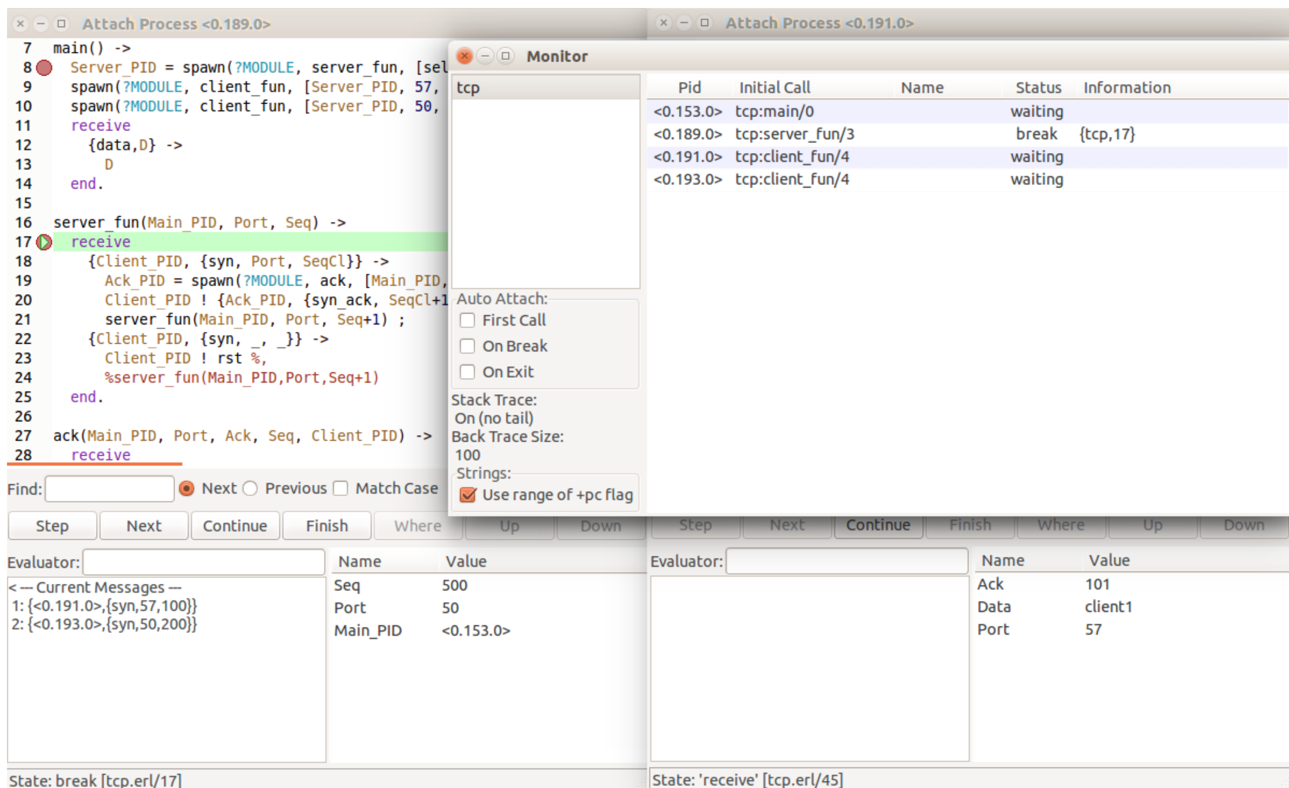


**Fig. 3** Debugging session using the Erlang built-in graphical debugger

values obtained can be different in the debugging session. This situation easily arises in our example: if we choose to first execute the `client2` process and then `client1`, the server would receive the messages in that order. The message from `client2` uses a valid port 50, so the TCP handshake succeeds and the program returns `{data, error_ack}` and finishes. This situation is completely different from the non-terminating behavior we were debugging.

As a final comment, popular integrated development environments for Erlang like *Erlide*[3] come with their own debuggers integrated. Their usage is more comfortable since their are combined with the editor, however, their capabilities are similar to the built-in Erlang debugger.

## 3.3 Unit and property-based testing

Another approach that could be used to debug programs is using *unit tests* [40]. In this kind of testing, individual program *units* (usually functions) are tested in isolation by comparing the actual output with the expected one for concrete arguments. Unit testing is integrated in almost every programming language, including Erlang with the EUnit [16] testing framework. In order to test a function, the developer must invoke it with suitable arguments and then check that the results are the expected ones by means of assertions. These assertions usually involve the comparison of terms, but they can also check whether the invocations raise certain exceptions or errors. EUnit executes automatically all the tests in a module and shows a summary containing the number of passed tests. If any test has failed, it shows the obtained and expected result. This information is very useful as it is the initial symptom of a wrong behavior, however, EUnit does not provide further help to detect the fragment of code that caused the bug.

Let us see how we could use EUnit to detect and fix the bugs in our TCP example. First, we need to decide which situations we want to test. Function `main()` represents a scenario where there are two connections, one rejected and one successful, but we should check other possibilities. In Figure 4 we consider 5 additional situations: `rst` for a single rejected connection, `connect` for a single successful connection, `rst_twice` for two rejected connections, `connect_twice` for two successful connections, and `interleaved` for three interleaved connections (two rejected and one successful in the mid-

[3] `http://erlide.org/`

```
48 tcp_test_() ->
49   [?_assertEqual(client2,main()), rst(), connect(), rst_twice(), connect_twice(), interleaved()].
50
51 get_messages() ->
52   timer:sleep(500), % Wait 0.5s for the latest messages
53   receive
54     M ->
55       [M|get_messages()]
56     after 0 ->
57       []
58   end.
59
60 rst() ->
61   Server_PID = spawn(?MODULE, server_fun, [self(), 50, 500]),
62   A = client_fun(Server_PID, 57, 100, client1),
63   Mbx = get_messages(),
64   [?_assertEqual(A,{port_rejected,57}), ?_assertEqual(Mbx,[])].
65
66 connect() ->
67   Server_PID = spawn(?MODULE, server_fun, [self(), 50, 500]),
68   A = client_fun(Server_PID, 50, 100, client1),
69   Mbx = get_messages(),
70   [?_assertEqual(A,{501,101,50,client1}), ?_assertEqual(Mbx,[{data,client1}])].
71
72 rst_twice() ->
73   Server_PID = spawn(?MODULE, server_fun, [self(), 50, 500]),
74   A = client_fun(Server_PID, 57, 100, client1),
75   B = client_fun(Server_PID, 60, 100, client2),
76   Mbx = get_messages(),
77   [?_assertEqual(A,{port_rejected,57}), ?_assertEqual(B,{port_rejected,60}), ?_assertEqual(Mbx,[])].
78
79 connect_twice() ->
80   Server_PID = spawn(?MODULE, server_fun, [self(), 50, 500]),
81   A = client_fun(Server_PID, 50, 100, client1),
82   B = client_fun(Server_PID, 50, 300, client2),
83   Mbx = get_messages(),
84   [?_assertEqual(A,{501,101,50,client1}), ?_assertEqual(B,{502,301,50,client2}),
85    ?_assertEqual(Mbx,[{data,client1},{data,client2}])].
86
87 interleaved() ->
88   Server_PID = spawn(?MODULE, server_fun, [self(), 50, 500]),
89   A = client_fun(Server_PID, 33, 0,   client1),
90   B = client_fun(Server_PID, 50, 100, client2),
91   C = client_fun(Server_PID, 51, 0,   client3),
92   Mbx = get_messages(),
93   [?_assertEqual(A,{port_rejected,33}), ?_assertEqual(B,{502,101,50,client2}),
94    ?_assertEqual(C,{port_rejected,51}), ?_assertEqual(Mbx,[{data,client2}])].
```

**Fig. 4** EUnit tests for the TCP three-way handshake example.

dle). Notice that in order to check that the results are the expected ones we use two sources of information. First, the value returned by `client_fun` in line 36: if the connection is rejected (line 41) then `{port_rejected, Port}` is returned, otherwise it returns a tuple with the sequence numbers, the port, and the data transmitted (line 45). Secondly, we use the function `get_messages` that returns a list of all the pending messages of the current process, i.e., those messages received and forwarded by the server to `Main_PID` in the `ack` function.

The previous functions are used by `tcp_test_()` to create a nested list of assertions that will be checked by EUnit. We only use equality assertions `?_assertEqual(E,O)`, where `E` is the expected value and `O` the obtained one. When we invoke `tcp:test()` EUnit executes all the assertions in the testing functions[4] of the module `tcp` and show a summary of the results. With the original code of `tcp`, however, this invocation does not return any value since it does not terminate. In order to detect the source of the bug, we must test each scenario individually, discovering that: a) the assertions in `rst` pass, b) `connect` and `connect_twice` terminates

[4] Functions whose name ends with `_test_()`.

with some failed assertions, and c) `main`, `rst_twice`, and `interleaved` do not terminate. This behavior make us suspect that the problem might reside in how the server handles connections to closed ports, since every scenario with a rejected connection followed by any other connection does not terminate. This suspicion leads us to the missing recursive call in line 18 after inspecting the code.

When we fix the first bug and execute again all the test, EUnit shows that there still are four assertions that fail (we show only the first and the last ones):

```
> tcp:test().
tcp:49: tcp_test_...*failed*
**error:{assertEqual,[{module,tcp},
    {line,49},
    {expression,"main ( )"},
    {expected,client2},
    {value,error_ack}]}
  output:<<"">>
(...)
tcp:94: interleaved...*failed*
**error:{assertEqual,[{module,tcp},
    {line,94},
    {expression,"Mbx"},
    {expected,[{data,client2}]},
    {value,[{data,error_ack}]}]}
  output:<<"">>

=====================================
  Failed: 4.  Skipped: 0.  Passed: 11.
error
```

All four failed assertions show a similar symptom: the mailbox of the main process should contain messages with the client data but it has only messages with the constant `error_ack`. Since `error_ack` messages are only generated in line 31 when the received message is malformed, a close inspection of the client code will reveal the bug resides in line 44 when submitting `data` messages.

Finally, after fixing the second bug, EUnit reports that all test cases have passed:

```
> tcp:test().
All 15 tests passed.
ok
```

Unit testing is a widely-used testing technique. However, its main goal is to detect wrong behaviors, so it cannot directly help developers to debug the code and identify the source of the problem. As we have shown with our example, depending on the quality, number, and granularity of the unit tests, it is possible to have some hint of the cause of a wrong behavior; however, they have not been designed for that task. On the other hand, applying unit testing requires an extra effort of test creation. Even in our simple example, there are infinite possibilities for the port numbers tried, the data transmitted, and the possible interleavings between rejected/successful connections. Therefore, the manual creation of unit tests is a complex and time-consuming task.

In order to overcome the mentioned disadvantages of unit testing *QuickCheck* [19] appears, mixing *random* and *property-based testing*. QuickCheck was initially proposed for Haskell but it has been ported to many programming languages. The Erlang version, *Quviq QuickCheck* [6] has a commercial license, so an open-source counterpart called *PropEr* [37,36] has been developed. The approach of these tools is different from unit testing: instead of writing tests for concrete scenarios, the developer defines general properties that must be verified and the tool checks them using many randomly generated values. The commonest properties have the form $\forall x \in T.P(x)$, where $x$ is a value of type $T$ and $P(x)$ is the property to verify on $x$. Furthermore, they have also support for stateful systems [7] where the response of an action depends on the hidden state, like a distributed database whose responses depend on the previous queries. In stateful systems the properties are more complex and require that developers define a state machine to create and update an abstract state that can be inspected and is consistent with the real hidden state. These abstract states are then used to check whether the responses are correct w.r.t. the current state or not. Another interesting feature of QuickCheck and PropEr is that when they find some random value that does not verify the property, they shrink it to find a minimal symptom.

Let us see how we could use PropEr to debug our TCP example, which will require to extend the program with the code in Figure 5. First, we need to define the property to verify. For simplicity, we will focus on the results of single-connection scenarios. We define a boolean function `is_correct_connection` that, given a port `SP` where the server is listening, a server sequence number `SS`, a port `CP` where the client will try to connect, a client sequence number `CS`, and some data `D` to transmit; checks whether the result of `client_fun` and the received messages are correct or not. If the server and client port coincide, then the client must return a 4-tuple containing the incremented sequence numbers and the server must receive a `{data,D}` message. Otherwise, the client must return `{port_rejected,CP}` and the server must not receive any messages. Then we define a Boolean function `prop_one_connection` that contains the PropEr property. This property uses the macro `?FORALL(V,G,E)` where `V` is the pattern that will contain the generated value, `G` is the generator of random values, and `E` is the Boolean expression to check, usually involving the variables in `V`. In line 109 we define a generator of 5-tuples:

```
96  is_correct_connection(SP,SS,CP,CS,D) ->
97    Server_PID = spawn(?MODULE, server_fun, [self(), SP, SS]),
98    A = client_fun(Server_PID, CP, CS, D),
99    Mbx = get_messages(),
100   case SP == CP of
101     true  ->
102       A == {SS+1,CS+1,SP,D} andalso Mbx == [{data,D}];
103     false ->
104       A == {port_rejected,CP} andalso Mbx == []
105   end.
106
107 prop_one_connection() ->
108   ?FORALL( {SP,SS,CP,CS,D},
109            {integer(10,20),integer(15,25),non_neg_integer(),non_neg_integer(),term()},
110            is_correct_connection(SP,SS,CP,CS,D)).
```

**Fig. 5** *PropEr* property used to debug single-connection TCP scenarios.

the first 4 elements are integer numbers, whereas the last element of the tuple is any Erlang term. We use two integer generators: `integer(Min,Max)` to generate port numbers in a range, and `non_neg_integer()` to generate non negative sequence numbers. For every 5-tuple generated, `is_correct_connection` (line 110) is invoked.

