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 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, 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, 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, test-case generators, and debuggers play an important role.

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
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 [39]. 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 [12]. 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 [32], functional [34], and object-oriented [24] 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 [39], 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 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.

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 fulfills 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 acknowledge-ment value ($x+1$), and the new acknowledgment 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

Fig. 2 TCP Three-way handshake in Erlang

```erlang
main() ->
  Server_PID = spawn(?MODULE, server_fun, [self(), 50, 500]),
  spawn(?MODULE, client_fun, [Server_PID, 57, 100, client1]),
  spawn(?MODULE, client_fun, [Server_PID, 50, 200, client2]),
  receive
    {data,D} ->
      D
  end.

server_fun(Main_PID, Port, Seq) ->
  receive
    {Client_PID, {syn, Port, SeqCl}} ->
      Ack_PID = spawn(?MODULE, 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
      % server_fun(Main_PID, Port, Seq+1)
  end.

ack(Main_PID, Port, Ack, Seq, Client_PID) ->
  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.

client_fun(Server_PID, Port, Seq, Data) ->
  Server_PID ! {self(), {syn, Port, Seq}},
  syn_ack(Port, Data, Seq+1).

syn_ack(Port, Data, Ack) ->
  receive
    rst ->
      {port_rejected, Port} ;
    {Ack_PID, {syn_ack, Ack, Seq}} ->
      Ack_PID ! {self(), {Ack, Port, Seq+1, Ack}}, % swap Ack <-> ack
      Ack_PID ! {self(), {data, Port, Data}},
      {Seq+1, Ack, Port, Data}
  end.
```

```erlang`
which they must report: the server reports to the current process—\texttt{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 (\texttt{client1} and \texttt{client2}, respectively), but in general represents the data that each client will try to transmit. The \texttt{main} function is expected to return the first data obtained after a successful connection, which in this example should be \texttt{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 \texttt{ack} that will receive 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 \texttt{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 \texttt{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 \texttt{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 \texttt{ack} and \texttt{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 "\texttt{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 \texttt{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 \texttt{server_fun} function we insert the following two expressions:

```erlang
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 \texttt{main()} prints:

```erlang
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: client_fun(<0.40.0>,57,100,client1) in <0.41.0>
```

The most interesting message is the following:

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

The function \texttt{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 \texttt{ack}, since it is the place where the value \texttt{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 \texttt{START} and \texttt{END} print expressions and modify the \texttt{ack} function:

```erlang
... 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 messages is syn_ack, a quick scrutiny of its code reveals the origin of the bug in line 13.

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. 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, using the built-in Erlang debugger 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

---


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 come with their own debuggers integrated. Their usage is more comfortable since they 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. 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 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 \texttt{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: \texttt{rst} for a single rejected connection, \texttt{connect} for a single successful connection, \texttt{rst_twice} for two rejected connections, \texttt{connect_twice} for two successful connections, and \texttt{interleaved} for three interleaved connections (two rejected and one successful in the mid-
tcp_test_() ->
[?_assertEqual(client2,main()), rst(), connect(), rst_twice(), connect_twice(), interleaved()].

get_messages() ->
timer:sleep(500), % Wait 0.5s for the latest messages
receive M ->
[M|get_messages()]
after 0 ->
[]
end.

rst() ->
Server_PID = spawn(?MODULE, server_fun, [self()], 50, 500),
A = client_fun(Server_PID, 57, 100, client1),
Mbx = get_messages(),
[?_assertEqual(A,[port_rejected,57]), ?_assertEqual(Mbx,[])].

connect() ->
Server_PID = spawn(?MODULE, server_fun, [self()], 50, 500),
A = client_fun(Server_PID, 50, 100, client1),
Mbx = get_messages(),
[?_assertEqual(A,[501,101,50,client1]), ?_assertEqual(Mbx,[data,client1])].

rst_twice() ->
Server_PID = spawn(?MODULE, server_fun, [self()], 50, 500),
A = client_fun(Server_PID, 57, 100, client1),
B = client_fun(Server_PID, 60, 100, client2),
Mbx = get_messages(),
[?_assertEqual(A,[port_rejected,57]), ?_assertEqual(B,[port_rejected,60]), ?_assertEqual(Mbx,[])].

connect_twice() ->
Server_PID = spawn(?MODULE, server_fun, [self()], 50, 500),
A = client_fun(Server_PID, 50, 100, client1),
B = client_fun(Server_PID, 50, 300, client2),
Mbx = get_messages(),
[?_assertEqual(A,[501,101,50,client1]), ?_assertEqual(B,[502,301,50,client2]),
?_assertEqual(Mbx,[data,client1],[data,client2])].

interleaved() ->
Server_PID = spawn(?MODULE, server_fun, [self()], 50, 500),
A = client_fun(Server_PID, 33, 0, client1),
B = client_fun(Server_PID, 50, 100, client2),
C = client_fun(Server_PID, 51, 0, client3),
Mbx = get_messages(),
[?_assertEqual(A,[port_rejected,33]), ?_assertEqual(B,[502,101,50,client2]),
?_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 for-
warded 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 of the module tcp
and show a summary of the results. With the origi-
nal code of tcp, however, this invocation does not re-
turn any value since it does not terminate. In order to
detect the source of the bug, we must test each sce-
nario 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 makes 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 13 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):

