Test-Driven Development of Concurrent Programs using Concuerror

Alkis Gotovos [1] Maria Christakis [1] Konstantinos Sagonas [1,2]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
{alkisg,mchrista,kostis}@softlab.ntua.gr kostis@it.uu.se

Abstract

This paper advocates the test-driven development of concurrent Erlang programs in order to detect early and eliminate the vast majority of concurrency-related errors that may occur in their execution. To facilitate this task we have developed a tool, called Concuerror, that exhaustively explores process interleaving (possibly up to some preemption bound) and presents detailed interleaving information of any errors that occur. We describe in detail the use of Concuerror on a non-trivial concurrent Erlang program that we develop step by step in a test-driven fashion.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Reliability, Verification

Keywords test-driven development, Erlang

1. Introduction

Erlang is currently gaining popularity mainly due to its built-in support for concurrency. This support, based on asynchronous message passing between processes that in principle do not share any memory, not only differs from the concurrency support of most other languages, but also greatly simplifies the development of certain concurrent programs and makes it significantly less error-prone. Still, concurrent programming is fundamentally harder than its sequential counterpart. Its difficulty in avoiding errors and reasoning about program correctness mainly lies in the non-determinism that is introduced by unanticipated process interleaving. Often, such interleaving results in subtle concurrency errors which are difficult to reproduce during testing. Erlang programs are not immune to such problems. Tools that help programmers detect these errors early in the development process are needed in Erlang just as much as in any language.

To ease the systematic test-driven development of concurrent Erlang programs, we have developed Concuerror (pronounced /ˈkɒŋkərər/, like conqueror), a totally automatic tool that given a set of tests and a program detects most kinds of concurrency-related runtime errors, namely process crashes, assertion violations and deadlocks, that might occur during the execution of these tests.[1]

The temptation to describe in detail the techniques used by Concuerror is very strong, but we will try to resist it; a forthcoming companion paper will deal with this subject. Instead, the purpose of the current paper is to advocate test-driven development of concurrent Erlang programs using small-step iterations and unit tests, while describing in detail how Concuerror can be used in practice as a testing and debugging aid. For this purpose, we will write and test in a loosely test-driven fashion a simple example program that is a typical use case of concurrent Erlang. We warn our readers that some of the concurrency errors that Concuerror will detect may surprise even seasoned Erlang programmers.

The next section briefly overviews the suitability, or lack thereof, of existing tools for test-driven development in Erlang. Section 3 overviews Concuerror's functionality and characteristics, followed by Section 4 that presents in detail the test-driven development of an example program with the iterative use of Concuerror. The paper ends with some concluding remarks.

2. Test-Driven Development in Erlang

Test-driven development [2] is a software development practice that suggests using a workflow of small three-step iterations, each of them consisting in writing a small failing test, quickly making it work, and refactoring the program to remove any code duplication. Although there exist several testing tools for Erlang, the majority of them fails to be effective when used for test-driven development in Erlang's concurrent setting.

EUnit [3] is the Erlang implementation of the popular xUnit testing framework and is reportedly the most used testing tool by Erlang developers [8]. Despite its ease of use, EUnit executes each test under a single interleaving and is inadequate for detecting concurrency errors.

Erlang'11, September 23, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0859-5/11/09. . . $10.00

[1] Concuerror's code, its manual and further examples can be found at `https://github.com/mariachris/Concuerror`.

Quviq's QuickCheck [1] is a property-based fuzz testing tool that has been extended with the PULSE scheduler [4] for randomly interleaving processes to detect concurrency errors. Besides the random nature of its testing procedure, that provides no correctness guarantees, the user is required to write properties the program should satisfy using a special notation. This implies the user's familiarity with the non-trivial task of writing properties and, additionally, excludes the use of existing unit tests.

McErlang [5, 6] is a model checker that utilizes a custom runtime simulator of Erlang's concurrency semantics to explore program state-space. The ability to deploy monitors containing linear temporal logic formulas makes McErlang very powerful in terms of verification capability. Nevertheless, McErlang in its default mode of use employs a very coarse-grained process interleaving semantics. The detection of subtle concurrency errors, similar to those found in Section 4, might require the manual insertion of unobvious code, a task that is tedious, alters the original test, and does not concord with the philosophy of test-driven development.

3. A Glimpse of Concuerror

As mentioned, Concuerror detects concurrency errors that may occur when an Erlang function (usually a test) is executed. Existing tests can be readily used in Concuerror without requiring any modifications of the program under test. By systematically interleaving the participating processes, Concuerror effectively explores the program's state space and detects interleaving sequences with unintended effects, namely abnormal process exits due to an exception, assertion violations and deadlocks. In Concuerror, any program state where one or more processes are blocked on a `receive` statement and no other process is available for scheduling is considered a deadlock.

Under the hood, Concuerror uses a stateless searching approach to produce interleaving sequences. *Preemption points*, i.e. points in the program where a context-switch is allowed to happen, are only placed at side-effecting operations, like sends, receives, process exits, etc. Interleaving redundancy is further reduced by avoiding sequences that involve process blocks on `receive` statements. To make the tool even faster, the user can opt for a heuristic prioritization technique called *preemption bounding* [7], that limits the number of produced interleaving sequences according to a user specified parameter.

Concuerror's operation can be summarized as follows: The user opens the tool's graphical user interface (GUI), imports a number of Erlang modules and chooses a test to run. As a first step, Concuerror's *instrumenter* applies an automatic parse transformation to the imported modules in order to insert preemption points into the code. After the transformed modules have been compiled and loaded, the tool's *scheduler* executes all possible interleaving sequences of the selected test—up to the chosen preemption bound—and reports any errors encountered. If the user chooses to replay an error, the scheduler executes the corresponding interleaving sequence and records detailed information about the processes' actions that is then displayed in the GUI. Using this information, the user can iteratively apply code changes and replay the erroneous interleaving sequence to observe how program execution is affected. If no errors are reported, a higher preemption bound may be selected for a more thorough exploration, depending on the program's complexity and the user's time constraints. If the test completes without detecting any errors and the preemption bound has not been reached, the program is indeed free from the kinds of concurrency errors detected by the tool.