Once we have defined the property, we launch PropEr:

```
> proper:quickcheck(tcp:prop_one_connection()).
......................................!
Failed: After 39 test(s).
{10,20,10,23,'R\007gro-a'}

Shrinking ....(4 time(s))
{10,15,10,0,0}
false
```

We obtain a fail after 39 tests, and the value that does not verify the property is `{10,15,10,0,0}` after shrinking, i.e., a situation where both the server and the client use port `10` and transmit the message `0`. Unlike EUnit, PropEr does not show further detailed information about what failed, so we need to invoke the Boolean function again with the found value to detect which equality is failing. After detecting that the problem resides in the `error_ack` message inside `Mbx`, a close inspection of the code would reveal the bug problem in line 44 when submitting `data` messages.

We know that the TCP example contains two bugs, but after fixing the first one PropEr passes all the tests:

```
> proper:quickcheck(tcp:prop_one_connection()).
.........................................
.........................................
..................
OK: Passed 100 test(s).
true
```

The reason is that the `prop_one_connection` property cannot reproduce the nontermination behavior, as it arises when the server receives a connection in a closed port followed by any other connection. Therefore we would need to create more complex properties chaining 2, 3, or any list of randomly generated connections. Since we are dealing with nontermination, we must use the wrapper `?TIMEOUT` to consider as failing those properties whose evaluation needs more time than a given limit; otherwise PropEr will block and will not show any information. Using these more complex properties, PropEr reports only the first shrunk value that break the property, so detecting the source of the error from this small information will be harder than in the EUnit case.

Concluding, random property-based testing frameworks like PropEr or QuickCheck overcome some limitations of unit testing, however, they require to write complex properties and particular generators to cover the desired situations. Moreover, they do not detect the cause of the bug but discover the values that generate a wrong behavior regarding a concrete property. In that sense they are purely testing tools and do not help to spot the source of the bugs.

### 3.4 Other automatic testing tools

Although EUnit and QuickCheck/PropEr are the most popular testing tools in Erlang, we would like to mention other important testing tools. The first tool to consider is the *DIscrepancy AnaLYZer for ERlang programs* (Dialyzer) [31], included in the OTP/Erlang system. This completely automatic tool performs static analysis to identify software discrepancies and bugs such as definite type errors [41], race conditions [18], unreachable code, redundant tests, unsatisfiable conditions, and more. Although useful in many situations, it cannot detect any discrepancy in the TCP example:

```
$ dialyzer tcp.erl
Proceeding with analysis... done in 0m0.32s
```

```
done (passed successfully)
```

*McErlang* [23] is a model checking tool [20] for Erlang programs. It accepts an Erlang program, which is the model, and checks its execution against a correctness property. These correctness properties are implemented as *monitors*: observers that can examine program states and actions in order to verify the program. Monitors are written manually by the developer, or they can be automatically created from linear temporal logic [8, 20] (LTL) formulas. If McErlang finds an error in the model, it generates a counterexample (an execution trace) where the property does not hold. If we want to use McErlang to debug our TCP example, we must implement our own monitor or generate it from an LTL formula. However, since the information that McErlang provides is an execution trace that does not verify the property, its debugging capabilities are somewhat limited: we would need to inspect the code globally using the hints we could obtain from the counterexample.

Finally, *Concuerror* [17] is also a model checking tool, however, its approach is different from McErlang's. Given an Erlang program and its test suite, Concuerror systematically explores process interleaving and presents detailed interleaving information about errors that occur during the execution of these tests: abnormal process exits, stuck processes, and assertion violations. In this case the developer does not need to provide an explicit specification of the property to check. An interesting feature of Concuerror is that it does not explore all the possible interleavings naively, but uses *Dynamic Partial Order Reduction* [1] (DPOR) techniques to eagerly prune equivalent interleavings. Similarly to McErlang, applying Concuerror to debug our TCP example will only detect the problematic symptoms and provide some detailed information about the interleaving. For example, executing Concuerror starting from function `interleaved` in Figure 4 will detect that all the processes have finished but one client is blocked in the `receive` expression at line 39.

```
Erroneous interleaving 1:
* Blocked at a 'receive' (when all other processes
  have exited):
P in tcp.erl line 39
```

Since we do not expect the server to exit, this information could help us to spot the first bug about the missing recursive call in the *rst* case. However, Concuerror cannot help with the second bug: it only informs that the server process is blocked waiting for messages when all other processes have exited, but this is the expected behavior. Concuerror cannot find out that the `syn_ack` message that clients send to the server (line 43) is ill-formed, since these messages do not cause abnormal process exits, stuck processes, or assertion violations.

### 3.5 Debugging with EDD

We propose to debug the TCP program of Figure 2 using EDD. As mentioned, this tool guides the debugging session by asking questions about some fragments of the computation until it finds the source of the error, namely a function or a `receive` expression. The version of EDD that we present here is an extension of the tool developed for sequential programs [10, 13], so it shows a similar interface and behavior. However, it has been profoundly modified to handle concurrency: process creation, message passing, and `receive` expressions.

In the rest of this section we show step-by-step how to use EDD to find the errors in the code of Figure 2. We will present all the questions with full detail. Although their length is greater than the other approaches, we remark that they focus on particular fragments of the computation and thus they are easy to understand and answer. In order to debug the TCP program with EDD, we start the debugger with `edd:cdd("tcp:main()",500)`, which launches a concurrent debugging session starting from `tcp:main()` with a timeout of 500 milliseconds. The debugger shows then a summary of the processes (user's answers are remarked with a box):

```
**********************************
Pid selection
**********************************
1.- <0.131.0>
  First call:
    tcp:client_fun(<0.129.0>, 50, 200, client2)
  Result:
    Blocked because it is waiting for a message
2.- <0.130.0>
  First call:
    tcp:client_fun(<0.129.0>, 57, 100, client1)
  Result:
    {port_rejected,57}
3.- <0.129.0>
  First call:
    tcp:server_fun(<0.127.0>, 50, 500)
  Result:
    rst
4.- <0.127.0>
  First call:
    tcp:main()
  Result:
    Blocked because it is waiting for a message
(...)

Please, insert a PID where you have observed a
  wrong behavior: [3]
```

We already know that `tcp:main` is not working properly, but the summary provided by EDD can guide us

to a concrete "suspicious" process to start the debugging session. With this information we are sure that something is happening with process <0.129.0>. This process represents the server (it executes the function tcp:server_fun), so it should keep listening for connections. However, this process has finished with result rst. Therefore, we select the option 3 to start the debugging session from this process.

After selecting the suspicious process <0.129.0>, EDD asks the first question:[5]

```
Process <0.129.0> called
  tcp:server_fun(<0.127.0>, 50, 500).
What is wrong?
1. - Previous evaluated receive:
  receive
    {Client_PID, {syn, Port, SeqCl}} ->
      Ack_PID = spawn(tcp, ack, [Main_PID,
        Port, SeqCl + 1, Seq + 1, Client_PID]),
      Client_PID ! {Ack_PID, {syn_ack,
        SeqCl + 1, Seq}},
      server_fun(Main_PID, Port, Seq + 1);
    {Client_PID, {syn, _, _}} ->
      Client_PID ! rst
  end
  in tcp.erl:11
  Context:
    'Client_PID' = <0.130.0>
    'Main_PID' = <0.127.0>
    'Port' = 50
    'Seq' = 500
  Received messages:
    {<0.130.0>,{syn,57,100}}
      (from <0.130.0> to <0.129.0>)
    {<0.131.0>,{syn,50,200}}
      (from <0.131.0> to <0.129.0>)
  Consumed message:
    {<0.130.0>,{syn,57,100}}
      (from <0.130.0> to <0.129.0>)
2. - Evaluated to value: rst
3. - Sent messages:
       rst (from <0.129.0> to <0.130.0>)
4. - No created processes
5. - Nothing

[1/2/3/4/5/t/d/c/s/p/r/u/h/a]:  2
```

The first lines show that the question is about the process <0.129.0>, which invoked tcp:server_fun with arguments <0.127.0>, 50, and 500. It follows a list of fragments of the computation and the user must select *the first option* that is wrong, or 5 if everything is correct. The first line states that process <0.129.0> has reached the receive expression in line 11 with a particular mapping of variables (called *context*) and the set of received messages. EDD also shows the message that the mentioned receive expression has consumed. This information seems correct, so we continue with the next option, which states that process <0.129.0> is evaluated to rst. The server should not finish with a value but keep listening for new connections, so this option is wrong. Therefore, we type 2 and obtain a new question:

```
Process <0.129.0> evaluated
receive
  {Client_PID, {syn, Port, SeqCl}} ->
    Ack_PID = spawn(tcp, ack, [Main_PID,
      Port, SeqCl + 1, Seq + 1, Client_PID]),
    Client_PID ! {Ack_PID, {syn_ack,
      SeqCl + 1, Seq}},
    server_fun(Main_PID, Port, Seq + 1);
  {Client_PID, {syn, _, _}} ->
    Client_PID ! rst
end
in tcp.erl:11
What is wrong?
1.- Context:
  'Client_PID' = <0.130.0>
  'Main_PID' = <0.127.0>
  'Port' = 50
  'Seq' = 500
2.- Received messages:
  {<0.130.0>,{syn,57,100}}
  (from <0.130.0> to <0.129.0>)
  {<0.131.0>,{syn,50,200}}
  (from <0.131.0> to <0.129.0>)
3.- Consumed message:
  {<0.130.0>,{syn,57,100}}
  (from <0.130.0> to <0.129.0>)
4. - Evaluated to value: rst
5. - Sent messages:
  rst (from <0.129.0> to <0.130.0>)
6. - No created processes
7. - Nothing

[1/2/3/4/5/6/7/t/d/c/s/p/r/u/h/a]:  4
```

This question focus on the receive expression at line 11, the same that was mentioned in the first option of the previous question. This question contains more details about the receive expression (final result, sent messages, and created processes) and it asks us to select *the first aspect* that is wrong. The context and received/consumed messages are correct (options 1–3). However, we do not expect the receive expression to be evaluated to rst. Instead, we expect that it keeps listening to connections. Therefore, we select option 4.

After answering the second question, EDD detects the error in the code:

```
The error has been detected:
The problem is in pid <0.129.0>
while running receive
receive
  {Client_PID, {syn, Port, SeqCl}} ->
    Ack_PID = spawn(tcp, ack, [Main_PID,
      Port, SeqCl + 1, Seq + 1, Client_PID]),
    Client_PID ! {Ack_PID, {syn_ack,
      SeqCl + 1, Seq}},
    server_fun(Main_PID, Port, Seq + 1);
  {Client_PID, {syn, _, _}} ->
    Client_PID ! rst
```

[5] EDD output has been slightly reshaped to fit in the page.

```
  end
  in tcp.erl:11
```

EDD spots that the origin of the bug is the `receive` expression at line 11 using only two questions. That is indeed the place of the first bug, as the second branch should invoke `server_fun` recursively after sending the `rst` message.