```erlang
> 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:<<"">>```

```
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:

```erlang
> 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 [37,38] 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 6. 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 \( \{ \text{data},D \} \) message. Otherwise, the client must return \( \{ \text{port_rejected},CP \} \) 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 \( \text{FORALL} (V,G,E) \) 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:
the first 4 elements are integer numbers, whereas the last element of the tuple is any Erlang term. We use two integer generators: \texttt{integer(Min,Max)} to generate port numbers in a range, and \texttt{non_neg_integer()} to generate non-negative sequence numbers. For every 5-tuple generated, \texttt{is_correct_connection} (line 110) is invoked.

Once we have defined the property, we launch PropEr:

\begin{verbatim}
> 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
\end{verbatim}

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 \texttt{error_ack} message inside \texttt{Mbx}, a close inspection of the code would reveal the bug problem in line 44 when submitting \texttt{data} messages.

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

\begin{verbatim}
> proper:quickcheck(tcp:prop_one_connection()).

OK: Passed 100 test(s).
\end{verbatim}

The reason is that the \texttt{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 \texttt{?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 \textit{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 \[31\], 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:

\begin{verbatim}
$ dialyzer tcp.erl
Proceeding with analysis... done in 0m0.32s
\end{verbatim}
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.

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()"), 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. - <0.133.0>
   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:

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

Context:
- 'Client_PID' = <0.130.0>
- 'Main_PID' = <0.127.0>
- 'Port' = 50
- 'Seq' = 500

Received messages:
- {<0.130.0>, {syn, 57, 100}}
- {<0.131.0>, {syn, 50, 200}}
- {<0.131.0>, {syn, _, _}}

Consumed message:
- {<0.130.0>, {syn, 57, 100}}

2. - Evaluated to value: rst
3. - Sent messages:
   rst (from <0.129.0> to <0.130.0>)
4. - No created processes
5. - Nothing
6. - Anything

[1/2/3/4/5/6/7/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

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}}
   - {<0.131.0>, {syn, 50, 200}}
   - {<0.131.0>, {syn, _, _}}
3. - Consumed message:
   - {<0.130.0>, {syn, 57, 100}}
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

\footnote{EDD output has been slightly reshaped to fit in the page.}
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
     `{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 `<0.616.0>`)

Consumed message:

- `{<0.620.0>,(201,50,ack,502)}`
  (from `<0.620.0>` to `<0.616.0>`)

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 particular, 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

```
[1/2/3/4/5/6/1/t/d/c/s/p/r/u/h/a]: 2
'*Ack' = 201
'Ack_PID' = <0.621.0>
'Data' = client2
'Port' = 50
'Seq' = 501
```

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>)
```

The error has been detected:

```
The error has been detected:
The problem is in pid <0.620.0>
in tcp.erl:39
```

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' = <0.620.0>
   'Data' = <0.621.0>
   'Port' = 50
   'Seq' = 501

2. - Received messages:
   {<0.621.0>,{syn_ack,201,501}}
   (from <0.618.0> to <0.620.0>)
   {<0.620.0>,{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>,{502,201,50,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
```

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 4.2 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 5 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) 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-

\[\text{Note that (ii) can be wrong either because more messages were expected or because some of them are incorrect.}\]
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 [42x115]. A Core Erlang program is a set of function definitions. Each function is defined by exactly one rule, so Erlang functions with various clause constructions are translated into a single rule that starts with a \texttt{case} expression that performs the dispatching. Other differences with Erlang is that in Core Erlang the arguments of function calls and \texttt{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 \texttt{let} expressions. Figure 3 shows the translation of the function \texttt{ack} (lines \texttt{[21]}-\texttt{[22]} in Figure 2) omitting some irrelevant parts. This function generates a Core function that takes 5 variables as arguments: \texttt{cor4}, \texttt{cor3}, ... \texttt{cor0}[42]. The sequence of receive expressions is translated as a let expression—note that the variable \texttt{cor5} is not used in the body of the \texttt{let}. Finally, the Core program explicitly contains the implicit \texttt{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 \( \langle \text{pid}, \langle \text{expr}, \theta \rangle, l \rangle \), where:

- \( \text{pid} \) is the (unique) process identifier.
- \( \text{expr} \) is the expression being evaluated in the process. Occasionally the notation \( E[\text{expr}] \) is used to represent evaluation contexts, indicating that the expression \( \text{expr} \) is a subexpression of \( E \).
- \( \theta \) is a substitution mapping variables to values and standing for the context where \( \text{expr} \) 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 addressee yet. It has the form \( l \equiv [\text{pid}_1 ! \text{val}_1, \ldots, \text{pid}_m ! \text{val}_m] \), indicating the order of the messages, which is important in the case of multiple repetitions of the same process identifier \( \text{pid}_i \) 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 (\( \text{mnf} \)), which stands for expressions that cannot be further reduced by themselves. More specifically, an \( \text{mnf} \) can be either a value, represented by \( \text{val} \) in CEC, or a tuple \( \langle \text{expr}, \theta \rangle \) with \( \text{expr} \) 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 \( \text{expr} \). In Core Erlang, receive expressions can only be evaluated in let bindings, thus \( \text{expr} \) has the general form \( \text{expr} \equiv \text{let } x_1 = (\ldots (\text{let } x_n = \text{receive } \ldots \text{end in } e_n) \ldots) \text{ in } e_1 \). 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. \( (\text{expr}, \theta) \rightarrow (\text{mnf}, l, \Pi) \), with \( \text{expr} \) a non-receive expression, which indicates that \( \text{expr} \) has reached the medium-sized normal form \( \text{mnf} \), sending the messages stored in the outbox \( l \), and creating the new process computations in \( \Pi \).
3. \( (\text{expr}, \theta) \xrightarrow{1.0} (\text{mnf}, l', \Pi) \), with \( \text{expr} \) a receive expression, which indicates that \( \text{expr} \) is evaluated to the medium-sized normal form \( \text{mnf} \) by receiving the \( i \)th message from an outbox \( l \).

A special case of the first type inference is \( \Pi \Rightarrow \text{endlock} \), 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 \( (\text{expr}, \theta) \xrightarrow{1.0} \text{lock} \), 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 \( \text{mnf} \) without consuming any message, while the type three
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 \( \bot \).