Having briefly discussed the characteristics of our tool, let us now see how Concuerror can be used in practice to detect subtle concurrency errors.

4. An Extended Example

4.1 Getting started

We intend to use Concuerror in a test-driven development process that involves the creation of a generic registration server, that can, for example, be used to manage limited system resources. The desired functionality includes starting and stopping the server as well as attaching and detaching processes to it. However, only a limited number of processes are allowed to be attached to the server at any moment.

First, we create two Erlang modules, one to contain the server code (`reg_server`), the other to contain the tests (`reg_server_tests`). Let us start by writing a `start/0` function for starting the server and a trivial test that checks that the function returns `ok`.

As some EUnit assertion macros can be readily used in Concuerror tests, we will use one of them, namely `?assertEqual`, to check the return value of `start/0`. This macro tests two expressions for equality with the convention that the first argument declares the expected value of the second argument. The `reg_server_tests` module, shown in Test 1, uses an `include_lib` attribute to make this macro available.

Test Code 1 The initial testing module

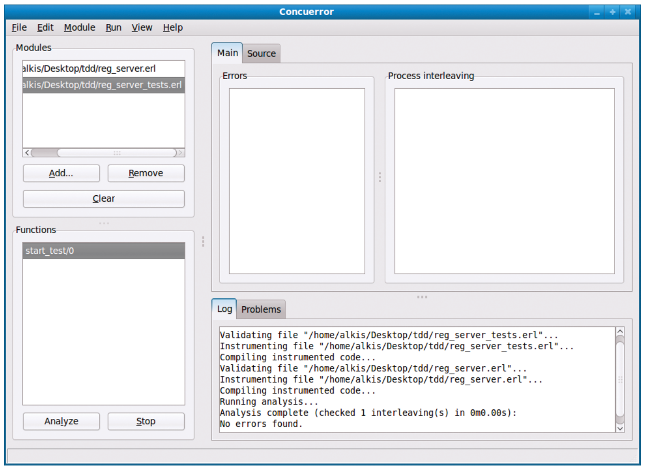
```
-module(reg_server_tests).

-include_lib("eunit/include/eunit.hrl").

start_test() ->
  ?assertEqual(ok, reg_server:start()).
```

Our first test uses `?assertEqual` to check that `start/0` returns `ok`. Note that functions with a "`_test`" suffix in their name are auto-exported by EUnit, thus no `export` attribute is needed. The first version of the server module is shown in Program 1. The module exports the `start/0` function, that just returns `ok`.

Program 1 The initial registration server module

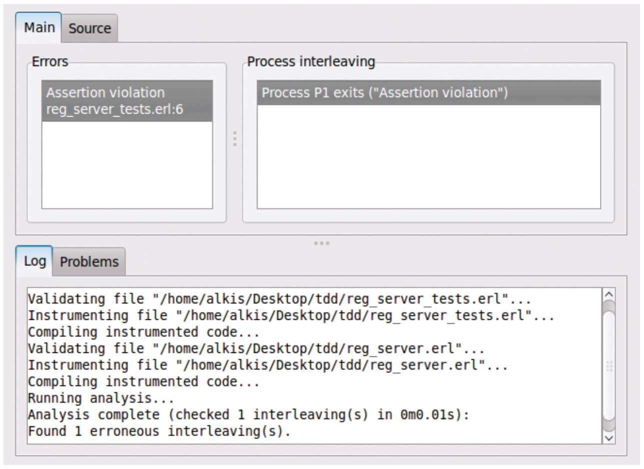
```
-module(reg_server).

-export([start/0]).

start() ->
  ok.
```

It is time to run our test using Concuerror. The Concuerror GUI is composed of several panels; see Figure 1. The Modules panel on the upper left side displays a list of



```
Validating file "/home/alkis/Desktop/tdd/reg_server_tests.erl"...
Instrumenting file "/home/alkis/Desktop/tdd/reg_server_tests.erl"...
Compiling instrumented code...
Validating file "/home/alkis/Desktop/tdd/reg_server.erl"...
Instrumenting file "/home/alkis/Desktop/tdd/reg_server.erl"...
Compiling instrumented code...
Running analysis...
Analysis complete (checked 1 interleaving(s) in 0m0.00s):
No errors found.
```

Figure 1. Running a test that passes in Concuerror's GUI



Assertion violation
reg_server_tests.erl:6

Process P1 exits ("Assertion violation")

```
Validating file "/home/alkis/Desktop/tdd/reg_server_tests.erl"...
Instrumenting file "/home/alkis/Desktop/tdd/reg_server_tests.erl"...
Compiling instrumented code...
Validating file "/home/alkis/Desktop/tdd/reg_server.erl"...
Instrumenting file "/home/alkis/Desktop/tdd/reg_server.erl"...
Compiling instrumented code...
Running analysis...
Analysis complete (checked 1 interleaving(s) in 0m0.01s):
Found 1 erroneous interleaving(s).
```

Figure 2. Information about an erroneous interleaving

all Erlang modules imported to Concuerror. These modules will be instrumented when Concuerror begins its analysis. The Functions panel on the lower left side displays a list of all exported functions defined in the module selected from the Modules panel. The Log panel on the lower right side displays information about Concuerror actions. Additionally, by selecting the Source tab next to Main, we can see the source code of the selected module.

Let us run our test before explaining the rest of the GUI functionality. To import our modules, we either click the Add button or select File→Add. Through the browse dialog, we locate our two modules in the file system, select them, and click Open. Our modules have now been imported and added to the Modules panel. By selecting module `reg_server_tests` from this panel, testing function `start_test/0` appears in the Functions panel. For the time being, we will disable preemption bounding by unchecking the Enable preemption bounding option under Edit→Preferences. We can now select the test function and click Analyze to execute the test under Concuerror. The Log panel informs us about the successful instrumentation and compilation of our modules as well as about the complete execution of one interleaving sequence without any errors (see Figure 1).