After fixing the first bug, we still observe an anomalous behavior: the message received is `error_ack` instead of `client2`. Therefore, we start a new debugging session using `edd:cdd("tcp:main()", 500)` as before.

```
**********************************
Pid selection
**********************************
1.- <0.621.0>
  First call:
    tcp:ack(<0.616.0>, 50, 201, 502, <0.620.0>)
  Result:
    {data,error_ack}
2.- <0.620.0>
  First call:
    tcp:client_fun(<0.618.0>, 50, 200, client2)
  Result:
    {502,201,50,client2}
3.- <0.619.0>
  First call:
    tcp:client_fun(<0.618.0>, 57, 100, client1)
  Result:
    {port_rejected,57}
4.- <0.618.0>
  First call:
    tcp:server_fun(<0.616.0>, 50, 500)
  Result:
    Blocked because it is waiting for a message
5.- <0.616.0>
  First call:
    tcp:main()
  Result:
    error_ack
6.- Choose an event
7.- None

Please, insert a PID where you have observed a
  wrong behavior: [1..7]: 6
```

This time we will start the debugging session by selecting a suspicious event from the *communication and creation* diagram. This sequence diagram is generated by EDD and shows the different concurrent events that have happened during the execution of the program. As shown in Figure 6, process creation is represented as horizontal red arrows, message submission as horizontal black arrows between processes and message consumption in a `receive` expression as vertical black arrows inside a process. We observe that there is a suspicious `error_ack` sent in event number 21, so it seems a good starting point for the debugging session. Therefore we select option `6` and indicate the event we want to start from:

```
**********************************
Select an event from the sequence diagram: 21
Selected event:
Sent message: {data,error_ack}
  (from <0.621.0> to <0.616.0>)
```

Then EDD shows the first question of the debugging session, concerning process `<0.621.0>`:

```
Process <0.621.0> called
  tcp:ack(<0.616.0>, 50, 201, 502, <0.620.0>).
What is wrong?
1. - Previous evaluated receive:
  receive
    {Client_PID, {ack, Port, Seq, Ack}} ->
      receive
        {Client_PID, {data, Port, D}} ->
          Main_PID ! {data, D};
        _ -> Main_PID ! {data, error_data}
      end;
    _ -> Main_PID ! {data, error_ack}
  end
  in tcp.erl:22
Context:
  'Ack' = 201
  'Client_PID' = <0.620.0>
  'Main_PID' = <0.616.0>
  'Port' = 50
  'Seq' = 502
Received messages:
{<0.620.0>,{201,50,ack,502}}
  (from <0.620.0> to <0.621.0>)
{<0.620.0>,{data,50,client2}}
  (from <0.620.0> to <0.621.0>)
Consumed message:
{<0.620.0>,{201,50,ack,502}}
  (from <0.620.0> to <0.621.0>)
2. - Evaluated to value: {data,error_ack}
3. - Sent messages:
{data,error_ack}
  (from <0.621.0> to <0.616.0>)
4. - No created processes
5. - Nothing

[1/2/3/4/5/t/d/c/s/p/r/u/h/a]: 1
```

Process `<0.621.0>` executes the `ack` function that handles the second part of the connection handshake of the second client, which has PID `<0.620.0>` and uses the correct port `50`. As before we must select the first option that is wrong, or `5` if everything is correct. It is correct that the process reached the received at line 22, and the context is legitimate, however, the received messages are wrong. In partticular, the first message `{<0.620.0>,{201,50,ack,502}}` is ill-formed: the constant `ack` must be the first element of the message body, but it appears in the third position. We could abort the debugging session at this moment and inspect manually the code trying to detect where `ack` messages are submitted. However we will continue with the session, and EDD will lead us directly to the code that is causing the bug.

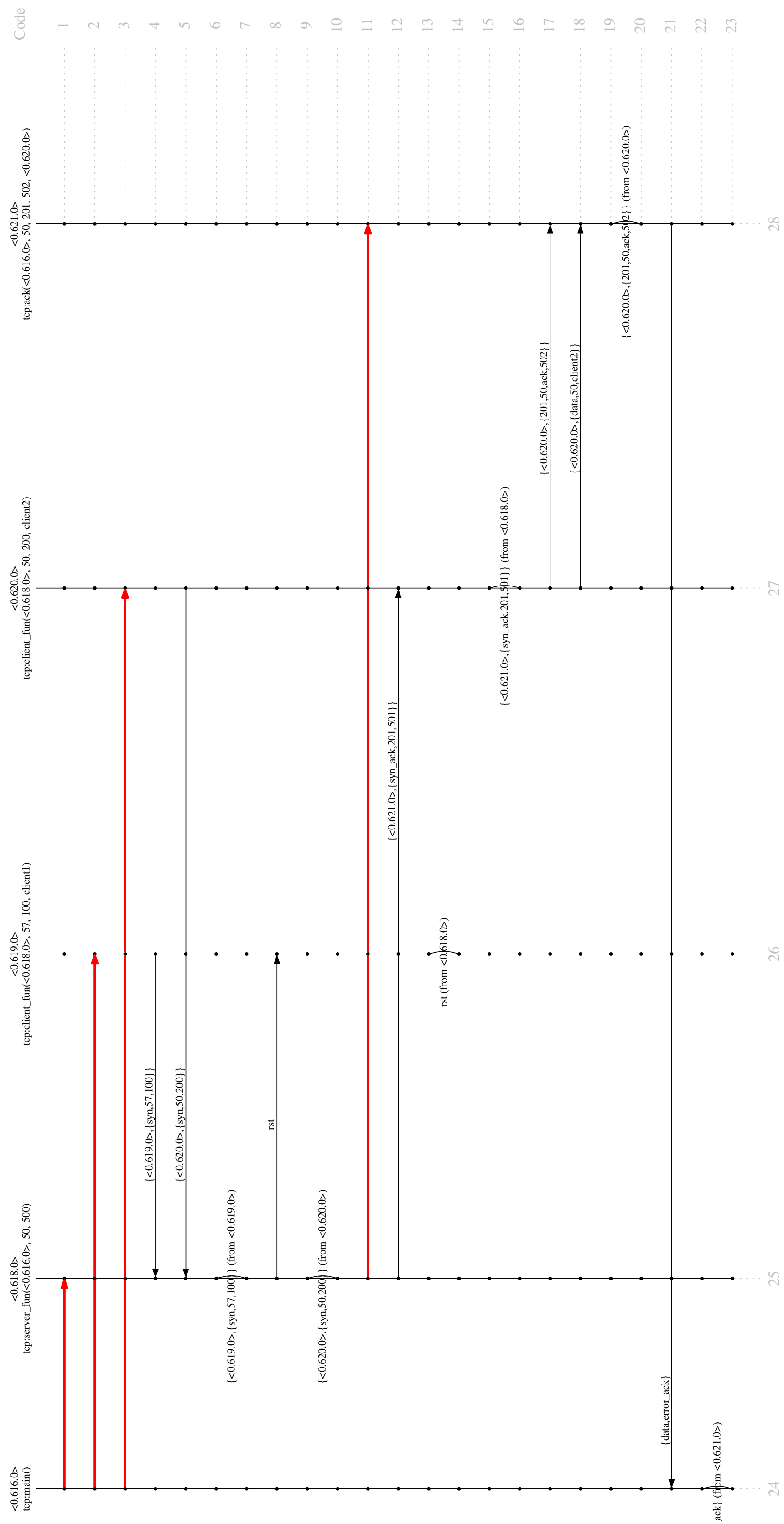

**Fig. 6** Communication and creation diagram created by EDD for the second debugging session.

After selecting 1 as the first wrong option, EDD asks about that particular receive expression:

```
Process <0.621.0> evaluated
  receive
    {Client_PID, {ack, Port, Seq, Ack}} ->
      receive
        {Client_PID, {data, Port, D}} ->
          Main_PID ! {data, D};
        _ -> Main_PID ! {data, error_data}
      end;
    _ -> Main_PID ! {data, error_ack}
  end
in tcp.erl:22
What is wrong?
1. - Context:
  'Ack' = 201
  'Client_PID' = <0.620.0>
  'Main_PID' = <0.616.0>
  'Port' = 50
  'Seq' = 502
2. - Received messages:
{<0.620.0>,{201,50,ack,502}}
  (from <0.620.0> to <0.621.0>)
{<0.620.0>,{data,50,client2}}
  (from <0.620.0> to <0.621.0>)
3. - Consumed message:
{<0.620.0>,{201,50,ack,502}}
  (from <0.620.0> to <0.621.0>)
4. - Evaluated to value: {data,error_ack}
5. - Sent messages:
{data,error_ack}
  (from <0.621.0> to <0.616.0>)
6. - No created processes
7. - Nothing

[1/2/3/4/5/6/7/t/d/c/s/p/r/u/h/a]: 2
*********************************
Which one is not expected?
1. - {<0.620.0>,{201,50,ack,502}}
  (from <0.620.0> to <0.621.0>)
2. - {<0.620.0>,{data,50,client2}}
  (from <0.620.0> to <0.621.0>)

[1/2/t/d/c/s/p/r/u/h/a]: 1
```

We answer that the received messages are wrong, and in the next question we select the ill-formed ack message (option 1). With this information EDD moves to the process <0.620.0> (from the initial process information we know that <0.620.0> is client2) to continue the debugging session:

```
Process <0.620.0> evaluated
  receive
    rst ->
      {port_rejected, Port};
    {Ack_PID, {syn_ack, Ack, Seq}} ->
      Ack_PID ! {self(), {Ack, Port, ack,
        Seq + 1}},
      Ack_PID ! {self(), {data, Port, Data}},
      {Seq + 1, Ack, Port, Data}
  end
in tcp.erl:39
What is wrong?
1. - Context:
  'Ack' = 201
  'Ack_PID' = <0.621.0>
  'Data' = client2
  'Port' = 50
  'Seq' = 501
2. - Received messages:
{<0.621.0>,{syn_ack,201,501}}
  (from <0.618.0> to <0.620.0>)
3. - Consumed message:
{<0.621.0>,{syn_ack,201,501}}
  (from <0.618.0> to <0.620.0>)
4. - Evaluated to value: {502,201,50,client2}
5. - Sent messages:
{<0.620.0>,{201,50,ack,502}}
  (from <0.620.0> to <0.621.0>)
{<0.620.0>,{data,50,client2}}
  (from <0.620.0> to <0.621.0>)
6. - No created processes
7. - Nothing

[1/2/3/4/5/6/7/t/d/c/s/p/r/u/h/a]: 5
```

We expect process <0.620.0> to reach the receive expression at line 39 (which is inside function syn_ack) and the context, received messages and consumed message are correct. However, the first sent message is the ill-formed ack message, so we select option 5. After selecting the ill-formed message as the unexpected submitted message in the next question, EDD locates the source of the bug in the receive expression at line 39, i.e., in the syn_ack function:

```
Which one is not expected?
1. - {<0.620.0>,{201,50,ack,502}}
  (from <0.620.0> to <0.621.0>)
2. - {<0.620.0>,{data,50,client2}}
  (from <0.620.0> to <0.621.0>)

[1/2/t/d/c/s/p/r/u/h/a]: 1
The error has been detected:
The problem is in pid <0.620.0>
while running receive
receive
  rst ->
    {port_rejected, Port};
  {Ack_PID, {syn_ack, Ack, Seq}} ->
    Ack_PID ! {self(), {Ack, Port, ack,
      Seq + 1}},
    Ack_PID ! {self(), {data, Port, Data}},
    {Seq + 1, Ack, Port, Data}
end
in tcp.erl:39
```

In summary, EDD has detected the exact location of both bugs using two debugging sessions, one for each bug. The former is shorter, only two questions, while the latter requires 5 questions. In both cases we have selected a suspicious process or event to start the debugging sessions. EDD has guided us to concrete isolated fragments of the computation, namely processes and receive expressions, whose validity is not trivial but easier to determine than considering the whole program. Moreover, we have not needed to modify the

program by instrumenting the code or to define properties and test cases to find the bugs.

## 4 EDD

We describe in this section the features of our Erlang Declarative Debugger, EDD. We describe the intuitive ideas underlying the debugger, the diagrams generated to ease the process, the questions performed by the tool, and the navigation strategies.

### 4.1 The ideas

The main ideas underlying declarative debugging are:

- A *debugging tree* representing the erroneous computation is built. The nodes of this tree represent the subcomputations that took place during the whole computation. Although details on this tree can be found in Section 5, from the user point of view it is more important to understand the computation that took place, described in Section 4.2, and the possible questions, shown in Section 4.3.
- This tree is traversed by using a *navigation strategy* and by asking questions to an oracle, usually the user. These strategies are described in Section 4.4.
- The navigation strategy looks for a node that (i) is invalid w.r.t. the behavior expected by the oracle and (ii) has only valid (w.r.t. the expected behavior) children. This is node the so-called *buggy node*, and it must point out the buggy code that generated the error in the computation. We describe in Section 4.5 the errors detected by EDD.

### 4.2 The diagrams

As we have introduced in the previous section, debugging concurrent applications is a tough process that requires the user to have in mind a lot of information about the different processes involved. In order to help the user EDD provides two different diagrams that can be used to answer the questions presented in the next section.

The first of these diagrams is the *communication and creation diagram*, as shown in Figure 6 in the previous section. It displays, for each process, the processes created, the messages sent and received, and the `receive` expressions executed.

The relation between processes is hierarchically displayed in the *creation tree*, as shown in Figure 7 for the example in Section 2. This tree depicts more succinctly the relations between process, helping the user to understand how processes are created.