**Definition 1 (Syntactic matching)**

\[
\begin{align*}
\text{match}(\text{var}, \text{val}) &= \{ \text{var} \mapsto \text{val} \} \\
\text{match}(\text{lit}_1, \text{lit}_2) &= \text{id}, \text{if } \text{lit}_1 &\equiv &\text{lit}_2 \\
\text{match}([\text{pat}_1] [\text{pat}_2], [\text{val}_1] [\text{val}_2]) &= \theta_1 \uplus \theta_2, \\
&\text{where } \theta_1 &\equiv &\text{match}(\text{pat}_1, \text{val}_1) \\
\text{match}([\text{pat}_1, \ldots, \text{pat}_n], [\text{val}_1, \ldots, \text{val}_n]) &= \theta_1 \uplus \ldots \uplus \theta_n, \\
&\text{where } \theta_i &\equiv &\text{match}(\text{pat}_i, \text{val}_i) \\
\text{match}(\text{var} = \text{pat}, \text{val}) &= \theta[\text{var} \mapsto \text{val}], \\
&\text{where } \theta &\equiv &\text{match}(\text{pat}, \text{val}) \\
\text{match}(\_\_\_\_) &= \bot, \text{if none of the previous rules apply.}
\end{align*}
\]

The operator \( \uplus \) stands for the union of two substitutions with disjoint domain. It verifies:

\[
\bot \uplus \theta = \theta = \bot \uplus \bot = \bot
\]

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 \( \text{fails}(\text{b}, \text{val}, \theta) \) holds, with \( \text{b} \) a receive branch of the form

\[
\text{pat when expr} \rightarrow \text{expr}'
\]

iff \( \theta' \equiv \text{match}(\text{pat} \theta, \text{val}) \) verifies either

1. \( \theta' = \bot \), or
2. \( \theta' \neq \bot \), \( \theta'' \equiv \theta \uplus \theta' \), and \( \| \text{expr} \theta'' \| \rightarrow \text{false} \)

The definition of \( \text{fails} \) 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:

\[
\begin{align*}
\text{receive } &\text{pat}_1 \text{ when } \text{expr}_1 \rightarrow r_1 \text{ expr}_1' \\
&\ldots \\
&\text{pat}_n \text{ when } \text{expr}_n \rightarrow r_n \text{ expr}_n'
\end{align*}
\]

\( l \) a list of messages, \( l \equiv [\text{val}_1, \ldots, \text{val}_m] \). Then we say that:

- \( \text{fails}(r, \text{val}, \theta) \) holds iff every branch \( b \) in \( r \),

\[
b \equiv \text{pat when expr} \rightarrow \text{expr}'
\]

in \( r \), verifies that \( \text{fails}(\text{val}, p.r, \theta) \)

- \( \text{fails}(r, l, \theta) \) holds iff \( \text{fails}(r, \text{val}_i, \theta) \) for every \( 1 \leq i \leq m \).

- \( \text{succeeds}(r, l, j, \theta) \rightarrow \text{expr}_j' \theta'' \) holds iff

\[
1. 1 \leq j \leq m, 1 \leq k \leq n. \\
2. \text{fails}(p.r, [\text{val}_1, \ldots, \text{val}_{j-1}], \theta). \\
3. \text{fails}(b_i, \text{val}_j, \theta) \text{ for every } i = 1 \ldots k - 1, \text{with } b_i \text{ the } i\text{th branch of the receive expression referenced by } r. \\
4. \text{Let } b_k \text{ be the } k\text{th branch of the receive statement referenced by } r, \\

\[
b_k \equiv \text{pat}_k \text{ when expr}_k \rightarrow \text{expr}'_k
\]

Then, \( \theta'' \equiv \text{match}(\text{pat} \theta, \text{val}_j) \) verifies

- \( \theta'' = \bot \),
- \( \theta'' \equiv \theta \uplus \theta' \), and
- \( \| \text{expr} \theta'' \| \rightarrow \text{true} \)
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 [22,23,21,45] consider that every process contains an inbox where incoming messages are stored. In these semantics, when an Erlang bang (!) instruction is executed to send a message \( m \) to a process \( \text{pid} \), \( m \) is immediately enqueued to the inbox of process \( \text{pid} \). 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 [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 \text{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 process will arrive first to \( P_2 \)? In single-node semantics the message hello will arrive always before \text{world}, as it will be inserted immediately into \( P_3 \) mailbox, but in a multi-node setting any of the two messages 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 (CONSUME) in Figure 9. In the example, when \( P_3 \) executes the receive statement it can fetch hello from the \( P_1 \) outbox or \text{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 \( \text{pid} \text{! message} \). In the example above, after sending the two messages, hello to \( P_2 \) and \text{world} to \( P_3 \), and assuming that no message has been consumed yet, the outbox of \( P_1 \) is the list \( l \equiv [P_2 \text{! hello}, P_3 \text{! world}] \). 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 \( lP \) 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 \( l|P_3 \equiv \text{[world]} \).
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 10 presents the rules for evolving processes, combining steps, and evaluating expressions after re-execution with exception propagation, can be found at [9].

