Finding Race Conditions in Erlang with QuickCheck and PULSE

Koen Claessen Michał Pałka
Nicholas Smallbone
Chalmers University of Technology,
Gothenburg, Sweden
koen@chalmers.se
michal.palka@chalmers.se
nicsma@chalmers.se

John Hughes Hans Svensson
Thomas Arts
Chalmers University of Technology and
Quviq AB
rjmh@chalmers.se
hans.svensson@ituniv.se
thomas.arts@ituniv.se

Ulf Wiger
Erlang Training and Consulting
ulf.wiger@erlang-consulting.com

Abstract

We address the problem of testing and debugging concurrent, distributed Erlang applications. In concurrent programs, race conditions are a common class of bugs and are very hard to find in practice. Traditional unit testing is normally unable to help finding all race conditions, because their occurrence depends so much on timing. Therefore, race conditions are often found during system testing, where due to the vast amount of code under test, it is often hard to diagnose the error resulting from race conditions. We present three tools (QuickCheck, PULSE, and a visualizer) that in combination can be used to test and debug concurrent programs in unit testing with a much better possibility of detecting race conditions. We evaluate our method on an industrial concurrent case study and illustrate how we find and analyze the race conditions.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Distributed debugging

General Terms Verification

Keywords QuickCheck, Race Conditions, Erlang

1. Introduction

Concurrent programming is notoriously difficult, because the non-deterministic interleaving of events in concurrent processes can lead software to work most of the time, but fail in rare and hard-to-reproduce circumstances when an unfortunate order of events occurs. Such failures are called *race conditions*. In particular, concurrent software may work perfectly well during *unit testing*, when individual modules (or "software units") are tested in isolation, but fail later on during *system testing*. Even if unit tests cover all aspects of the units, we still can detect concurrency errors when all components of a software system are tested together. Timing delays caused by other components lead to new, previously untested, schedules of actions performed by the individual units. In the worst case, bugs may not appear until the system is put under heavy load in production. Errors discovered in these late stages are far more expensive to diagnose and correct, than errors found during unit testing. Another cause of concurrency errors showing up at a late stage is when well-tested software is ported from a single-core to a multi-core processor. In that case, one would really benefit from a hierarchical approach to testing legacy code in order to simplify debugging of faults encountered.

The Erlang programming language (Armstrong 2007) is designed to simplify concurrent programming. Erlang processes do not share memory, and Erlang data structures are immutable, so the kind of *data races* which plague imperative programs, in which concurrent processes race to read and write the same memory location, simply cannot occur. However, this does not mean that Erlang programs are immune to race conditions. For example, the order in which messages are delivered to a process may be non-deterministic, and an unexpected order may lead to failure. Likewise, Erlang processes can share *data*, even if they do not share memory—the file store is one good example of shared mutable data, but there are also shared data-structures managed by the Erlang virtual machine, which processes can race to read and write.

Industrial experience is that the late discovery of race conditions is a real problem for Erlang developers too (Cronqvist 2004). Moreover, these race conditions are often caused by design errors, which are particularly expensive to repair. If these race conditions could be found during unit testing instead, then this would definitely reduce the cost of software development.

In this paper, we describe tools we have developed for finding race conditions in Erlang code during unit testing. Our approach is based on *property-based testing* using QuickCheck (Claessen and Hughes 2000), in a commercial version for Erlang developed by Quviq AB (Hughes 2007; Arts et al. 2006). Its salient features are described in section 3. We develop a suitable property for testing parallel code, and a method for generating parallel test cases, in section 4. To test a wide variety of schedules, we developed a randomizing scheduler for Erlang called PULSE, which we explain in section 5. PULSE records a trace during each test, but interpreting the traces is difficult, so we developed a trace visualizer which is described in section 6. We evaluate our tools by applying them to an industrial case study, which is introduced in section 2, then used as a running example throughout the paper. This code was already known to contain bugs (thanks to earlier experiments with QuickCheck in 2005), but we were previously unable to *diagnose* the problems. Using the tools described here, we were able to find and fix two race conditions, and identify a fundamental flaw in the API.