### 4.3 The questions

We have developed EDD taking into account that, in a concurrent system with several processes, it is too difficult to apprehend the behavior of the whole system and hence it is easier for the user to consider the behavior of each specific process. However, when a process receives several messages it is also difficult to understand the complete execution of a process, so it is worth splitting it into smaller pieces that the user can easily examine. Following these premises, we consider a natural point to stop the execution those pieces of code where a function "stopped" because it could not progress any further without external information: `receive` expressions. Therefore, the questions presented by EDD focus on steps between these `receive` expressions, distinguishing the sequential part (the computed values) and the concurrent one (the message received, the messages sent, and the processes spawned). More specifically, our debugger asks questions of the form:

- *Is this evaluation correct?* This question asks whether, given a function call and the value for its arguments, the following values, obtained when executing the program, are correct: (i) the reached expression, which can be either a value or a `receive` expression, (ii) the messages sent, and (iii) the processes spawned. If all these values are correct then the node is valid; otherwise the node is invalid.
- *Is this transition correct with the given state?* This question evaluates how a function, currently evaluating a `receive` expression, should behave in a given state when a message is consumed. To simplify the presentation, and taking into account that Erlang, as described in Section 3, does not guarantee a certain delivery order when several processes are involved, EDD only displays the messages received from the process that sent the consumed message; we will show in Section 6 that this is enough to debug the system. Hence, EDD presents (i) the values bound to each variable in scope, (ii) the messages received thus far, (iii) the consumed message, and the results obtained once this message is consumed: (iv) the expression reached, (v) the messages sent, and (vi) the processes spawned.
  If (i) or (ii)[6] is found to be invalid by the user EDD discards the standard navigation strategies presented below and focuses on the functions that generated these values, which happened *before* this execution took place. If any result in (iii) - (vi) is incorrect then the node is invalid and the debugging pro-

[6] Note that (ii) can be wrong either because more messages were expected or because some of them are incorrect.

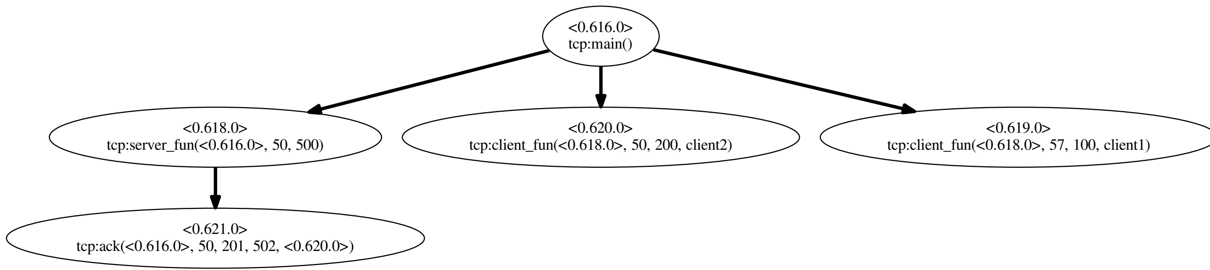


**Fig. 7** Creation tree for the example in Section 2, second debugging session.

cess continues with the standard navigation strategies, which will focus on the functions that generated these results *after* the question currently asked. Otherwise, the node is valid.

- *Is this* `receive` *expression being correctly executed?* Note that the questions above, if found invalid, would lead the debugger to a buggy function call (revealed by the first question). However, since `receive` expressions are the key feature used by concurrent Erlang systems we consider they deserve a more refined error detection, and hence EDD asks the user questions about the correctness of these expression. In particular, EDD presents the same question as the one described above for a specific `receive` expression, and process the answers in the same way.
- *Did you expect to reach a deadlock?* We find a particular case of the question above when no messages can be consumed. In this case EDD presents the `receive` expression, including (i) the current state of the variable and (ii) the messages sent to the current process by *all* processes, and ask the user to point out whether it is incorrect (i) to have this state for the variables, (ii) to receive these messages, or (iii) to fail to consume a message. If (i) or (ii) are incorrect then EDD focuses on the functions that generated these values, while for (iii) it focuses in the `receive` expression.

### 4.4 The navigation strategies

The tree can be traversed following different *navigation strategies*. EDD implements three different strategies:

Heaviest-first top-down. This strategy starts the navigation from the root and then asks about the correctness of its children. When an invalid children is found it is used as new root and, when all the children of the current root are correct, then a buggy node has been found.

Divide and query. This strategy asks in each step about the correctness of a node rooting a subtree with approximately half the size of the whole tree. If the node is correct then the subtree is discarded; otherwise, the subtree is set as current tree and the rest is discarded.

Process-directed. This strategy, developed specifically for EDD, requires the user to point out a process whose execution went wrong. The debugger will focus in this process until the error is found or another process is suspicious (because it sent an unexpected message), hence changing the focus to this process.

### 4.5 The detected errors

Buggy nodes point to erroneous parts of the code. EDD can find the following errors in Erlang programs:

Wrong functions. This error indicates that the body of a function is buggy. Note that EDD has already discarded errors in the calls performed in this function and in the `receive` expressions used, so the user can focus in the rest of the function.

Wrong `receive` expressions. This error indicates that a `receive` expression (identified by its line in the source code) is buggy. Since the function calls have already been checked by EDD, the user must check whether the patterns, the conditions, or the expressions used in the `receive` expression are correct.

## 5 A Calculus for Concurrent Erlang

The debugging tree employed by EDD is obtained from a proof tree in a suitable semantic calculus for concurrent Erlang programs, called *CEC* (Concurrent Erlang Calculus) and presented in this section. This calculus defines the semantics of Core Erlang [14, 15], a flattened and normalized version of Erlang used as an intermediate step during bytecode compilation.

### 5.1 Core Erlang

Core Erlang can be considered as a flat version of Erlang where the syntactic constructs have been reduced by removing syntactic sugar. It is interesting in our context, because it simplifies both the tool implementation and the theoretical results since they have to deal with a smaller and more uniform set of syntactic elements. Moreover, the use of Core Erlang does not impose any limitation in the debugging process, as the Core constructions can always be related back to their original Erlang fragments.

The complete syntax of Core Erlang and its behavior can be found in [14,15]. A Core Erlang program is a set of function definitions. Each function is defined by exactly one rule, so Erlang functions with various clauses are translated into a single rule that starts with a `case` expression that performs the dispatching. Other difference with Erlang is that in Core Erlang the arguments of function calls and `case` constructions are not arbitrary expressions but only values or variables, which have been previously computed by means of `let` expressions. Finally, sequence of Erlang instructions are translated into nested Core `let` expressions. Figure 8 shows the translation of the function `ack` (lines 21–32 in Figure 2) omitting some irrelevant parts. This function generates a Core function that takes 5 variables as arguments: `_cor4`, `_cor3`, ..., `_cor0`.[7] The sequence of `receive` expressions is translated as a `let` expression—note that the variable `_cor5` is not used in the body of the `let`. Finally, the Core program explicitly contains the implicit `after` clauses of `receive` expressions to set timeouts when receiving messages.

### 5.2 A calculus for concurrent Core Erlang

A process computation in *CEC* is represented as a tuple of the form $\ll pid, \langle expr, \theta\rangle, l \gg$, where:

- *pid* is the (unique) process identifier.
- *expr* is the expression being evaluated in the process. Occasionally the notation $E[expr]$ is used to represent evaluation contexts, indicating that the expression *expr* is a subexpression of $E$.
- $\theta$ is a substitution mapping variables to values and standing for the *context* where *expr* appears. This substitution is not necessary when the expression is a value. We denote by *id* the empty substitution.
- $l$ is the outbox for the messages sent by the process and not received by the addresee yet. It has the form $l \equiv [pid_1\ !\ val_1, \ldots, pid_m\ !\ val_m]$, indicating the order of the messages, which is important in the case of multiple repetitions of the same process identifier $pid_i$ in the list.

A set of process computations is called a *configuration* in our framework and represented as $\Pi$. An important idea in CEC is that of *medium-sized normal form* (*mnf*), which stands for expressions that cannot be further reduced by themselves. More specifically, an *mnf* can be either a value, represented by *val* in *CEC*, or a tuple $\langle expr, \theta\rangle$ with *expr* is a *receive expression*, that is an expression whose leftmost, innermost subexpression is a `receive` statement, and $\theta$ is a substitution with the *context* for *expr*. In Core Erlang, `receive` expressions can only be evaluated in `let` bindings, thus *expr* has the general form $expr \equiv$ `let` $x_1 = (\ldots($`let` $x_n =$ `receive ... end in` $e_n)\ldots)$ `in` $e_1$. Finally, we use *references* in some rules. References are unique identifiers that point to specific parts of the code, and they can be understood as a tuple containing the module name, the line, and the row in the source code; we use them to distinguish between the actual code and the behavior the user had in mind when writing the code, as explained in the next section. The *CEC* calculus proves three kinds of statements:

1. $\Pi \Rightarrow \Pi'$: indicates that the configuration $\Pi$ evolves into $\Pi'$ by evaluating the processes in $\Pi$ and (possibly) creating new processes.
2. $\langle expr, \theta\rangle \rightarrow (mnf, l, \Pi)$, with *expr* a non-`receive` expression, which indicates that *expr* has reached the medium-sized normal form *mnf*, sending the messages stored in the outbox $l$, and creating the new process computations in $\Pi$.
3. $\langle expr, \theta\rangle \xrightarrow{l,i} (mnf, l', \Pi)$, with *expr* a receive expression, which indicates that *expr* is evaluated to the medium-sized normal form *mnf* by receiving the $i$th message from an outbox $l$.

A special case of the first type inference is $\Pi \Rightarrow$ *endlock*, meaning that a non-empty subset of the configuration is blocked in `receive` statements with no possibility to continue. The third statement type also has a special case with the form $\langle expr, \theta\rangle \xrightarrow{l,0}$ *lock*, which indicates that no message can be consumed. Although the three statements occur in the calculus, only statements two and three are used in the debugging trees since the validity of statements of type one will be inferred readily from the validity of statements of type two and three.

In particular, statements of type two correspond to non-`receive` expressions that can be evaluated to *mnf* without consuming any message, while the type three

[7] Variables starting with an underscore (_) are generated by the Erlang-to-Core translation.

```
1  'ack'/5 =
2    fun (_cor4,_cor3,_cor2,_cor1,_cor0) ->
3      let _cor5 = receive
4        {_cor10,_cor11,'ack',_cor12,_cor13} when ... ->
5          'ok'
6        _cor20 when 'true' ->
7          call 'erlang':'!'(_cor4, {'data','error_ack'})
8        after 'infinity' ->
9          'ok'
10     in receive
11       {_cor21,_cor22,'data',D} when ... ->
12         call 'erlang':'!'(_cor4, {'data',D})
13       _cor25 when 'true' ->
14         call 'erlang':'!'(_cor4, {'data','error_data'})
15       after 'infinity' ->
16         'true'
```

**Fig. 8** Translation of the `ack` function into Core Erlang

stands for receive expressions that progress by consuming a message.

### 5.3 Matching

In Erlang, variables are bound to values through the pattern matching mechanism. The result of the following syntactic matching function is a substitution $\theta$. The case where the matching fails is represented by $\perp$.

**Definition 1 (Syntactic matching)**
$match(var, val) = [var \mapsto val]$
$match(lit_1, lit_2) = id$, if $lit_1 \equiv lit_2$
$match([pat_1|pat_2], [val_1|val_2]) = \theta_1 \uplus \theta_2$,
 where $\theta_i \equiv match(pat_i, val_i)$
$match(\{pat_1, \ldots, pat_n\}, \{val_1, \ldots, val_n\}) = \theta_1 \uplus \ldots \uplus \theta_n$
 where $\theta_i \equiv match(pat_i, val_i)$
$match(var = pat, val) = \theta[var \mapsto val]$,
 where $\theta \equiv match(pat, val)$
$match(_, _) = \perp$, if none of the previous rules apply.