![Fig. 10 Rules for processes II](image)

- **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.

- **CONSUME1** and **CONSUME2**. Configurations also evolve when a process that is currently evaluating a receive expression consumes a message from an outbox. The rule **CONSUME1** indicates that this message is taken from the outbox of another process, while **CONSUME2** 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.

- **RCV1** and **RCV2**. 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 **RCV1** 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, **RCV2** places the new lock in the `let` expression.

- **RCV3.** 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 `θ` as an extension of `θ` including the binding due to the matching. 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 **CONSUMEp** 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 `P1` sends message `hello` to `P2`, `world` to `P3`, and assuming that no message has been consumed yet is:

```plaintext
<< P1, mnf1, [P2 ! hello, P3 ! world] >>,
<< P2, mnf2, l2 >>,
<< P3, mnf3, l3 >>
```

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 1 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 5.  

- 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 (LET1) evaluates the inner expression in a \( \text{let} \) 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 (LET1), the rule (LET2) 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 \( | e | \) for the translation of the expression \( e \) to Core Erlang. It is extended to medium-sized normal forms as \( |val| = val \) and \(|(\text{expr},\theta)| = |(\text{expr}),\theta|\), and to configurations as \( |\langle \text{pid}_1,e_1,l_1 \rangle \ldots \langle \text{pid}_n,e_n,l_n \rangle| = |\langle \text{pid}_1|e_1|l_1 \rangle \ldots \langle \text{pid}_n|e_n|l_n \rangle|\).

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 \( \text{pid} \); \( h \) 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 \( \text{pid} \) received as argument:

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

The code above is translated into Core Erlang by following the ideas discussed in Section 6.1 Function \( f \) just applies function \( g \); function \( g \) requires two \text{let} expressions for evaluating the function \( \text{self}() \), evaluating \text{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 \text{let} expression for evaluating \( \text{self}() \) and sending the message using only variables and constants:
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 \( \text{self}(f) \) 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 \( \langle r_f, \theta_0 \rangle \to \text{mnf} \), where \( r_f \) is the reference to function \( f \). \( \text{mnf} \equiv \{ \text{receive}_{\theta_0}, \{ \_\text{cor}_0 \mapsto p_2, \_\text{cor}_1 \mapsto 4 \}, \theta_0 \equiv \{ \_\text{cor}_0 \mapsto 4 \} \), and \( \Pi \equiv \{ p_2, \{ r_1, \{ \_\text{cor}_0 \mapsto p_1 \} \} \} \). 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 (\( \cdot \cdot \cdot /1 \)) and the parameters (\( 4 \)). Since this inference is trivial in this case, we have substituted the corresponding trees by \( \nabla_0 \) and \( \nabla_1 \), 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 \( \text{let}_4 \) 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 (LET1) rule.
- The argument of the let expression is evaluated to a value (\( p_1 \), abbreviated in the subtree \( \nabla_3 \)) while its body is another let expression (identified as \( \text{let}_6 \)), since it is located at line 6. Hence, we have \( \theta_1 \equiv \theta_0 \cup \{ \_\text{cor}_1 \mapsto p_1 \} \) for evaluating \( \text{let}_6 \) with the rule (LET1).
- This inference is computed by first using (SPAWN) for generating a new process and then using (APPLY) with the substitution \( \theta_2 \equiv \theta_1 \cup \{ \text{PID} \mapsto p_2 \} \).
- Finally, using the substitution \( \theta_3 \equiv \{ \_\text{cor}_0 \mapsto \text{pid}_2, \_\text{cor}_1 \mapsto 4 \} \) 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 \textsc{(PROC)} rule evaluates the reference generated by \texttt{spawn} in the previous computation.
- As described for the previous tree, the \textsc{(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 \texttt{let} expression, we will use the rule \textsc{(LET1)}.
- The rule \textsc{(LET1)} evaluates \texttt{self()} (in \( \triangledown_4 \)) and then evaluates the bang expression with the substitution \( \theta_4 \equiv \{ \_\text{cor0} \mapsto p_1, \{ \_\text{cor1} \mapsto p_2 \} \} \).
- Finally, the message is sent just by evaluating the expressions to be sent, the atoms 3 and \( p_2 \), which is computed in \( \triangledown_5 \) and \( \triangledown_6 \), 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), \( I_0 \equiv \ll\ll_{p_1, mnf, [\emptyset]} \gg \ll p_2, \{3, p_2\}, p_1 ! \{3, p_2\} \gg \), and \( I_f \equiv \ll\ll_{p_1, T_1, [\emptyset]} \gg \ll p_2, \{3, p_2\}, p_1 ! \{3, p_2\} \gg \). In this tree we just use the transition 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 \texttt{(CONSUME1)}. Note that we use \texttt{(RCV3)}, where \( l \equiv [p_1 ! \{3, p_2\}] \), to consume the only message in the list. The tree \( \triangledown_7 \) is used to compute \texttt{succeeds}, while \( \triangledown_8 \) evaluates the final expression.