Let us now change the return value of `start/0` from `ok` to `error` and run the analysis again. This time an error appears in the Errors panel, namely an assertion violation on line 6 of our testing module. Furthermore, in the Process interleaving panel we can see the erroneous interleaving sequence. It consists solely of our testing process' abnormal exit (see Figure 2). We can also select the Problems tab next to Log to switch to the panel that displays additional information about the specific error: The expected value was `ok` but the call to `start/0` returned `error`. We change the return value back to `ok` before proceeding.

These are the basics about importing modules and running tests using the Concuerror GUI. In the next section we begin implementing the server's functionality.

4.2 Starting and stopping the server: The basics

We will create a server process and register it under the name `reg_server`. To test that the server is spawned and registered as expected, we will create an auxiliary `ping/0` function that returns `pong` if the server responds. In case the server is down, the ping function should probably timeout after a while, but we leave this task for later. We also leave for later the case of `start/0` being called more than once—by either one or more processes. At this point, we want to spawn the server process, register it and make it respond to a ping call. To this end we create `ping_test/0`, shown in Test 2. We call `ping/0` twice to make sure that the server loop is correct.

Test Code 2 Add a ping test

```
-module(reg_server_tests).

-include_lib("eunit/include/eunit.hrl").

start_test() ->
  ?assertEqual(ok, reg_server:start()).

ping_test() ->
  reg_server:start(),
  ?assertEqual(pong, reg_server:ping()),
  ?assertEqual(pong, reg_server:ping()).
```

To make the test pass, we spawn a new process to start the server, register it under the name `reg_server`, and make it execute function `loop/0`, as shown in Program 2. Inside its loop, the server receives ping requests and replies with pong messages. Function `ping/0` sends a ping request to the server and waits for a pong response. The macros `REG_REQUEST` and `REG_REPLY` are used to avoid confusion with messages sent by other processes.

Running our tests in Concuerror results in both of them reporting a deadlock. We should be expecting this, because the server runs in a loop and is blocked waiting for a request, even after the client process, i.e. the process running the test function, has exited. Note that Concuerror reports a "deadlock", because the only process alive is blocked on a `receive` statement. To avoid the aforementioned behavior, we have to implement the functionality of stopping the server and use it at the end of our tests. Again, we leave the handling of multiple stop calls for later.

Program 2 Add spawn, register and ping to the server

```
-module(reg_server).

-export([ping/0, start/0]).

-define(REG_NAME, reg_server).
-define(REG_REQUEST, reg_request).
-define(REG_REPLY, reg_reply).

start() ->
  Pid = spawn(fun() -> loop() end),
  register(?REG_NAME, Pid),
  ok.

ping() ->
  ?REG_NAME ! {?REG_REQUEST, self(), ping},
  receive
    {?REG_REPLY, Reply} -> Reply
  end.

loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      Target ! {?REG_REPLY, pong},
      loop()
  end.
```

Program 3 Add `stop/0` to the server

```
-module(reg_server).

-export([ping/0, start/0, stop/0]).

...

stop() ->
  ?REG_NAME ! {?REG_REQUEST, self(), stop},
  receive
    {?REG_REPLY, Reply} -> Reply
  end.

loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      Target ! {?REG_REPLY, pong},
      loop();
    {?REG_REQUEST, Target, stop} ->
      Target ! {?REG_REPLY, ok}
  end.
```

Function `stop/0` will be used to stop the server as shown in the modified tests of Test 3; its implementation is shown in Program 3. Similarly to `ping/0`, a message is sent to the server and the latter replies with `ok`. However, in this case, the server terminates instead of looping again.

Now both tests pass when run in Concuerror. Note that even these small tests produce more than one interleaving sequence—three in this case—due to the different exit orders of the two testing processes. Before we continue, let us refactor the code of Program 3 to remove some of the duplication that was introduced during our last step. In Program 4, we define functions `request/1` and `reply/2` to handle the sending and receiving of messages between client and server. The tests still pass after the refactoring,

Test Code 3 Stop the server at the end of our tests

```
start_stop_test() ->
  ?assertEqual(ok, reg_server:start()),
  ?assertEqual(ok, reg_server:stop()).

ping_test() ->
  reg_server:start(),
  ?assertEqual(pong, reg_server:ping()),
  ?assertEqual(pong, reg_server:ping()),
  reg_server:stop().
```

Program 4 The refactored code of Program 3

```
stop() ->
  request(stop).

ping() ->
  request(ping).

loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      reply(Target, pong),
      loop();
    {?REG_REQUEST, Target, stop} ->
      reply(Target, ok)
  end.

request(Request) ->
  ?REG_NAME ! {?REG_REQUEST, self(), Request},
  receive
    {?REG_REPLY, Reply} -> Reply
  end.

reply(Target, Reply) ->
  Target ! {?REG_REPLY, Reply}.
```

Test Code 4 Test for two stop calls by one process

```
multiple_stops_test() ->
  reg_server:start(),
  ?assertEqual(ok, reg_server:stop()),
  ?assertEqual(server_down, reg_server:stop()).
```

so at this point we are able to perform the basic operations of starting, pinging and stopping our server.

4.3 Starting and stopping the server: Advanced issues

Let us now handle the case of multiple stop calls. We will first test the case of two consecutive stop calls by one process and then that of two concurrent stop calls by two processes. As shown in Test 4, when a process calls `stop/0` and the server is not running, the return value should be `server_down`. Running this test in Concuerror results in three erroneous interleaving sequences, two due to an exception and one due to a deadlock. Looking at the Process interleaving panel, we notice that the first exception happens when the client process attempts to send a message to the already exited server process using its registered name. To fix this error, we can use the `whereis/1` built-in to check whether the server name is registered, that is, whether the server is running, before sending the stop

Program 5 Use `whereis/1` before sending a message to the server

```
request(Request) ->
  case whereis(?REG_NAME) of
    undefined -> server_down;
    _Pid ->
      ?REG_NAME ! {?REG_REQUEST, self(), Request},
      receive
        {?REG_REPLY, Reply} -> Reply
      end
  end.
```

message. In fact, we can do this inside `request/1`, so that the check is performed before any message is sent to the server, as shown in Program 5.