The operator $\uplus$ stands for the union of two substitutions with disjunct domain. It verifies:

$$\perp \uplus\ \theta = \theta\ \uplus \perp = \perp$$

for any substitution $\theta$. Notice that the translation to Core Erlang generates patterns without duplicated variables, therefore the substitutions $\theta_i$ obtained for compound patterns must have disjoint domains and $\uplus$ can be applied. From this definition of match it is easy to define when a `receive` branch fails:

**Definition 2 (Failing `receive` branch)**
We say that the expression $fails(b,\ val,\ \theta)$ holds, with $b$ a `receive` branch of the form

$$pat \texttt{ when } expr \texttt{ -> } expr'$$

iff $\theta' \equiv match(pat\theta, val)$ verifies either

1. $\theta' = \perp$, or
2. $\theta' \neq \perp$, $\theta'' \equiv \theta\ \uplus\ \theta'$, and $\parallel expr\theta'' \parallel \to$ `'false'`

The definition of *fails* can be extended to check whether a `receive` statement fails to accept a message of a list of messages, and also to define when a message in a list is accepted:

**Definition 3 (Failing and succeeding `receive`)**
Let $r$ be a reference to a `receive` statement of the form:

$$\texttt{receive}\quad pat_1 \texttt{ when } expr'_1 \texttt{ ->}^{r_1} expr''_1$$
$$\ldots$$
$$pat_n \texttt{ when } expr'_n \texttt{ ->}^{r_n} expr''_n \texttt{ end}$$

and $l$ a list of messages, $l \equiv [val_1, \ldots val_m]$. Then we say that:

- $fails(r,\ val,\ \theta)$ holds iff every branch $b$ in $r$,

$$b \equiv pat \texttt{ when } expr \texttt{ -> } expr'$$

 in $r$, verifies that $fails(val, p.r, \theta)$
- $fails(r,\ l,\ \theta)$ holds iff $fails(r,\ val_i, \theta)$ for every $1 \leq i \leq m$.
- $succeeds(r,\ l, j, \theta) \to expr'_k\theta''$ holds iff
 1. $1 \leq j \leq m$, $1 \leq k \leq n$.
 2. $fails(p.r,\ [val_1, \ldots, val_{j-1}], \theta)$.
 3. $fails(b_i,\ val_j, \theta)$ for every $i = 1 \ldots k-1$, with $b_i$ the $i$th branch of the `receive` expression referenced by $r$.
 4. Let $b_k$ be the $k$th branch of the `receive` statement referenced by $r$,

$$b_k \equiv pat_k \texttt{ when } expr_k \texttt{ -> } expr'_k$$

 Then, $\theta' \equiv match(pat\theta, val_j)$ verifies
 - $\theta' \neq \perp$,
 - $\theta'' \equiv \theta\ \uplus\ \theta'$, and
 - $\parallel expr\theta'' \parallel \to$ `'true'`

$$(\mathsf{PROC})\ \frac{\langle expr, \theta\rangle \to (mnf, l', \Pi')}{\Pi, \ll pid, \langle expr, \theta\rangle, l \gg \Rightarrow \Pi, \ll pid, mnf, l + l' \gg, \Pi'}$$

$$(\mathsf{CONSUME_1})\ \frac{\langle expr, \theta\rangle \xrightarrow{l'_2, j} (mnf', l', \Pi')}{\Pi, \ll pid_1, \langle expr, \theta\rangle, l_1 \gg, \ll pid_2, mnf_2, l_2 \gg \Rightarrow \Pi, \Pi' \ll pid_1, mnf', l_1 + l' \gg, \ll pid_2, mnf_2, l''_2 \gg}$$

where $l'_2 \equiv l_2|_{pid_1}$, $1 \leq j \leq |l'_2|$, $i$ the position of the $j$th message of $l'_2$ in $l_2$, $l''_2$ is $l_2$ after removing the $i$th message.

$$(\mathsf{CONSUME_2})\ \frac{\langle expr, \theta\rangle \xrightarrow{l', j} (mnf', l_1, \Pi')}{\Pi, \ll pid, \langle expr, \theta\rangle, l \gg \Rightarrow \Pi, \Pi' \ll pid, mnf', l'' + l_1 \gg}$$

where $l' \equiv l|_{pid}$, $1 \leq j \leq |l'|$, $i$ the position of the $j$th message of $l'$ in $l$, $l''$ is $l$ after removing the $i$th message.

$$(\mathsf{Tr})\ \frac{\Pi \Rightarrow \Pi_1 \qquad \Pi_1 \Rightarrow \Pi'}{\Pi \Rightarrow \Pi'}$$

$$(\mathsf{RCV_1})\ \frac{\langle expr_1, \theta\rangle \xrightarrow{l_1, j} (val, l, \Pi) \qquad \langle expr_2\theta', \theta'\rangle \to (mnf, l', \Pi')}{\langle \mathtt{let}\ var\ \mathtt{=}\ expr_1\ \mathtt{in}\ expr_2, \theta\rangle \xrightarrow{l_1, j} (mnf, l + l', (\Pi, \Pi'))}$$

where $\theta' \equiv \theta \uplus \{var \mapsto val\}$.

$$(\mathsf{RCV_2})\ \frac{\langle expr_1, \theta\rangle \xrightarrow{l_1, j} (\langle mnf, \theta'\rangle, l, \Pi)}{\langle \mathtt{let}\ var\ \mathtt{=}\ expr_1\ \mathtt{in}\ expr_2, \theta\rangle \xrightarrow{l_1, j} (expr', l, \Pi)}$$

where $mnf$ is not a value and $expr' \equiv \langle \mathtt{let}\ var\ \mathtt{=}\ mnf\ \mathtt{in}\ expr_2, \theta'\rangle$.

$$(\mathsf{RCV_3})\ \frac{succeeds(r, l, j, \theta) \to expr\theta' \qquad \langle expr\theta', \theta'\rangle \to (mnf, l', \Pi)}{\langle r, \theta\rangle \xrightarrow{l, j} (mnf, l', \Pi)}$$

where $val_j \in l$ and $r$ is the reference to a `receive` statement, and *succeeds* defined as in Definition 3

**Fig. 9** Rules for processes I

### 5.4 Outboxes versus inboxes

An important novelty of our calculus is that it employs outboxes for message passing instead of inboxes. The majority of semantics [22,26,21,45] consider that every process contains an inbox where incoming messages are stored. In these semantics, when a Erlang bang (!) instruction is executed to send a message $m$ to a process $pid$, $m$ is *immediately* enqueued to the inbox of process $pid$. In *single-node* environments this behavior is acceptable, since the communication delay is insignificant. However, in a distributed system with multiple nodes, messages can suffer different delays. One of the fundamental ideas behind Erlang is that *message passing between a pair of processes is assumed to be ordered* [4], but the behavior of messages from different processes is not guaranteed, so any possibility must be supported by the semantic calculus. This distributed situation is explained in detail in [43] by means of an example: suppose three processes $P_1$, $P_2$, and $P_3$, each one running in a different node. Process $P_1$ sends message `hello` to $P_2$, *and then* sends `world` to $P_3$. Process $P_3$ simply resends to $P_2$ any message that it receives. The key question is: in this scenario, which message will arrive first to $P_2$? In single-node semantics the message `hello` will arrive always before `world`, as it will be inserted immediately into $P_3$ mailbox, but in a multi-node setting any of the two mesagges can be received first. The semantic calculus presented in this section is not limited to single-node environments, because sent messages are inserted into the sender outbox instead of the destination inbox—see rule (BANG) in Figure 10. When a process reads a message using the `receive` statement, it will fetch *the first message* destined for it *from any outbox*—rules ($\mathsf{CONSUME_*}$) in Figure 9. In the example, when $P_3$ executes the `receive` statement it can fetch `hello` from the $P_1$ outbox or `world` from $P_2$ outbox, since both process have messages to $P_3$.

In our setting outboxes are represented as lists with elements of the form *pid ! message*. In the example above, after sending the two messages, `hello` to $P_2$ and `world` to $P_3$, and assuming that no message has ben consumed yet, the outbox of $P_1$ is the list $l \equiv [P_2\ !\ \mathtt{hello}, P_3\ !\ \mathtt{world}]$. In the calculus we use the notation $|l|$ to represent the number of elements in an outbox list, $l + l'$ to indicate the concatenation, and $l|_P$ to indicate the restriction of $l$ to the messages for process $P$ and containing only the messages, not the addressee. For instance, in the same example $l|_{P_3} = [\mathtt{world}]$.

$$(\mathsf{ENDLOCK})\ \frac{\langle expr_1, \theta_1\rangle \xrightarrow{l'_1, 0} lock \quad \ldots \quad \langle expr_k, \theta_k\rangle \xrightarrow{l'_k, 0} lock}{\Pi \Rightarrow endlock}$$

$\Pi \equiv \ll pid_1, mnf_1, l_1 \gg, \ldots, \ll pid_k, mnf_k, l_k \gg, \ll pid_{k+1}, val_{k+1}, l_{k+1} \gg, \ldots, \ll pid_n, val_n, l_n \gg$,
$k > 0$, $l'_i \equiv l_1 + \ldots + l_n$ restricted to messages addressed for $pid_i$ for $i = 1 \ldots k$, and $mnf_i \equiv \langle expr_i, \theta_i\rangle$, with $1 \le i \le k$.

$$(\mathsf{LOCK})\ \frac{fails(r,\ l,\ \theta)}{\langle expr[r], \theta\rangle \xrightarrow{l, 0} lock}$$

where $r$ is reference to a leftmost innermost `receive` statement to be evaluated in $expr$ and $fails$ as defined in Definition 3

$$(\mathsf{SPAWN})\ \frac{\langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n, \theta\rangle \to (val_n, l_n, \Pi_n)}{\langle spawn(m, f, [expr_1, \ldots, expr_n]), \theta\rangle \to (pid, l, \Pi)}$$

with `f` defined as $\mathtt{f} = \mathtt{fun}(var_1, \ldots, var_n)\ \texttt{->}\ expr$, $l \equiv l_1 + \ldots + l_n$, $\Pi \equiv (\Pi_1, \ldots, \Pi_n, \ll pid, \langle r_f, \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}\rangle, [\,] \gg)$, $r_f$ a reference to the function `f`, and $pid$ a new process identifier.

$$(\mathsf{BANG})\ \frac{\langle expr_1, \theta\rangle \to (pid, l_1, \Pi_1) \quad \langle expr_2, \theta\rangle \to (val, l_2, \Pi_2)}{\langle expr_1\,!\,expr_2, \theta\rangle \to (val, l_1 + l_2 + [pid\,!\,val], (\Pi_1, \Pi_2))}$$

**Fig. 10** Rules for processes II

5.5 Inference rules

The inference rules defining the *CEC* calculus are shown in Figures 9, 10, and 11. To ease the description we only present the fragment of rules relevant for debugging. A complete description, including the inference rules dealing with exception propagation, can be found at [9].

Figure 9 presents the rules for evolving processes, combining steps, and evaluating expressions after receiving a message. The former may occur by either evaluating the inner expressions or by consuming a message. More specifically, we define the following rules:

- (PROC). This rule indicates how configurations can evolve by processing non-`receive` expressions. This is done by selecting a process $pid$ with a current non-`receive` expression, and performing a evaluation step, which yields to a tuple $(mnf, l', \Pi')$. The values $l'$ and $\Pi'$ refer to the list of messages sent and the new processes created during the computation step, respectively. The list $l'$ is incorporated to the output list of $pid$, while the new processes $\Pi'$ are included in the configuration set.
- ($\mathsf{CONSUME}_1$) and ($\mathsf{CONSUME}_2$). Configurations also evolve when a process that is currently evaluating a receive expression consumes a message from an outbox. The rule ($\mathsf{CONSUME}_1$) indicates that this message is taken from the outbox of another process, while ($\mathsf{CONSUME}_2$) illustrates the case where the message is taken from the outbox of the same process that consumes the message (the so called *self* messages).
- (Tr). The transitivity rule combines the inferences for evolving processes and consuming messages.
- ($\mathsf{RCV}_1$) and ($\mathsf{RCV}_2$). The rules for evaluating receive expressions take advantage of the Core transformation sketched before, since this expression is only nested in `let` expressions. The rule ($\mathsf{RCV}_1$) reduces the argument in the `let` expression, when it is evaluated to a value, and continues the evaluation with the body by using the appropriate substitution; otherwise, ($\mathsf{RCV}_2$) places the new lock in the `let` expression.
- ($\mathsf{RCV}_3$). Once the `receive` statement is reached, *succeeds* returns the body of the computed `receive` branch that accepts the message j (otherwise this rule cannot be applied), including $\theta'$ as an extension of $\theta$ including the binding due to the mathching. The right-hand side premise of the inference evaluates the expression until it reaches a medium-sized normal form.

It is important to note that, as ($\mathsf{CONSUME}_*$) rules take from the outboxes the first (oldest) message directed to the process, they guarantee the order of messages between a pair of processes. They also solve the problem shown by the example in Section 5.4. In this example, a possible configuration after $P_1$ sends message `hello` to $P_2$, `world` to $P_3$, and assuming that no message has been consumed yet is:

$$\begin{array}{l}\ll P_1, mnf_1, [P_2\ !\ \texttt{hello}, P_3\ !\ \texttt{world}] \gg,\\ \ll P_2, mnf_2, l_2 \gg,\\ \ll P_3, mnf_3, l_3 \gg\end{array}$$

Figure 10 shows the rules in charge of detecting deadlocks, as well as the rules for sending messages and creating processes. More specifically:

- The rule (ENDLOCK) characterizes locks. It requires that all the processes contain either a value or an expression in medium-sized normal form which is

locally locked, that is, there are no messages that can be accepted by the `receive` expression that must be evaluated next.
- The rule (LOCK) is in charge of checking that a particular process cannot receive any of the messages addressed to it.
- The rule (SPAWN) shows that a *spawn* expression is reduced to the new process identifier, creating a new process during its execution Note that the evaluation of the subexpressions may generate new messages and processes, which are added to the final tuple.
- Similarly, the rule (BANG) indicates that a *bang* is evaluated to the sent message, which is also returned to add it into the outbox.