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
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:

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

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

$$\mathcal{I}_{\text{fun}} = \{\ldots, (f, \theta) \rightarrow (\text{mnf}, l, \Pi), \ldots\}$$

- $f$ is a reference to a function defined as $f/n = \text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{exprs}$.
- The domain of the substitution $\theta$ must be $\{\text{var}_1, \ldots, \text{var}_n\}$.

Thus, $\mathcal{I}_{\text{fun}}$ 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 for simplicity. The other set conforming $\mathcal{I}$ refers to the intended behavior of receive expressions and it is a set union of the form:

$$\mathcal{I}_{\text{rcv}} = \bigcup_{(p.r,l,\theta)} \mathcal{I}_{(p.r,l,\theta)}$$

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. $\mathcal{I}_{(p.r,l,\theta)}$ 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 $\mathcal{I}_{(p.r,l,\theta)}$:

1. $\mathcal{I}_{pr} = \{\mathcal{I}_{\text{fails}}(r, l, \theta)\}$ if the user expected the process to be blocked.

2. $\mathcal{I}_{pr} = \{\mathcal{I}_{\text{succeeds}}(p.r, l, j, \theta) \rightarrow (\text{mnf}, l', \Pi)\}$ if the computation of $p$ is expected to continue consuming the $j$th message of $l$, reaching the new medium-sized normal form $\text{mnf}$, 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 $[\text{val}_1, \ldots, \text{val}_m]$, while $l'$ is an output box list of the form $\text{val}_1, \ldots, \text{val}_m$.

A particular case of $\mathcal{I}_{pr}$ are the sets $\mathcal{I}_{rr}$ 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. $\text{(ZBFUN)}$, which adds the following additional side condition to (BFUN):

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

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

   $$\text{succeeds}_{Z}(p.r, l_1, j, \theta) \rightarrow R$$

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 concluding 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. $(\text{IRCV}_{3})$ adds to $(\text{RCV}_{3})$, the new side condition:

$$(\mathcal{I}\text{succeeds}(r, r, l, j, \theta) \rightarrow (\text{mnf}, l', II)) \in \mathcal{I}$$

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

The differences between ICEC and CEC are easy to understand. The first point forces $(\mathcal{I}_\text{BFUN})$ to check whether the first step of a function call is a valid statement in $\mathcal{I}$. Analogously, $(\mathcal{I}_\text{RCV}_1)$ and $(\mathcal{I}_\text{RCV}_3)$ only allow expected transitions. In the rule $(\mathcal{I}_\text{RCV}_1)$ 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, $(\text{ITOCK})$ 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