Running the test again, we see that the first error is fixed but the other two are still there. Now is a good point to try running the tests using EUnit rather than Concuerror. This can be done by first compiling the modules and then calling `eunit:test(reg_server_tests)`. EUnit reports that all three tests pass. We can try it again and again; the result is the same. The reason is that the errors displayed in Concuerror are caused by interleaving sequences that are very unlikely to happen in practice. The interleaving scenarios that are responsible for these errors are almost impossible to reveal with traditional testing tools, like EUnit; yet, it is still possible that they actually occur.

Back to Concuerror and in the Process interleaving panel, we can see that both errors have the same cause: Between the server's reply and its actual exit, the client process manages to call `whereis/1`—it even manages to send its second stop request in one of the two interleaving sequences. Because the server has not exited yet, the call to `whereis/1` does not return `undefined`. Subsequently, the server exits and, as a result, the client either fails with an exception, in case it tries to send a stop request to a process that is not registered anymore, or blocks, in case it has already sent the request and is waiting for an answer from the non-existing server. The detailed sequence for the second case, as viewed in the Concuerror GUI, is shown in Figure 3.

Before fixing the problem, let us run our last test in McErlang using its default configuration. Like EUnit, McErlang reports no errors. The reason is that McErlang in its default mode of use only preempts processes at `receive` expressions, although we have just seen that additional preemption points are needed to reveal the particular defect. There are three ways to detect the problem in McErlang.

The first is to manually insert preemption points using the `mce_erl:pause/1` call provided by the tool. In Program 6 we have used Concuerror's details about the problem to insert just enough preemption points for McErlang to also detect it. Manually inserting preemption points is clearly a tedious task, even when their locations are priorly known, which is typically not the case when ignorant about the possibility of concurrency errors at particular program points. Moreover, the need for alterations in the original program makes the tool less usable and code maintenance more difficult.

Program 6 Manual insertion of preemption points to detect the problem in McErlang

```
request(Request) ->
  Whereis = whereis(?REG_NAME),
  mce_erl:pause(
    fun() ->
      case Whereis of
        undefined -> server_down;
        _Pid ->
          ?REG_NAME ! {?REG_REQUEST, self(), Request},
          receive
            {?REG_REPLY, Reply} -> Reply
          end
      end
    end).

reply(Target, Reply) ->
    Target ! ?REG_REPLY, Reply,
    mce_erl:pause(fun() -> ok end).
```

The second way also involves modifying the program so that every process is spawned on a separate node. This approach not only alters the existing code, but also changes its intended semantics. In theory, McErlang also provides an experimental option for executing the test under a simulated distributed semantics without the need for code modifications. However, in practice, enabling this option does not reveal the defect.

The third way is to activate McErlang's `sends_are_sefs` option, which causes some actions, like sends, links and monitors, to be considered side-effects, thus enabling a finer-grained interleaving semantics. This method is by far the most convenient, yet it is still experimental according to McErlang's user manual at the time of this writing.

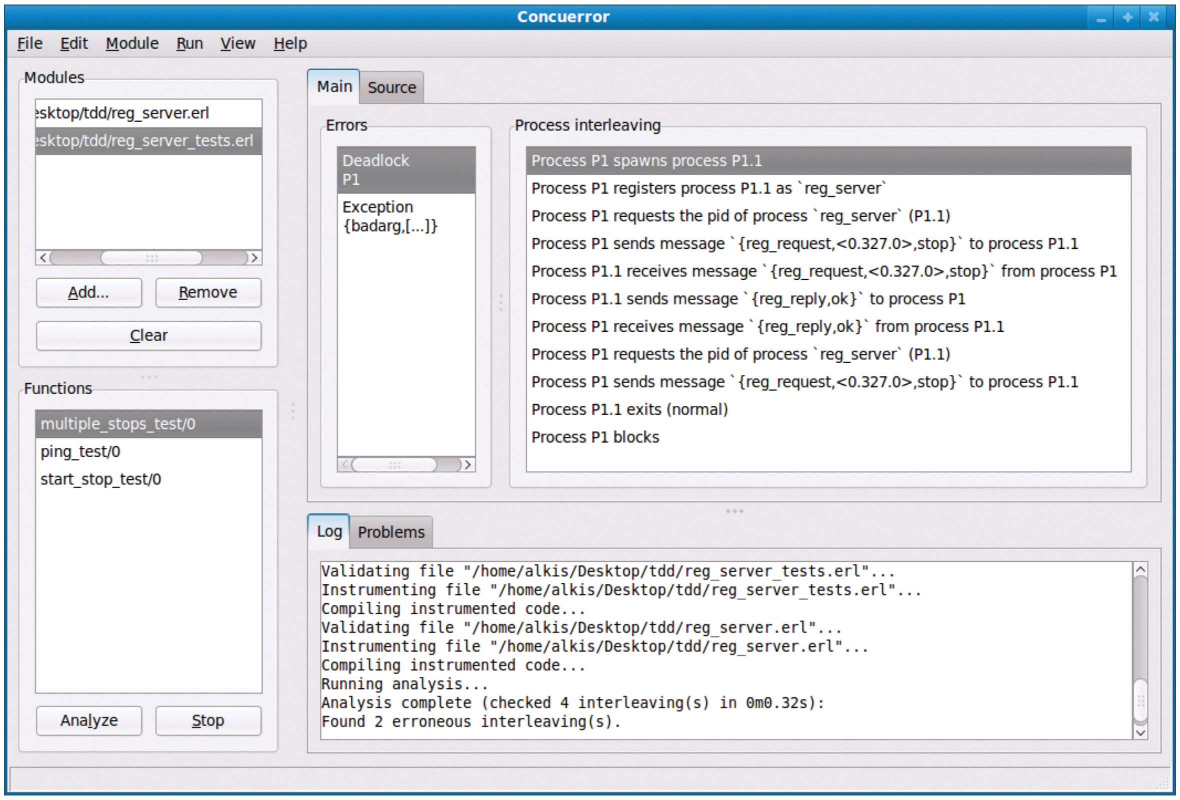
Returning to the problem of the client calling `whereis/1` between the server's reply and exit, a possible fix is to have the server unregister itself before replying to a stop request. The corrected code, that makes all three of our tests pass in Concuerror, is shown in Program 7.