We present in Figure 11 the main rules for sequential evaluation:

- Function references are evaluated by the (BFUN) rule, which just evaluates the reference to the function with the given substitution for its arguments. This rule will be invoked by the following (APPLY) and (CALL) rules and its main goal is to abstract the actual function application in a common node of the resulting debugging tree—see Section 6.
- The rule (APPLY) indicates that first we need to obtain the name of the function, which must be defined in the current module `r` (extracted from the reference to the reserved word `apply`) and then compute the arguments of the function. Finally the function, described by its reference, is evaluated using the substitution obtained by binding the variables in the function definition to the values for the arguments.
- Similarly, the rule (CALL) evaluates a function defined in another module.
- The rule ($\mathsf{LET}_1$) evaluates the inner expression in a `let` expression to a value, binds this value to the appropriate variable, and continues by evaluating the body of the expression.
- In contrast to the rule ($\mathsf{LET}_1$), the rule ($\mathsf{LET}_2$) cannot evaluate the inner expression to a value but a medium-sized normal form. Hence, it just updates the expression but does not continue with the body of the expression.

In the rest of the paper we will use the notation $\mid e \mid$ for the translation of the expression $e$ to Core Erlang. It is extended to medium-sized normal forms as $|val| = val$ and $|\langle expr, \theta\rangle| = \langle |expr|, \theta\rangle$, and to configurations as $\mid \ll pid_1, e_1, l_1 \gg \ldots \ll pid_n, e_n, l_n \gg \mid = \ll pid_1, |e_1|, l_1 \gg \ldots \ll pid_n, |e_n|, l_n \gg$.

5.6 Evaluating expressions using the calculus

We present here a simple example to illustrate how the calculus works, focusing on how medium-sized normal forms are obtained and solved. Assume we have the following Erlang program. Here, we implement a dummy function `f` that just returns the value computed by `g` when using the same parameter as `f`; `g` creates a new process that will execute function `i` taking as parameter the identifier of the process being currently executed, keeps the process identifier of this new process, and then returns the value generated by `h` when receiving the same parameter as `g` and the new *pid*; `h` waits for a message from the new process created in `g` and, once such a message is received, returns the value obtained by adding the message and the input argument; finally, the function `i` just sends a tuple with the number `3` and its own process identifier to the *pid* received as argument:

```
1  f(X) -> g(X) .
2
3  g(X) -> Pid = spawn(?MODULE,i,[self()]),
4          h(X, Pid) .
5
6  h(X, Pid) ->
7   receive
8     {Pid, M} -> M + X
9   end.
10
11 i(X) -> X ! {self(), 3} .
```

The code above is translated into Core Erlang by following the ideas discussed in Section 5.1. Function `f` just applies function `g`; function `g` requires two `let` expressions for evaluating the function `self()`, evaluating `spawn`, and finally calling `h` with variables; function `h` is translated as a `receive` expression with function calls and a default `after` branch; finally, function `i` uses a `let` expression for evaluating `self()` and sending the message using only variables and constants:

```
1  'f'/1 = fun (_cor0) -> apply 'g'/1(_cor0)
2
3  'g'/1 = fun (_cor0) ->
4   let <_cor1> =
5       call 'erlang':'self'()
6   in let <Pid> =
7        call 'erlang':'spawn'('test','i',[_cor1|[]])
8      in apply 'h'/2(_cor0, Pid)
9
10 'h'/2 = fun (_cor1,_cor0) ->
11  receive <{M,_cor4}>
12      when call 'erlang':'=:='(_cor0, _cor4) ->
13           call 'erlang':'+'(M, _cor1)
14      after 'infinity' -> 'true'
15
16 'i'/1 = fun (_cor0) ->
17  let <_cor1> =
18     call 'erlang':'self'()
19  in
```

$$(\mathsf{BFUN})\ \frac{\langle expr\theta, \theta\rangle \to (mnf, l, \Pi)}{\langle f, \theta\rangle \to (mnf, l, \Pi)}$$

where $f$ references a function `f` defined as $\texttt{f} = \texttt{fun}(var_1, \ldots, var_n)$ `->` $expr$

$$(\mathsf{APPLY})\ \frac{\begin{array}{c}\langle expr, \theta\rangle \to (Atom/n, l, \Pi) \\ \langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n, \theta\rangle \to (val_n, l_n, \Pi_n) \\ \langle r_f, \theta'\rangle \to (mnf, l', \Pi')\end{array}}{\langle \texttt{apply}^r\ expr(expr_1, \ldots, expr_n), \theta\rangle \to (mnf, nl, n\Pi)}$$

where $Atom/n$ is a function defined in the module $r.mod$ as $Atom/n$ = `fun` $(var_1, \ldots, var_n)$ `->` $expr$, $r_f$ its reference, $\theta' \equiv \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$, $nl \equiv l + l_1 + \ldots + l_n + l'$, and $n\Pi \equiv (\Pi, \Pi_1, \ldots, \Pi_n, \Pi')$.

$$(\mathsf{CALL})\ \frac{\begin{array}{c}\langle expr_{n+1}, \theta\rangle \to (Atom_1, l, \Pi) \quad \langle expr_{n+2}, \theta\rangle \to (Atom_2, l', \Pi') \\ \langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n, \theta\rangle \to (val_n, l_n, \Pi_n) \\ \langle r_f, \theta'\rangle \to (mnf, l'', \Pi'')\end{array}}{\langle \texttt{call}\ expr_{n+1}\texttt{:}expr_{n+2}(expr_1, \ldots, expr_n), \theta\rangle \to (mnf, nl, n\Pi)}$$

where $Atom_2/n$ is a function defined as $Atom_2/n$ = `fun` $(var_1, \ldots, var_n)$ `->` $expr$ in the $Atom_1$ module ($Atom_1$ must be different from the built-in module `erlang`), $r_f$ its reference, $\theta' \equiv \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$, $nl \equiv l + l' + l_1 + \ldots + l_n + l''$, and $n\Pi \equiv (\Pi, \Pi', \Pi_1, \ldots, \Pi_n, \Pi'')$.

$$(\mathsf{LET}_1)\ \frac{\langle expr_1, \theta\rangle \to (val, l, \Pi) \quad \langle expr_2\theta', \theta'\rangle \to (mnf, l', \Pi')}{\langle \texttt{let}\ var\ \texttt{=}\ expr_1\ \texttt{in}\ expr_2, \theta\rangle \to (mnf, l + l', (\Pi, \Pi'))}$$

where $\theta' \equiv \theta \uplus \{var \mapsto val\}$.

$$(\mathsf{LET}_2)\ \frac{\langle expr_1, \theta\rangle \to (\langle mnf, \theta'\rangle, l, \Pi)}{\langle \texttt{let}\ var\ \texttt{=}\ expr_1\ \texttt{in}\ expr_2, \theta\rangle \to (\langle \texttt{let}\ var\ \texttt{=}\ mnf\ \texttt{in}\ expr_2, \theta'\rangle, l, \Pi)}$$

where $mnf$ is not a value.

**Fig. 11** Rules for sequential Erlang

```
20      call 'erlang':'!'(_cor0, {_cor1, 3})
```

Assume we want to compute the value for `f(4)`. In our framework, we consider this program is executed in three steps:

1. `f(4)` is evaluated while possible, which means that `g(4)` is called. In turn, `g(4)` evaluates `self()` to a value, e.g. $p_1$, creates a new process with ID $p_2$ that will execute `i(`$p_1$`)`, and calls `h(4,` $p_2$`)`, which does not finish because there are no messages for this process. Note that, right now, the call to `f(4)` has been reduced to the `receive` expression in `h`.
2. The new process $p_2$ sends the message `{`$p_2$`,4}` to $p_1$ and it is evaluated to `{`$p_2$`,4}`.
3. The process $p_1$ receives the message sent in the previous step, hence evaluating the `receive` expression we obtained in the first step.

The proof tree in Figure 12(top) depicts the inference for $\langle r_f, \theta_0\rangle \to mnf$, where $r_f$ is the reference to function `f`, $mnf \equiv \langle \texttt{receive}_h, \{\texttt{_cor0} \mapsto p_2, \texttt{_cor1} \mapsto 4\ \}$, $\theta_0 \equiv \{\texttt{_cor0} \mapsto 4\}$, and $\Pi \equiv \ll p_2, \langle r_i, \{\texttt{_cor0} \mapsto p_1\}\rangle, [\,] \gg$ We briefly describe the different nodes, starting from the root:

– The root applies the rule (BFUN) to evaluate the reference to the function `f`.
– Since the body of the function `f` corresponds to the application of another function, we use the rule (APPLY) to evaluate it.
– The rule for applying a function first reduce the function name (`'g'/1`) and the parameters (`4`). Since this inference is trivial in this case, we have substituted the corresponding trees by $\bigtriangledown_0$ and $\bigtriangledown_1$, respectively. The third premise required by the rule requires to evaluate the reference to `g`.
– In this case, the body of the function `g` consists of a `let` expression (identified as $\texttt{let}_4$ because it is the `let` expression located in line 4 in the core program above) whose argument can be fully evaluated, so it is inferred with the ($\mathsf{LET}_1$) rule.
– The argument of the `let` expression is evaluated to a value ($p_1$, abbreviated in the subtree $\bigtriangledown_3$) while its body is another `let` expression (identified as $\texttt{let}_6$, since it is located at line 6. Hence, we have $\theta_1 \equiv \theta_0 \cup \{\texttt{_cor1} \mapsto p_1\}$ for evaluating $\texttt{let}_6$ with the rule ($\mathsf{LET}_1$).
– This inference is computed by first using (SPAWN) for generating a new process and then using (APPLY) with the substitution $\theta_2 \equiv \theta_1 \cup \{\texttt{PID} \mapsto p_2\}$.
– Finally, using the substitution $\theta_3 \equiv \{\texttt{_cor0} \mapsto pid_2, \texttt{_cor1} \mapsto 4\}$ we can apply (BFUN), which just substitutes the call to the function by its body.

The proof tree in Figure 12(bottom) continues the execution by evaluating the function in process $p_2$:

- The (PROC) rule evaluates the reference generated by `spawn` in the previous computation.
- As described for the previous tree, the (BFUN) rule substitutes the reference by the body of the function in the premises to compute the final result. Since in this case we have a `let` expression, we will use the rule ($\mathsf{LET}_1$).
- The rule ($\mathsf{LET}_1$) evaluates `self()` (in $\bigtriangledown_4$) and then evaluates the bang expression with the substitution $\theta_4 \equiv \{\texttt{_cor0} \mapsto p_1, \{\texttt{_cor1} \mapsto p_2\}$.
- Finally, the message is sent just by evaluating the expressions to be sent, the atoms 3 and $p_2$, which is computed in $\bigtriangledown_5$ and $\bigtriangledown_6$, respectively.

Finally, we sketch the final tree in Figure 13, where we reuse the variables for substitutions, expressions, and configurations from Figure 12. We also use $T_1$ for the tree in Figure 12(top), $T_2$ for the tree in Figure 12(bottom), $\mathit{\Pi}_i \equiv \ll p_1, mnf, [] \gg \ll p_2, \{3, \mathtt{p_2}\}, p_1 \; ! \; \{3, \mathtt{p_2}\} \gg$, and $\mathit{\Pi}_f \equiv \ll p_1, 7, [] \gg \ll p_2, \{3, \mathtt{p_2}\}, p_1 \; ! \; \{3, \mathtt{p_2}\} \gg$. In this tree we just use the transitivity rule to put together the steps explained above for this program. First we evaluate the function in the initial process, then we evaluate the new process, and finally we consume the message by using ($\mathsf{CONSUME}_1$). Note that we use ($\mathsf{RCV}_3$), where $l \equiv [p_1 \; ! \; \{3, \mathtt{p_2}\}]$, to consume the only message in the list. The tree $\bigtriangledown_7$ is used to compute *succeeds*, while $\bigtriangledown_8$ evaluates the final expression.