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

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 $\text{ICEC} \not\models_{(P, \mathcal{I}, T)} \mathcal{E}$ 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 $\text{ICEC} \models_{(P, \mathcal{I}, T)} \mathcal{E}$, and invalid when $\text{ICEC} \not\models_{(P, \mathcal{I}, T)} \mathcal{E}$.
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 incorrect 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 role 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 $e\theta \rightarrow (\text{mnf}, l, II)$ is computed by some Erlang system with respect to $P$ iff $\text{CEC} \models_P (|e|, \theta) \rightarrow (|\text{mnf}|, l, |II|)$.
2. An evaluation of the form $e\theta \rightarrow (\text{mnf}, l, II)$ computed by some Erlang system is considered unexpected with respect to ICEC by the user iff $\text{ICEC} \not\models_P (|e|, \theta) \rightarrow (|\text{mnf}|, l, |II|)$.

Analogous considerations are applied for $e\theta \rightarrow \text{lock}$, $e\theta \rightarrow (\text{mnf}, l, II)$, and $\langle \text{pid}, (e, id), [] \rangle \rightarrow II$.

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 (LOCK).
2. $r$ is erroneously succeeding if it is the receive statement referenced in a buggy node of $T$ rooted by the label (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 (RCV$_1$) inference rule in $T$.
4. $f$ contains an erroneous first step if it is the function referenced in a buggy (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 (LOCK) and considering the definition of buggy node (invalid conclusion with valid premises) it says in other words that:

*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 $II$.”*
We say that a Core Erlang program is a \textit{wrong program} when it contains any of the errors mentioned in the previous definition. We also say that an Erlang program is a \textit{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:

\textbf{Proposition 1} \ Let \( P \) a Core Erlang program. Let \( T \) a CEC computation tree with invalid root. Then:

\begin{enumerate}
  \item \( T \) contains at least one buggy node.
  \item Every buggy node corresponds to either a \textit{(BFUN)} \((RCV_1), (RCV_3), \) or \textit{(LOCK)} inference.
\end{enumerate}

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 \textit{(BFUN)}, \textit{(RCV_1)}, \textit{(RCV_3)}, and \textit{(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:

\textbf{Theorem 1} \ Let \( P \) be an Erlang program, \( \mathcal{I} \) its expected interpretation and \( e \) an unexpected evaluation in \( P \). Then:

\begin{enumerate}
  \item \( P \) is a wrong program.
  \item A debugger examining the computation tree representing this computation in Core Erlang will find at least one of the errors described in Definition 7
\end{enumerate}

\textbf{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 \( \mathcal{I} \) there is a computation tree \( T \) such that

\[ \mathsf{CEC} \models_{\mathcal{I}} (e, \theta) \rightarrow (|\mathsf{mnf}|, l, |\mathcal{I}|) \]

since \( (e, \theta) \) is evaluated by the program \( P \). Moreover, the same assumption ensures that since the result obtained is unexpected we have

\[ \mathsf{ICEC} \not\models_P (e, \theta) \rightarrow (|\mathsf{mnf}|, l, |\mathcal{I}|) \]

Then, applying the Definition \( \mathcal{I} \) we have that the root of \( T \), which is the node \( (|e|, \theta) \rightarrow (|\mathsf{mnf}|, l, |\mathcal{I}|) \)

is invalid. Then, by Proposition 4 \( T \) contains a buggy node which corresponds to either a \textit{(BFUN)}, \textit{(RCV_1)}, \textit{(RCV_3)}, or \textit{(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 \textbf{Related work}

One of the first papers that focus on the complexity of debugging concurrent programs is \cite{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 \cite{25}, to the reproduction of bugs once they are discovered \cite{33}, or to the slicing of programs to obtain a program fragment that causes an anomalous behavior \cite{28}. The declarative approach to debugging was extended to reactive systems in \cite{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 \cite{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 (\textit{ask} constraints), and the execution of a process may change the store (\textit{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 \cite{38}, 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 specificities 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 [10] 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 (TIN2012-39137), 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

http://artemis.cslab.ntua.gr/el_thesis/ 


http://maude.sip.ucm.es/~adrian/pub.html