Program 7 Unregister the server before replying to a stop request

```
loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      reply(Target, pong),
      loop();
    {?REG_REQUEST, Target, stop} ->
      unregister(?REG_NAME),
      reply(Target, ok)
  end.
```

Next, we have to test the case where multiple processes attempt to stop the server concurrently. We will test this scenario by spawning two processes, making them stop the server, and collecting the return values of their `stop/0` calls. We expect that one call will return `ok` and the other `server_down`, as shown in Test 5.

Running this test in Concuerror results in a large number of erroneous interleaving sequences. Looking at the first of them, we notice that both processes call `whereis/1` before the server stops. The problem is similar to the one we had in our previous step. If the server has exited before



```
Process P1 spawns process P1.1
Process P1 registers process P1.1 as `reg_server`
Process P1 requests the pid of process `reg_server` (P1.1)
Process P1 sends message `{reg_request,<0.327.0>,stop}` to process P1.1
Process P1.1 receives message `{reg_request,<0.327.0>,stop}` from process P1
Process P1.1 sends message `{reg_reply,ok}` to process P1
Process P1 receives message `{reg_reply,ok}` from process P1.1
Process P1 requests the pid of process `reg_server` (P1.1)
Process P1 sends message `{reg_request,<0.327.0>,stop}` to process P1.1
Process P1.1 exits (normal)
Process P1 blocks
```

```
Validating file "/home/alkis/Desktop/tdd/reg_server_tests.erl"...
Instrumenting file "/home/alkis/Desktop/tdd/reg_server_tests.erl"...
Compiling instrumented code...
Validating file "/home/alkis/Desktop/tdd/reg_server.erl"...
Instrumenting file "/home/alkis/Desktop/tdd/reg_server.erl"...
Compiling instrumented code...
Running analysis...
Analysis complete (checked 4 interleaving(s) in 0m0.32s):
Found 2 erroneous interleaving(s).
```

Figure 3. Concuerror's detailed interleaving information helps understand and fix the error

Test Code 5 Test for two concurrent stop calls by two processes

```
multiple_concurrent_stops_test() ->
  Self = self(),
  reg_server:start(),
  spawn(fun() -> Self ! reg_server:stop() end),
  spawn(fun() -> Self ! reg_server:stop() end),
  ?assertEqual(lists:sort([ok, server_down]),
               lists:sort(receive_two())).

receive_two() ->
  receive
    Result1 ->
      receive
        Result2 -> [Result1, Result2]
      end
  end.
```

Program 8 Monitor the server to deal with multiple concurrent stop calls

```
request(Request) ->
  case whereis(?REG_NAME) of
    undefined -> server_down;
    Pid ->
      Ref = monitor(process, Pid),
      Pid ! {?REG_REQUEST, self(), Request},
      receive
        {?REG_REPLY, Reply} ->
          demonitor(Ref, [flush]),
          Reply;
        {'DOWN', Ref, process, Pid, _Reason} ->
          server_down
      end
  end.
```

the process sends its request, the requesting process will fail with an exception. Alternatively, if the server exits after the request has been sent, the requesting process will block waiting for a server reply that will never come.

We have to provide a way for a process to be informed of whether the server has been stopped by another process after the execution of its `whereis/1` call. To avoid the exceptions, instead of the server's registered name, we can use its process identifier (pid) returned by `whereis/1` in `request/1`. Doing this will result in the transformation of the exception errors into deadlocks due to processes blocking on the `receive` expression of `request/1`. We avoid these deadlocks by monitoring the server process through a `monitor/2` call just before the request message is sent. If the server has already exited or exits after the monitoring call, the client process will receive a `'DOWN'` message from the Erlang runtime and can return `server_down`. Otherwise, the process will receive a normal reply from the server. In any case, a message will be received and, therefore, the process will never block. The changes made in the server code are shown in Program 8. A call to `demonitor/2` with the `flush` option is used to stop monitoring the server and discard the `'DOWN'` message in case the server has already exited.

Test Code 6 Test calling `ping/0` when the server is not running

```
ping_failure_test() ->
  ?assertEqual(server_down, reg_server:ping()),
  reg_server:start(),
  reg_server:stop(),
  ?assertEqual(server_down, reg_server:ping()).

ping_concurrent_failure_test() ->
  reg_server:start(),
  spawn(fun() ->
          R = reg_server:ping(),
          Results = [pong, server_down],
          ?assertEqual(true, lists:member(R, Results))
        end),
  reg_server:stop().
```

After making the above changes, all tests pass. Note that our last test produces a large number of interleaving sequences (more than 800,000) and takes several minutes to complete. At this point, we can turn on preemption bounding by selecting the Enable preemption bounding option under Edit→Preferences and use the default value of two for the preemption bound. Running the test again, a considerably lower number of interleaving sequences (about 1,700) is produced in a matter of seconds.

However, we would also like to know if the previous defect would have been detected if a preemption bound of two were used. Temporarily reverting the changes of Program 8 and running the test again, we can see that the defect is indeed detected. In fact, the defect is even detected with a preemption bound of zero. This suggests that a low preemption bound may often be enough to reveal many common concurrency errors.