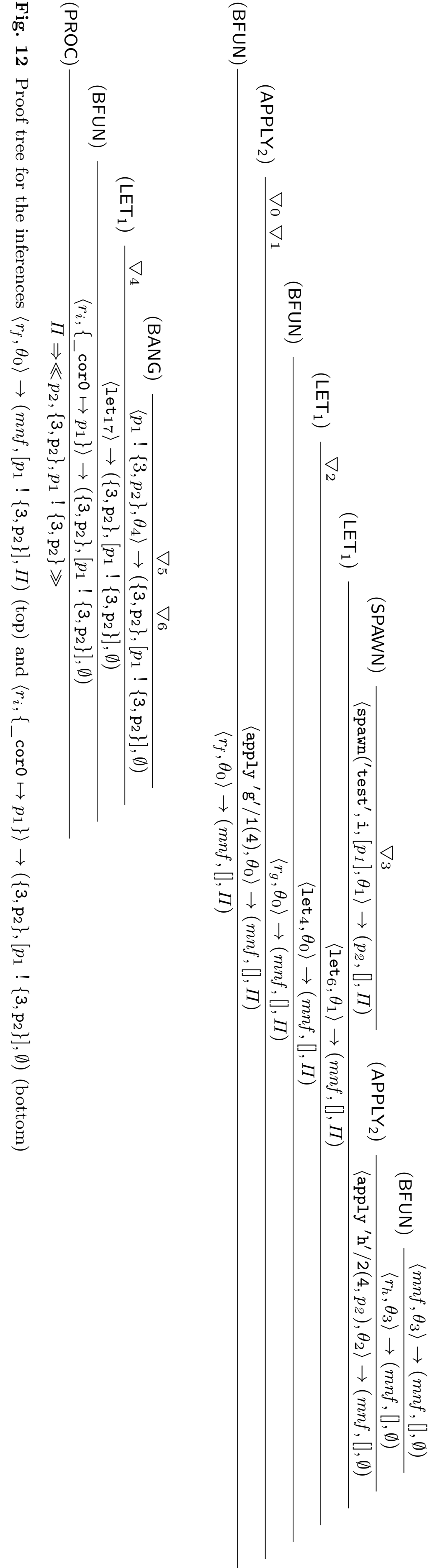

**Fig. 12** Proof tree for the inferences $\langle r_f, \theta_0 \rangle \to (mnf, [p_1 \; ! \; \{3, \mathtt{p_2}\}], \mathit{\Pi})$ (top) and $\langle r_i, \{\texttt{_cor0} \mapsto p_1\} \rangle \to (\{3, \mathtt{p_2}\}, [p_1 \; ! \; \{3, \mathtt{p_2}\}], \emptyset)$ (bottom)

## 6 Theoretical Basis

In this section we define formally the errors that our proposal can detect and the correctness of the proposal. This is done by defining first the concept of *intended interpretation*, represented as $\mathcal{I}$, which corresponds to the behavior expected by the user for the basic pieces of the program. Choosing these basic pieces is important. On the one hand, small pieces of code can lead to very precise errors, but also to a large number of complicated questions during the debugging session. On the other hand, if the piece of code is very large, debugging sessions would be easier but the error would not very informative.

In the case of distributed Erlang programs we choose two basic pieces of code: user functions and receive statements. In the following we denote by $f$ a reference to a user function, by $r$ a reference to a receive statement, and by $p$ either $r$ or $f$.

We assume that the user knows the expected behavior of these pieces. This behavior is then extrapolated

$$
(\mathsf{Tr})\,\dfrac{(\mathsf{PROC})\,\dfrac{T_1}{\ll p_1, \langle r_f, \theta_0 \rangle, [\,] \gg \Rightarrow \ll p_1, mnf, [\,] \gg \Pi} \qquad (\mathsf{Tr})\,\dfrac{T_2 \qquad (\mathsf{CONSUME_1})\,\dfrac{(\mathsf{RCV_3})\,\dfrac{\triangledown 7 \qquad \triangledown 8}{mnf \xrightarrow{l,0} (7, [\,], \emptyset)}}{\Pi_i \Rightarrow \Pi_f}}{\ll p_1, mnf, [\,] \gg \Pi \Rightarrow \Pi_f}}{\ll p_1, \langle r_f, \theta_0 \rangle, [\,] \gg \Rightarrow \Pi_f}
$$

**Fig. 13** Proof tree for the inference $\ll p_1, \langle r_f, \theta_0 \rangle, [\,] \gg \Rightarrow \Pi_f$

to complete programs by suitable calculi *ICEC* presented in this section. The discrepancies between the results produced by the program, represented by *CEC* and the results expected by the user, represented by *ICEC*, define the buggy parts of the program.

## 6.1 Intended Interpretations

The intended interpretation of a program $P$ is defined as the union of two sets:

$$\mathcal{I} = \mathcal{I}_{fun} \;\cup\; \mathcal{I}_{rcv}$$

The first set defines the expected values for function calls, and has the form:

$\mathcal{I}_{fun} = \{\ldots,\; \langle f, \theta \rangle \to (mnf, l, \Pi),\; \ldots\}$ where

- $f$ is a reference to a function defined as `f/n = fun (` $var_1$ `, ... ,` $var_n$ `) ->` *exprs*.
- The domain of the substitution $\theta$ must be $\{var_1, \ldots, var_n\}$.

Thus, $\mathcal{I}_{fun}$ contains the results expected for any possible function call to program functions. This notion can be easily extended to lambda abstractions, which are not discussed here por simplicity. The other set conforming $\mathcal{I}$ refers to the intended behavior of `receive` expressions and it is a set union of the form:

$$\mathcal{I}_{rcv} = \bigcup_{\langle\, p.r,\, l,\, \theta \,\rangle} \mathcal{I}_{\langle\, p.r,\, l,\, \theta \,\rangle}$$

With $p$ any program piece of code, $r$ any `receive` statement reachable from $p$ during a computation, $\theta$ a variable substitution, and $l$ a list of incoming messages. $\mathcal{I}_{\langle\, p.r,\, l,\, \theta \,\rangle}$ represents the expected behavior of $p$ when stopped in $r$ and in a presence of the incoming messages $l$. As usual, $\theta$ represents the context, that is, the binding of variables in $p.r$ occurred so far.

Asuming that $l$ is the incoming list for the process computing $p.r$, we can distinguish two possible forms for each set $\mathcal{I}_{\langle\, p.r,\, l,\, \theta \,\rangle}$:

1. $Ipr = \{\mathcal{I}fails(r, l, \theta)\}$ if the user expected the process to be blocked.
2. $Ipr = \{\mathcal{I}succeeds(p.r,\, l,\, j,\, \theta) \to (mnf, l', \Pi)\}$ if the computation of $p$ is expected to continue consuming the $j$th message of $l$, reaching the new medium-sized normal form $mnf$, producing in between the list of output messages $l'$, and the new processes $\Pi$. It is worth noticing that $l$ is a list of input messages to this processes of the form $[val_1, \ldots, val_n]$, while $l'$ is an output box list of the form

$$l' \equiv [pid_1 \;!\; val_1, \ldots, pid_m \;!\; val_m]$$

A particular case of $Ipr$ are the sets $Irr$ that contain the expect behavior of the `receive` statement without taking into account the piece of code where the computation has been originated.

## 6.2 Intended Semantic Calculus

The validity of the nodes in a *CEC*-proof tree is obtained by defining the *intended interpretation calculus*. This calculus, called *ICEC* is described in the following definition:

**Definition 4** The calculus *ICEC* contains the same inference rules as *CEC*, prefixing the label of each rule by $\mathcal{I}$ to avoid confusion. The definitions of the rules are also the same in both calculi, except by:

1. ($\mathcal{I}$BFUN), which adds the following additional side condition to (BFUN):

$$\langle f, \theta \rangle \to (mnf, l, \Pi) \;\in\; \mathcal{I}_{fun}$$

2. ($\mathcal{I}$RCV$_1$) also adds to (RCV$_1$) a new side condition of the form

$$succeeds_{\mathcal{I}}(p.r,\, l_1,\, j,\, \theta) \to R$$

where:
- $R$ is the right-hand side of the statement that can be found as conclusion of the inference rule (RCV$_1$) in Figure 9.
- $r$ a reference to the leftmost, innermost `receive` in the left-hand side of the conclusing of the inference rule (RCV$_1$).
- $p$ is the smallest (closest) piece of code (either a function or a `receive` statement) that contains the `let` statement.

3. ($\mathcal{I}\mathsf{RCV}_3$) adds to ($\mathsf{RCV}_3$), the new side condition:

$$(\mathcal{I}succeeds(r.r,\ l,\ j,\ \theta) \to (mnf, l', \Pi)) \in \mathcal{I}$$

4. ($\mathcal{I}\mathsf{LOCK}$), which adds a new side condition $\mathcal{I}fails(r, l, \theta) \in \mathcal{I}$ to ($\mathsf{LOCK}$).

The differences between *ICEC* and *CEC* are easy to understand. The first point forces ($\mathcal{I}$_$\mathsf{BFUN}$) to check whether the first step of a function call is a valid statement in $\mathcal{I}$. Analogously, ($\mathcal{I}$_$\mathsf{RCV}_1$) and ($\mathcal{I}$_$\mathsf{RCV}_3$) only allow expected transitions. In the rule ($\mathcal{I}$_$\mathsf{RCV}_1$) we only check the outer `let` expression, but other inner expressions are also checked while evaluating the premises of the inference rule. It is worth observing that the closest piece of code can be either the function $f$ that contains the associated Erlang code (the code that has given raise to the `let`), or a `receive` statement defined in $f$, which contains the `let` in the body of some rule. Finally, ($\mathcal{I}\mathsf{LOCK}$) ensure that failures are only accepted if expected in the intended interpretation.

Summarizing, we can say that *ICEC* only computes intended values. Analogously to the case of *CEC*, the notation

$$ICEC \models_{(P,\mathcal{I},T)} \mathcal{E}$$

indicates that the evaluation $\mathcal{E}$ can be proven w.r.t. the program $P$ and the intended interpretation $\mathcal{I}$ with proof tree $T$ in *ICEC*, while $ICEC \not\models_{(P,\mathcal{I})} \mathcal{E}$ indicates that $\mathcal{E}$ cannot be proven in *ICEC*. The tree $T$, the program $P$, and the intended interpretation $\mathcal{I}$ are only made explicit when they are needed.

*CEC* is used by our tool in order to represent actual Erlang computations, while *ICEC* represents the knowledge that the tool obtains from the user. In practice only the minimum number of nodes of *ICEC* are obtained during the debugging session, since each one corresponds to a question. *ICEC* determines the validity of computations as indicates the following definition.

**Definition 5** Let $P$ be a Core Erlang Program, let $T$ be a *CEC* computation tree with respect to $P$, and let $N$ be a node in $T$ containing an evaluation $\mathcal{E}$.

1. $N$ is *valid* with respect to *ICEC* when $ICEC \models_{(P,\mathcal{I})} \mathcal{E}$, and *invalid* when $ICEC \not\models_{(P,\mathcal{I})} \mathcal{E}$.
2. $N$ is called *buggy* with respect to *ICEC* if $\mathcal{E}$ is invalid with all its children valid (in both cases w.r.t. *ICEC*).

The idea behind these definitions is that a buggy node represents an erroneous computation based on correct subcomputations. Thus, the piece of code associated to a buggy node represents an error in the program. This informal idea is formalized in the rest of the section.

The rôle of the two calculi is further clarified by the next assumption:

**Assumption 1** *Let $P$ be an Erlang program and $|\cdot|$ the transformation that converts an Erlang expression into a Core expression. Then:*

1. *An evaluation of the form $e\theta \to (mnf, l, \Pi)$ is computed by some Erlang system*[8] *with respect to $P$ iff $CEC \models_P \langle |e|, \theta\rangle \to (|mnf|, l, |\Pi|)$.*
2. *An evaluation of the form $e\theta \to (mnf, l, \Pi)$ computed by some Erlang system is considered unexpected with respect to ICEC by the user iff $ICEC \not\models_P \langle |e|, \theta\rangle \to (|mnf|, l, |\Pi|)$.*

*Analogous considerations are applied for $e\theta \to lock$, $e\theta \xrightarrow{l,i} (mnf, l, \Pi)$, and $\ll pid, \langle e, id\rangle, [] \gg \Rightarrow \Pi$.*

Now we are ready to define the errors detected by our tool.

### 6.3 Errors in Erlang Programs revisited

Next we define precisely the errors that can detected with our technique:

**Definition 6** Let $P$ be an Erlang program, $\mathcal{I}$ its expected interpretation, and $T$ a *CEC* computation tree. Let $r$ be a reference to a `receive` expression statement in $P$ and $f$ a reference to a program function. Then, we say that:

1. $r$ is *erroneously failing* if it is the `receive` statement referenced in a buggy node of $T$ rooted by the label ($\mathsf{LOCK}$).
2. $r$ is *erroneously succeeding* if it it is the `receive` statement referenced in a buggy node of $T$ rooted by the label ($\mathsf{RCV}_3$).
3. $r$ (respectively $f$) contains an *erroneous transition* if its body (in the case of $r$ the body of the branch reached accepting a previous message) is the conclusion of a buggy ($\mathsf{RCV}_1$) inference rule in $T$.
4. $f$ contains an *erroneous first step* if it is the function referenced in a buggy ($\mathsf{BFUN}$).

The errors are detected as discrepancies between the actual computations represented by *CEC* and the 'ideal' computations represented by *ICEC*.

Observe, for instance the definition of erroneously failing `receive`. Looking to the inference rule ($\mathsf{LOCK}$) and considering the definition of buggy node (invalid conclusion with valid premises) it says in other words that:

[8] In our context this sentence must be understood as "by executing $e$ with the values indicated by $\theta$ in Erlang we reach *mnf*, generating during the execution the messages in $l$ and creating the processes in $\Pi$."