2. Introducing our case study: the process registry

We begin by introducing the industrial case that we apply our tools and techniques to. In Erlang, each process has a unique, dynamically-assigned identifier ("pid"), and to send a message to a process, one must know its pid. To enable processes to discover the pids of central services, such as error logging, Erlang provides a *process registry*—a kind of local name server—which associates static names with pids. The Erlang VM provides operations to *register* a pid with a name, to *look up* the pid associated with a name, and to *unregister* a name, removing any association with that name from the registry. The registry holds only *live* processes; when registered processes crash, then they are automatically unregistered. The registry is heavily used to provide access to system services: a newly started Erlang node already contains 13 registered processes.

However, the built-in process registry imposes several, sometimes unwelcome, limitations: registered names are restricted to be atoms, the same process cannot be registered with multiple names, and there is no efficient way to search the registry (other than by name lookup). This motivated Ulf Wiger (who was working for Ericsson at the time) to develop an extended process registry *in Erlang*, which could be modified and extended much more easily than the one in the virtual machine. Wiger's process registry software has been in use in Ericsson products for several years (Wiger 2007).

In our case study we consider an earlier prototype of this software, called `proc_reg`, incorporating an optimization that proved not to work. The API supported is just: `reg(Name,Pid)` to register a pid, `where(Name)` to look up a pid, `unreg(Name)` to remove a registration, and `send(Name,Msg)` to send a message to a registered process. Like the production code, `proc_reg` stores the association between names and pids in *Erlang Term Storage* ("ETS tables")—hash tables, managed by the virtual machine, that hold a set of tuples and support tuple-lookup using the first component as a key (cf. Armstrong 2007, chap 15). It also creates a *monitor* for each registered process, whose effect is to send `proc_reg` a "`DOWN`" message if the registered process crashes, so it can be removed from the registry. Two ETS table entries are created for each registration: a "forward" entry that maps names to pids, and a "reverse" entry that maps registered pids to the monitor reference. The monitor reference is needed to turn off monitoring again, if the process should later be unregistered.

Also like the production code, `proc_reg` is implemented as a server process using Erlang's *generic server* library (cf. Armstrong 2007, chap 16). This library provides a robust way to build client-server systems, in which clients make "synchronous calls" to the server by sending a `call` message, and awaiting a matching reply[1]. Each operation—`reg`, `where`, `unreg` and `send`—is supported by a different `call` message. The operations are actually executed by the server, one at a time, and so no race conditions can arise.

At least, this is the theory. In practice there is a small cost to the generic server approach: each request sends two messages and requires two context switches, and although these are cheap in Erlang, they are not free, and turn out to be a bottleneck in system start-up times, for example. The prototype `proc_reg` attempts to optimize this, by moving the creation of the first "forward" ETS table entry into the clients. If this succeeds (because there is no previous entry with that name), then clients just make an "asynchronous" call to the server (a so-called `cast` message, with no reply) to inform it that it should complete the registration later. This avoids a context switch, and reduces two messages to one. If there *is* already a registered process with the same name, then the `reg` operation fails (with an exception)—unless, of course, the process is dead. In this case, the process will soon be removed from the registry by the server; clients ask the server to "audit" the dead process to hurry this along, then complete their registration as before.

[1] Unique identifiers are generated for each call, and returned in the reply, so that no message confusion can occur.

This prototype was one of the first pieces of software to be tested using QuickCheck at Ericsson. At the time, in late 2005, it was believed to work, and indeed was accompanied by quite an extensive suite of unit tests—including cases designed specifically to test for race conditions. We used QuickCheck to generate and run random sequences of API calls in two concurrent processes, and instrumented the `proc_reg` code with calls to `yield()` (which cedes control to the scheduler) to cause fine-grain interleaving of concurrent operations. By so doing, we could show that `proc_reg` was incorrect, since our tests failed. But the failing test cases we found were large, complex, and very hard to understand, and we were unable to use them to diagnose the problem. As a result, this version of `proc_reg` was abandoned, and development of the production version continued without the optimization.