Our next task is dealing with the case of pinging the server when it is not running. The current version of `request/1` should readily handle this case without the need for a timeout. To check this, we employ the tests shown in Test 6. The first test checks that a call to `ping/0` returns `server_down` before starting the server for the first time as well as after starting and stopping the server. The second test checks that when a process attempts to ping the server while another process is trying to stop it, the `ping/0` call will return either `pong` or `server_down`. Both tests pass successfully and we proceed to our next task, namely dealing with multiple calls to `start/0`.

Like we did with multiple stop calls, we will test multiple start calls by first using one process and then two. A call to `start/0` should return `already_started` in case the server is already running. The corresponding single process test is shown in Test 7. To make this test work, we call `whereis/1` in `start/0` before starting the server. If `whereis/1` returns `undefined`, the server is spawned and registered as before, otherwise the server is already running, so `already_started` is returned. The newly added server code is shown in Program 9. The test passes with the new version of `start/0` and we are ready to write the two-process variant.

The test for two concurrent `start/0` calls is shown in Test 8 and is similar to the one we used in Test 5 to test for two concurrent `stop/0` calls. Two processes are spawned and concurrently attempt to start the server. One of them should observe a return value of `ok` and the other a return value of `already_started`. The new test fails when run in Concuerror. As seen from the detailed interleaving information, when the two processes call `whereis/1` before the server is started, they both try to spawn and register the server and one of them fails with an exception. Ideally, we would like the block of code containing `whereis`, `spawn` and `register` to be executed atomically, but this is currently not possible in Erlang.

Test Code 7 Test for two start calls by one process

```
multiple_starts_test() ->
  reg_server:start(),
  ?assertEqual(already_started, reg_server:start()),
  reg_server:stop().
```

Program 9 Check if the server is already running before starting it

```
start() ->
  case whereis(?REG_NAME) of
    undefined ->
      Pid = spawn(fun() -> loop() end),
      register(?REG_NAME, Pid),
      ok;
    _Pid -> already_started
  end.
```

Test Code 8 Test for two concurrent start calls by two processes

```
multiple_concurrent_starts_test() ->
  Self = self(),
  spawn(fun() -> Self ! reg_server:start() end),
  spawn(fun() -> Self ! reg_server:start() end),
  ?assertEqual(lists:sort([already_started, ok]),
               lists:sort(receive_two())),
  reg_server:stop().
```

Instead, we may use the solution of allowing multiple processes to be spawned, but only one of them to be properly registered as the server process. Subsequently, processes that fail to be registered have to be killed. To accomplish the above, we use a `try...catch` expression around the `register/2` call and send a message to the spurious process, forcing it to exit, as shown in Program 10. Note that we cannot use a call to `request/1` to kill such a process, because it has not been registered.

Our last test produces a large number of interleaving sequences, thus it is preferable to start with a small preemption bound and gradually increase it while finding and correcting any errors encountered. At this point, all of our tests pass with or without preemption bounding, so we have successfully completed the implementation of the functions for starting and stopping our server.

4.4 Attaching processes

Any process should be able to become attached to our server. A unique integer, that for our purposes will be called a registration number, is assigned to each attached process. The server only allows for a limited number of attached processes. We will define the maximum number

Program 10 Use `try...catch` to avoid spurious server processes

```
start() ->
  case whereis(?REG_NAME) of
    undefined ->
      Pid = spawn(fun() -> loop() end),
      try register(?REG_NAME, Pid) of
        true -> ok
      catch
        error:badarg ->
          Pid ! {?REG_REQUEST, kill},
          already_started
      end;
    _Pid -> already_started
  end.

...

loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      reply(Target, pong),
      loop();
    {?REG_REQUEST, Target, stop} ->
      unregister(?REG_NAME),
      reply(Target, ok);
    {?REG_REQUEST, kill} -> killed
  end.
```

Test Code 9 Test for attaching two processes to the server

```
attach_test() ->
  Self = self(),
  reg_server:start(),
  RegNum1 = reg_server:attach(),
  spawn(fun() ->
          RegNum2 = reg_server:attach(),
          ?assertEqual(RegNum2, reg_server:ping()),
          ?assertEqual(false, RegNum1 =:= RegNum2),
          Self ! done
        end),
  ?assertEqual(RegNum1, reg_server:ping()),
  receive done -> reg_server:stop() end.
```

of attached processes in the `MAX_ATTACHED` macro of an external `reg_server.hrl` file, so that it can be included in both the server code and the tests.

To be able to check if a process is attached, we will extend the `ping/0` function to return the calling process' registration number in case it is attached to the server. The attachment of two processes can be checked using the test of Test 9. We use `attach/0` to attach the calling processes to the server. Function `attach/0` returns the registration number that was assigned to the calling process. The test also checks that the two processes are not given equal registration numbers. What happens when a process calls `attach/0` while it is already attached or when the maximum number of attached processes has been reached? Should a process become detached when it exits? All these matters will be handled later.

For now, let us try to make our last test pass. The server can use an ordered set to keep track of the free registration numbers. Initially, this set will contain the numbers from 1 to `MAX_ATTACHED`. Additionally, the server can

Program 11 Add `attach/0` to the server

```
-module(reg_server).

-export([attach/0, ping/0, start/0, stop/0]).

-include("reg_server.hrl").

...

-record(state, {free, reg}).

attach() ->
  request(attach).

start() ->
  case whereis(?REG_NAME) of
    undefined ->
      Pid = spawn(fun() -> loop(initState()) end),
    ...
  end.

initState() ->
  FreeList = lists:seq(1, ?MAX_ATTACHED),
  #state{free = ordsets:from_list(FreeList),
         reg = dict:new()}.

...

loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, attach} ->
      [RegNum|NewFreeList] = ordsets:to_list(Free),
      NewReg = dict:store(Target, RegNum, Reg),
      reply(Target, RegNum),
      NewFree = ordsets:from_list(NewFreeList),
      NewState = State#state{free = NewFree,
                             reg = NewReg},
      loop(NewState);
    {?REG_REQUEST, Target, ping} ->
      case dict:find(Target, Reg) of
        {ok, RegNum} -> reply(Target, RegNum);
        error -> reply(Target, pong)
      end,
      loop(State);
    ...
  end.
```

use a dictionary to store mappings from registered processes' pids to their corresponding registration numbers. Both structures can be packed into a record that will represent the state of the server and will be passed as an argument to its loop. Other than that, the request-reply infrastructure that we have created so far can also be used here, in order to automatically handle the case of calling `attach/0` when the server is down. The implementation of all of the above is shown in Program 11.