- *fails*$(r, l, \theta)$, but
- *ICEC* $\nvDash_{(P,\mathcal{I})}$ $\mathcal{I}$*fails*$(r, l, \theta)$, that is, $\mathcal{I}$*fails*$(r, l, \theta) \notin \mathcal{I}$

We say that a Core Erlang program is a *wrong program* when it contains any of the errors mentioned in the previous definition. We also say that an Erlang program is a *wrong program* if its Core representation is wrong. The next proposition ensures that only the inference rules mentioned in the previous definition can be buggy:

**Proposition 1** *Let $P$ a Core Erlang program. Let $T$ a CEC computation tree with invalid root. Then:*

1. *$T$ contains at least one buggy node.*
2. *Every buggy node corresponds to either a* (BFUN). ($\mathsf{RCV}_1$), ($\mathsf{RCV}_3$), *or* (LOCK) *inference.*

The first item is a general property of computation trees which can be proved easily by induction on the size of the tree: in every tree with invalid root it is possible to find at least one invalid node with all its children valid.

The second item is a direct consequence of Definition 4 since only (BFUN), ($\mathsf{RCV}_1$), ($\mathsf{RCV}_3$), and (LOCK) are different in both calculus. In particular the *ICEC* version of the rules is more restrictive since they add new side conditions. Then it is straightforward to check that only inferences rooted by these rules can be buggy.

Now we are ready to state our main theoretical result:

**Theorem 1** *Let $P$ be an Erlang program, $\mathcal{I}$ its expected interpretation and $\epsilon$ an unexpected evaluation in $P$. Then:*

1. *$P$ is a wrong program.*
2. *A debugger examining the computation tree representing this computation in Core Erlang will find at least one of the errors described in Definition 6.*

**Proof sketch**

1. To prove the first item it is enough to check that the Core version of the program is a wrong program. According to Assumption 1 there is a computation tree $T$ such that

$$CEC \models_{P,T} \langle |\epsilon|, \theta\rangle \rightarrow (|mnf|, l, |\Pi|)$$

since $\langle \epsilon, \theta\rangle$ is evaluated by the program $P$. Moreover, the same assumption ensures that since the result obtained is unexpected we have

$$ICEC \nvDash_{P} \langle |\epsilon|, \theta\rangle \rightarrow (|mnf|, l, |\Pi|)$$

Then, applying the Definition 1 we have that the root of $T$, which is the node $\langle |\epsilon|, \theta\rangle \rightarrow (|mnf|, l, |\Pi|)$ is invalid. Then, by Proposition 1, $T$ contains a buggy node which corresponds to either a (BFUN). ($\mathsf{RCV}_1$), ($\mathsf{RCV}_3$), or (LOCK) inference rule, which in turn means that contains some of the errors described in Definition 6, and thus it is wrong program.
2. A top-down navigation strategy can find at least one buggy node, since the tree is finite.

Thus, our debugger based on the technique depicted in this paper is suitable for finding the errors in the Erlang programs with unexpected behavior.

## 7 Related work

One of the first papers that focus on the complexity of debugging concurrent programs is [32]. This seminal survey covers the problems related to debugging concurrent programs and presents some techniques that do not include declarative debugging. Apart from the mentioned paper, the debugging of concurrent programs has received much attention during the last two decades. There are papers devoted to the organization of debuggers [25], to the reproduction of bugs once they are discovered [33], or to the slicing of programs to obtain a program fragment that causes an anomalous behavior [28]. The declarative approach to debugging was extended to reactive systems in [30], concretely to Flat Concurrent Prolog (FCP). In the underlying concurrent model the processes are tuples containing the original goal, a list of input and output events that bind variables, and the final state. We could see these variable-binding events as message passing, but in this case the behavior is different from the Erlang model: binding a variable can affect several processes, whereas in Erlang messages are sent to exactly one process. In [24], the declarative debugging approach is applied to concurrent constraint programs. These programs are similar to logical ones, but have a store of constraints for the synchronization of processes. Before spawning a process the store may be forced to fulfill some preconditions (*ask* constraints), and the execution of a process may change the store (*tell* constraints). The semantics of a process is a set of pairs $(s, t)$ where $s$ is the initial store and $t$ the final store, and the questions of the debugger focus on these kind of transitions. As the store is shared by all the processes, it shows a similar behavior to binding events in FCP. The closest work to our paper is [35], that presents the application of a declarative debugging approach to a procedural language—concretely the language C, and integrated into GDB (GNU Project Debugger). Although the underlying model is procedural and not functional, it presents some similarities with

the Erlang model like the absence of shared memory and the use of send and receive expressions (less powerful than Erlang's). On the other hand, it does not support the creation of processes, having a fixed number of them. Furthermore, it does not use a formal semantics to build a tree, but a graph where the nodes are the send and receive events and the edges are obtained from a *happened before* relation [29].

The semantic calculus presented in this paper is inspired by other standard operational semantics for Erlang [26, 45], adapted to the syntax of Core Erlang [14]. They have in common that the semantic rules rewrite configurations of processes, which are tuples composed by a PID, an expression, and a mailbox. However, it presents two main differences: First, message passing is modeled by outboxes instead of inboxes. Thus, we describe more realistic Erlang computations, where messages from different processes may arrive in any order, solving the limitations of single-node semantics [43]. Second, we use *medium-sized normal forms*, which are expressions that cannot be further reduced because they are values or are `receive` expressions that need to read a message to continue. These normal forms are essential when debugging because they delimit the minimum transitions of the system whose validity is asked to the user.

## 8 Concluding Remarks and Ongoing Work

Debugging concurrent Erlang programs can be a puzzling task. A usual computation involves many processes that combine sequential portions of code with message passing and process creation. The usual step-by-step methodology employed in usual sequential trace debuggers is no longer valid, and the users must design their own strategy for tracing the source of the error employing any of the (powerful) tools available such as the Erlang trace debugger.

For this reason in this paper we present a semi-automatic methodology that guides the user in the difficult task of finding the error. The step is defined here as a 'jump' between two `receive` statements, focusing on the messages that allow the computation to progress. We think that this is a quite natural way of understanding a concurrent computation. Moreover, it has been claimed that message passing errors are the most difficult errors to find in terms of time [38].

The debugger has been built on top of a previous debugger for sequential Erlang programs [11]. Thus, our tool can debug both concurrent and sequential programs, adapting to the specifities of each computation.

From the point of view of the underlying logics, we have extended the semantic calculus for Core Erlang presented in [10] to deal with sets of processes (*configurations*), message passing instructions, and process creation (**`spawn`**). In this calculus, processes are accompanied by outboxes of sent messages not processed yet, overcoming the lack of expressiveness of some single-node semantics [43]. The proof trees are traversed interacting with the user through questions until a buggy node is found and the corresponding error is pointed out. The soundness and completeness of our debugging approach [9] has been established, and we have implemented a prototype of the declarative debugger that can be applied to average Erlang programs. Our setting even allows debugging non-terminating programs, a feature scarcely provided by declarative debuggers.

There are several lines of future work. Following the Erlang approach *"let it crash and let someone else deal with it,"* processes can be linked to provide robustness: when a process dies abnormally it sends the exit signal to all the processes linked to it. These linked processes can terminate abnormally or trap the signal to release some resources or restart the terminated process. A natural line of future work is extending the debugger to deal with process linkage and signals, which will require the extension of the calculus. Moreover, we are also interested in simplifying the questions asked to the user by taking into account the previous answers.

**Acknowledgements** Research supported by the Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731), by the MINECO Spanish projects *StrongSoft* (TIN2012-39391-C04-04), *VIVAC* (TIN2012-38137), *CAVI-(ROSE/ART)* (TIN2013-44742-C4-(1/3)-R), *LOBASS* (TIN2015-69175-C4-2-R), and *TRACES* (TIN2015-67522-C3-3-R), and by the European Union project POLCA (STREP FP7-ICT-2013.3.4 610686)

## References

1. P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, New York, NY, USA, 2014. ACM.
2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.
3. J. Armstrong. Concurrency oriented programming in erlang. *Invited talk, FFG*, 2003.
4. J. Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology, Sweden, December 2003.
5. J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.
6. T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

7. E. Arvaniti. Automated Random Model-Based Testing of Stateful Systems. Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, July 2011. http://artemis.cslab.ntua.gr/el_thesis/artemis.ntua.ece/DT2011-0142/DT2011-0142.pdf.
8. P. Blackburn, J. F. A. K. v. Benthem, and F. Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., New York, NY, USA, 2006.
9. R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. A declarative debugger for concurrent Erlang programs (extended version). Technical Report 15/13, Departamento de Sistemas Informáticos y Computación, December 2013. http://maude.sip.ucm.es/~adrian/pubs.html.
10. R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. A declarative debugger for sequential Erlang programs. In M. Veanes and L. Vigan, editors, *Proceedings of the 7th International Conference on Tests and Proofs, TAP 2013*, volume 7942 of *Lecture Notes in Computer Science*, pages 96–114. Springer, 2013.
11. R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. EDD: a declarative debugger for sequential Erlang programs. In E. Ábrahám and K. Havelund, editors, *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 581–586. Springer, 2014.
12. R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. A zoom-declarative debugger for sequential Erlang programs. *Science of Computer Programming*, 110:104–118, 2015.
13. R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. A zoom-declarative debugger for sequential Erlang programs. *Science of Computer Programming*, 110:104 – 118, 2015.
14. R. Carlsson. An introduction to Core Erlang. In *Proceedings of the Erlang Workshop 2001, in connection with PLI 2001*, pages 5–18, 2001.
15. R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. *Core Erlang 1.0.3 language specification*, November 2004. Available at http://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.
16. R. Carlsson and M. Rémond. EUnit: a lightweight unit testing framework for Erlang. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, pages 1–1. ACM, 2006.
17. M. Christakis, A. Gotovos, and K. Sagonas. 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, pages 154–163, Washington, DC, USA, 2013. IEEE Computer Society.
18. M. Christakis and K. Sagonas. Static detection of race conditions in Erlang. In M. Carro and R. Pena, editors, *Practical Aspects of Declarative Languages, PADL 2010*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2010.
19. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.
20. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
21. A. Farrugia and A. Francalanza. Towards a formalisation of erlang failure and failure detection (extended abstract). Technical report, University of Malta, 2012. WICT.
22. L.-Å. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, The Royal Institute of Technology, Sweden, August 2001.
23. L.-A. Fredlund and H. Svensson. Mcerlang: A model checker for a distributed functional programming language. *SIGPLAN Not.*, 42(9):125–136, Oct. 2007.
24. M. P. J. Fromherz. Towards declarative debugging of concurrent constraint programs. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, pages 88–100, London, UK, UK, 1993. Springer-Verlag.
25. J. Gait. A debugger for concurrent programs. *Software: Practice and Experience*, 15(6):539–554, 1985.
26. F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP 1999*, pages 261–272, New York, NY, USA, 1999. ACM.
27. D. Insa and J. Silva. An algorithmic debugger for Java. In M. Lanza and A. Marcus, editors, *Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM 2010*, pages 1–6. IEEE Computer Society, 2010.
28. J. Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, Sept. 2003.
29. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
30. Y. Lichtenstein and E. Shapiro. Concurrent algorithmic debugging. *SIGPLAN Not.*, 24(1):248–260, Nov. 1988.
31. T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In W.-N. Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems, APLAS 2004*, volume 3302 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2004.
32. C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989.
33. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280. USENIX Association, 2008.
34. H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.
35. T. Ohta, H. Kinoshita, T. Kimata, and T. Mizuno. A framework of an assertion-based algorithmic debugging for distributed programs. In *Proceedings of the The 15th International Conference on Information Networking*, ICOIN '01, pages 319–324, Washington, DC, USA, 2001. IEEE Computer Society.
36. M. Papadakis. Automatic Random Testing of Function Properties from Specifications. Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, October 2010. http://artemis.cslab.ntua.gr/el_thesis/artemis.ntua.ece/DT2010-0295/DT2010-0295.pdf.
37. M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*, pages 39–50. ACM Press, 2011.
38. J. B. Pedersen and M. Jones. Error classifications for parallel message passing programs: A case study. *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 387–394, 2012.
39. J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

40. P. Runeson. A survey of unit testing practices. *IEEE Softw.*, 23(4):22–29, July 2006.
41. K. Sagonas, J. Silva, and S. Tamarit. Precise explanation of success typing errors. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, PEPM 2013, pages 33–42, New York, NY, USA, 2013. ACM.
42. E. Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1983.
43. H. Svensson and L.-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, ERLANG 2007, pages 43–54, New York, NY, USA, 2007. ACM.
44. S. Tamarit, A. Riesco, E. Martin-Martin, and R. Caballero. Debugging Meets Testing in Erlang. In K. B. Aichernig and A. C. Furia, editors, *Proceedings of the 10th International Conference on Tests and Proofs (TAP 2016)*, volume 9762 of *Lecture Notes in Computer Science*, pages 171–180. Springer International Publishing, 2016.
45. G. Vidal. Towards Erlang verification by term rewriting. In G. Gupta and R. Peña, editors, *Proc. 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2013), Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 109–126. Springer International Publishing, 2014.