While we were pleased that QuickCheck was able to reveal bugs in `proc_reg`, we were unsatisfied that it could not help us to find them. Moreover, the QuickCheck property we used to test it was hard-to-define and ad hoc—and not easily reusable to test any other software. This paper is the story of how we addressed these problems—and returned to apply our new methods successfully to the example that defeated us before.

3. An Overview of Quviq QuickCheck

QuickCheck (Claessen and Hughes 2000) is a tool that tests universally quantified *properties*, instead of single test cases. QuickCheck generates random test cases from each property, tests whether the property is true in that case, and reports cases for which the property fails. Recent versions also "shrink" failing test cases automatically, by searching for similar, but smaller test cases that also fail. The result of shrinking is a "minimal"[2] failing case, which often makes the root cause of the problem very easy to find.

Quviq QuickCheck is a commercial version that includes support for model-based testing using a state machine model (Hughes 2007). This means that it has standard support for generating sequences of API calls using this state machine model. It has been used to test a wide variety of industrial software, such as Ericsson's Media Proxy (Arts et al. 2006) among others. State machine models are tested using an additional library, `eqc_statem`, which invokes call-backs supplied by the user to generate and test random, well-formed sequences of calls to the software under test. We illustrate `eqc_statem` by giving fragments of a (sequential) specification of `proc_reg`.

Let us begin with an example of a generated test case (a sequence of API calls).

```
[{set,{var,1},{call,proc_reg_eqc,spawn,[]}},
 {set,{var,2},{call,proc_reg,where,[c]}},
 {set,{var,3},{call,proc_reg_eqc,spawn,[]}},
 {set,{var,4},{call,proc_reg_eqc,kill,[{var,1}]}},
 {set,{var,5},{call,proc_reg,where,[d]}},
 {set,{var,6},{call,proc_reg_eqc,reg,[a,{var,1}]}},
 {set,{var,7},{call,proc_reg_eqc,spawn,[]}}]
```

`eqc_statem` test cases are lists of *symbolic commands* represented by Erlang terms, each of which binds a symbolic variable (such as `{var,1}`) to the result of a function call, where `{call,M,F,Args}` represents a call of function `F` in module `M` with arguments `Args`[3]. Note that previously bound variables can be used in later calls. Test cases for `proc_reg` in particular randomly spawn processes (to use as test data), kill them (to simulate crashes at random times), or pass them to `proc_reg` operations. Here `proc_reg_eqc` is the module containing the specification of `proc_reg`, in which we define local

[2] In the sense that it cannot shrink to a failing test with the shrinking algorithm used.

[3] In Erlang, variables start with an uppercase character, whereas atoms (constants) start with a lowercase character.

versions of reg and unreg which just call proc_reg and catch any exceptions. This allows us to write properties that test whether an exception is raised correctly or not. (An *uncaught* exception in a test is interpreted as a failure of the entire test).

We model the state of a test case as a list of processes spawned, processes killed, and the {Name,Pid} pairs currently in the registry. We normally encapsulate the state in a record:

```
-record(state,{pids=[],regs=[],killed=[]}).
```

eqc_statem generates random calls using the call-back function command that we supply as part of the state machine model, with the test case state as its argument:

```
command(S) ->
  oneof(
    [{call,?MODULE,spawn,[]}] ++
    [{call,?MODULE,kill, [elements(S#state.pids)]}
           || S#state.pids/=[]] ++
    [{call,?MODULE,reg,[name(),elements(S#state.pids)]}
           || S#state.pids/=[]] ++
    [{call,?MODULE,unreg,[name()]}] ++
    [{call,proc_reg,where,[name()]}]).

name() -> elements([a,b,c,d]).
```

The function oneof is a QuickCheck generator that randomly uses one element from a list of generators; in this case, the list of candidates to choose from depends on the test case state. ([X||P] is a degenerate list comprehension, that evaluates to the empty list if P is false, and [X] if P is true—so reg and kill can be generated only if there are pids available to pass to them.) We decided not to include send in test cases, because its implementation is quite trivial. The macro ?MODULE expands to the name of the module that it appears in, proc_reg_eqc in this case.