Every time a process requests to become attached, the server removes a free registration number from its set and adds a 'pid to registration number' mapping to the dictionary. Note that the header file `reg_server.hrl` contains only the attribute `-define(MAX_ATTACHED, 2).` (A small number is used here to simplify our testing.) We do not need to import this header file in Concuerror; it is automatically recognized since it resides in the same directory

Test Code 10 Test for already attached processes and full server

```
-module(reg_server_tests).

-include_lib("eunit/include/eunit.hrl").
-include("reg_server.hrl").

already_attached_test() ->
  reg_server:start(),
  RegNum = reg_server:attach(),
  ?assertEqual(RegNum, reg_server:attach()),
  reg_server:stop().

max_attached_proc_test() ->
  reg_server:start(),
  L = lists:seq(1, ?MAX_ATTACHED),
  Ps = [spawn_attach() || _ <- L],
  ?assertEqual(server_full, reg_server:attach()),
  lists:foreach(fun(Pid) -> Pid ! ok end, Ps),
  reg_server:stop().

attach_and_wait(Target) ->
  reg_server:attach(),
  Target ! done,
  receive ok -> ok end.

spawn_attach() ->
  Self = self(),
  Pid = spawn(fun() -> attach_and_wait(Self) end),
  receive done -> Pid end.
```

as the `.erl` files. With the above additions the test passes successfully.

We will work on our next two tasks in one step because they are fairly simple. The first is to handle an `attach/0` call from an already attached process. The server can either ignore the call and return the already allocated registration number or, alternatively, return a special value indicating an error. We will choose the first option here and check it with the first test shown in Test 10. The second task is to handle the case of a process requesting to become attached when the server is already full of attached processes, as determined by `MAX_ATTACHED`. In this case we would like the `attach/0` call to return `server_full`, as shown in the second test of Test 10.

For the tests to pass, before attaching the requesting process, we need to check whether it already has a registration number and, if not, whether there are any free registration numbers. This is easily done inside the server loop, as shown in Program 12.

Our new tests pass successfully in Concuerror. The second test produces 822 interleaving sequences for a preemption bound of two. The current execution of our test contains a total of four processes—the initial process, the server process and two additional spawned processes. Let us change the value of `MAX_ATTACHED` to one, so that our test contains three processes. Leaving the preemption bound at two, there are now only 65 interleaving sequences produced. Trying the same with `MAX_ATTACHED` equal to three and four, 9,789 and 118,038 interleaving sequences are produced, respectively. This clearly indicates that the number of interleaving sequences grows exponentially as the number of testing processes increases.

Program 12 Extend server to deal with already registered processes and full server

```
loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, attach} ->
      case dict:find(Target, Reg) of
        {ok, RegNum} ->
          reply(Target, RegNum),
          loop(State);
        error ->
          case ordsets:to_list(Free) of
            [] ->
              reply(Target, server_full),
              loop(State);
            [RegNum|NewFreeList] ->
              NewReg = dict:store(Target, RegNum, Reg),
              reply(Target, RegNum),
              NewFree = ordsets:from_list(NewFreeList),
              NewState = State#state{free = NewFree,
                                     reg = NewReg},
              loop(NewState)
          end
      end;
    ...
  end.
```

Test Code 11 Test for detaching

```
detach_test() ->
  reg_server:start(),
  reg_server:attach(),
  reg_server:detach(),
  ?assertEqual(pong, reg_server:ping()),
  reg_server:stop().
```

Test Code 12 Test for reattaching after detaching

```
detach_attach_test() ->
  Self = self(),
  reg_server:start(),
  L = lists:seq(1, ?MAX_ATTACHED - 1),
  Ps = [spawn_attach() || _ <- L],
  LastProc = spawn(fun() ->
                     RegNum = reg_server:attach(),
                     reg_server:detach(),
                     Self ! RegNum,
                     receive ok -> ok end
                   end),
  receive RegNum -> ok end,
  ?assertEqual(RegNum, reg_server:attach()),
  lists:foreach(fun(Pid) -> Pid ! ok end, [LastProc|Ps]),
  reg_server:stop().
```

4.5 Detaching processes

We have finished our tasks concerning the attachment of processes and now proceed to handling their detachment from the server. When a process is detached from the server using a `detach/0` call, it should no longer have an assigned registration number, thus a `ping/0` call to the server after the process' detachment should return `pong`. This is checked by the test in Test 11. Moreover, the server should make the corresponding registration number available to future processes requesting attachment. To check

Program 13 Add `detach/0` to the server

```
-module(reg_server).

-export([attach/0, detach/0, ping/0, start/0, stop/0]).

...

detach() ->
  request(detach).

...

loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, detach} ->
      RegNum = dict:fetch(Target, Reg),
      NewReg = dict:erase(Target, Reg),
      NewFree = ordsets:add_element(RegNum, Free),
      reply(Target, ok),
      NewState = State#state{free = NewFree,
                             reg = NewReg},
      loop(NewState);
    ...
  end.
```

Test Code 13 Test for trying to detach an unattached process

```
detach_non_attached_test() ->
  reg_server:start(),
  ?assertEqual(ok, reg_server:detach()),
  reg_server:stop().
```

this, we can extend the second test of Test 10 so that when one of the processes is detached, another process becomes attached with the same registration number, which is the only one available at the time, as shown in Test 12. To make the tests pass, the server needs to remove the existing mapping from the dictionary and add the freed registration number back to the set, as shown in Program 13.

After having successfully run our last tests and before proceeding to our final task, we have to deal with one more issue, namely handling the case of a process calling `detach/0` without being attached. Similarly to the case of the double `attach/0` call that we encountered previously, we can either ignore this case and return `ok` or consider it an error. Again, we opt for the silent approach, as shown in Test 13. The fix is fairly simple and is shown in Program 14.

Currently, an attached process is detached only when it calls `detach/0`. If an attached process exits without having been detached on its own, its registration number will remain occupied forever. Consequently, our last task is to enforce detaching a process as soon as it exits. To check this, we modify the test in Test 11 so that the last process being spawned, instead of explicitly detaching itself, simply terminates its execution. The main testing process tries to attach itself after receiving the `'EXIT'` message from this previously attached process. For the `'EXIT'` message to be received, a `process_flag/2` call is used to activate the `trap_exit` flag of the main process. The test is shown in Test 14.

Program 14 Deal with trying to detach an unattached process

```
loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, detach} ->
      case dict:is_key(Target, Reg) of
        false ->
          reply(Target, ok),
          loop(State);
        true ->
          RegNum = dict:fetch(Target, Reg),
          ...
          loop(NewState)
      end;
    ...
  end.
```

Test Code 14 Test for detaching a process as soon as it exits

```
detach_on_exit_test() ->
  Self = self(),
  reg_server:start(),
  L = lists:seq(1, ?MAX_ATTACHED - 1),
  Ps = [spawn_attach() || _ <- L],
  process_flag(trap_exit, true),
  LastProc = spawn_link(fun() ->
                          Self ! reg_server:attach()
                        end),
  receive RegNum -> ok end,
  receive {'EXIT', LastProc, normal} -> ok end,
  ?assertEqual(RegNum, reg_server:attach()),
  lists:foreach(fun(Pid) -> Pid ! ok end, [LastProc|Ps]),
  reg_server:stop().
```

The server needs to know when a process exits in order to take action and detach it. This means that we have to use either links or monitors to keep track of attached processes. There is no reason to use two-sided links here, thus we will have the server create a monitor for every process that becomes attached. Also, as soon as the server receives a `'DOWN'` message about an attached process, that process will be detached. In Program 15 we implement the above functionality and move the detachment operation into a separate function to avoid code duplication.

At this point, our final test passes successfully in Concuerror, as do all of our previous tests. We do not claim that our code is entirely free of bugs. However, the tests that we have written check the basic functionality of our server and make us confident that any scenario that may occur in practice and has been covered by our tests will actually work as expected.

5. Concluding Remarks

Summing up, we would like to stress some important points that came up during the presentation of our development process. First, we quickly realized that conventional testing using EUnit was not able to expose the concurrency errors we encountered, whereas trying to test with the current version of McErlang presented some serious usability issues. Given that our server was destined to be used in a highly concurrent environment, Concuerror not only allowed us to verify that our tests pass under

Program 15 Detach a process as soon as it exits

```
loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, attach} ->
      ...
            [RegNum|NewFreeList] ->
            NewReg = dict:store(Target, RegNum, Reg),
            monitor(process, Target),
            reply(Target, RegNum),
      ...
    {?REG_REQUEST, Target, detach} ->
      {Reply, NewFree, NewReg} =
        detach_proc(Target, Free, Reg),
      reply(Target, Reply),
      NewState = State#state{free = NewFree,
                             reg = NewReg},
      loop(NewState);
    {'DOWN', _Ref, process, Target, _Info} ->
      NewState =
        case dict:is_key(Target, Reg) of
          true ->
            {ok, NewFree, NewReg} =
              detach_proc(Target, Free, Reg),
            State#state{free = NewFree,
                        reg = NewReg};
          false -> State
        end,
      loop(NewState)
  end.

detach_proc(Target, Free, Reg) ->
  case dict:is_key(Target, Reg) of
    false -> {ok, Free, Reg};
    true ->
      RegNum = dict:fetch(Target, Reg),
      NewReg = dict:erase(Target, Reg),
      NewFree = ordsets:add_element(RegNum, Free),
      {ok, NewFree, NewReg}
  end.
```

some random interleaving, but we were guaranteed that under any interleaving our program was still robust and correct with respect to our test suite. Second, for all of our tests, a preemption bound of two was enough to uncover any concurrency-related defect. It is usually convenient to start with a low preemption bound and gradually increase it for more thorough testing. Third, we observed the exponential increase in the number of interleaving sequences with respect to the number of processes. This suggests writing our tests to use only a few processes and then generalizing their results for an arbitrary number of them. Last, but not least, besides revealing errors, Concuerror also helped us understand their cause by displaying detailed interleaving information. (This is better experienced by actually running Concuerror through the example of this paper.) By walking through the erroneous interleaving sequences, we could quickly understand and correct the source of each defect.

The complete code of our example can be found at `https://github.com/mariachris/Concuerror` under the `resources/tdd` directory.

References

[1] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM.

[2] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[3] R. Carlsson and M. Rémond. EUnit: A lightweight unit testing framework for Erlang. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 1–1, New York, NY, USA, 2006. ACM.

[4] K. Claessen, M. Pałka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 149–160, New York, NY, USA, 2009. ACM.

[5] C. B. Earle and L.-Å. Fredlund. Recent improvements to the McErlang model checker. In *Proceedings of the 8th ACM SIGPLAN Workshop on Erlang*, pages 93–100, New York, NY, USA, 2009. ACM.

[6] L.-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 125–136, New York, NY, USA, 2007. ACM.

[7] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455, New York, NY, USA, 2007. ACM.

[8] T. Nagy and A. Nagyné Víg. Erlang testing and tools survey. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 21–28, New York, NY, USA, 2008. ACM.