The next_state function specifies how each call is supposed to change the state:

```
next_state(S,V,{call,_,spawn,_}) ->
  S#state{pids=[V|S#state.pids]};
next_state(S,V,{call,_,kill,[Pid]}) ->
  S#state{killed=[Pid|S#state.killed],
          regs=[{Name,P} ||
                 {Name,P} <- S#state.regs, Pid /= P]};
next_state(S,_V,{call,_,reg,[Name,Pid]}) ->
  case register_ok(S,Name,Pid) andalso
         not lists:member(Pid,S#state.killed) of
       true ->
         S#state{regs=[{Name,Pid}|S#state.regs]};
       false ->
         S
  end;
next_state(S,_V,{call,_,unreg,[Name]}) ->
  S#state{regs=lists:keydelete(Name,1,S#state.regs)};
next_state(S,_V,{call,_,where,[_]}) ->
  S.

register_ok(S,Name,Pid) ->
  not lists:keymember(Name,1,S#state.regs).
```

Note that the new state can depend on the *result* of the call (the second argument V), as in the first clause above. Note also that killing a process removes it from the registry (in the model), and that registering a dead process, or a name that is already registered (see register_ok), should not change the registry state. We do allow the same pid to be registered with several names, however.

When running tests, eqc_statem checks the postcondition of each call, specified via another call-back that is given the state before the call, and the actual result returned, as arguments. Since we catch exceptions in each call, which converts them into values of the form {'EXIT',Reason}, our proc_reg postconditions can test that exceptions are raised under precisely the right circumstances:

```
postcondition(S,{call,_,reg,[Name,Pid]},Res) ->
  case Res of
       true ->
         register_ok(S,Name,Pid);
       {'EXIT',_} ->
         not register_ok(S,Name,Pid)
  end;
postcondition(S,{call,_,unreg,[Name]},Res) ->
  case Res of
       true ->
         unregister_ok(S,Name);
       {'EXIT',_} ->
         not unregister_ok(S,Name)
  end;
postcondition(S,{call,_,where,[Name]},Res) ->
  lists:member({Name,Res},S#state.regs);
postcondition(_S,{call,_,_,_},_Res) ->
  true.

unregister_ok(S,Name) ->
  lists:keymember(Name,1,S#state.regs).
```

Note that reg(Name,Pid) and unreg(Name) are required to return exceptions if Name is already used/not used respectively, but that reg always returns true if Pid is dead, even though no registration is performed! This may perhaps seem a surprising design decision, but it is consistent. As a comparison, the built-in process registry sometimes returns true and sometimes raises an exception when registering dead processes. This is due to the fact that a context switch is required to clean up.

State machine models can also specify a *precondition* for each call, which restricts test cases to those in which all preconditions hold. In this example, we could have used preconditions to exclude test cases that we expect to raise exceptions—but we prefer to allow any test case, and check that exceptions are raised correctly, so we define all preconditions to be true.

With these four call-backs, plus another call-back specifying the initial state, our specification is almost complete. It only remains to define the top-level property which generates and runs tests:

```
prop_proc_reg() ->
  ?FORALL(Cmds,commands(?MODULE),
     begin
       {ok,ETSTabs} = proc_reg_tabs:start_link(),
       {ok,Server} = proc_reg:start_link(),
       {H,S,Res} = run_commands(?MODULE,Cmds),
       cleanup(ETSTabs,Server),
       Res == ok
     end).
```

Here ?FORALL binds Cmds to a random list of commands generated by commands, then we initialize the registry, run the commands, clean up, and check that the result of the run (Res) was a success. Here commands and run_commands are provided by eqc_statem, and take the current module name as an argument in order to find the right call-backs. The other components of run_commands' result, H and S, record information about the test run, and are of interest primarily when a test fails. This is not the case here: *sequential* testing of proc_reg does not fail.

4. Parallel Testing with QuickCheck

4.1 A Parallel Correctness Criterion

In order to test for race conditions, we need to generate test cases that are executed in parallel, and we also need *a specification of the correct parallel behavior*. We have chosen, in this paper, to use a specification that just says that *the API operations we are testing should behave atomically*.

How can we tell from test results whether or not each operation "behaved atomically"? Following Lamport (1979) and Herlihy and

Wing (1987), we consider a test to have passed if the observed results are the same as some possible sequential execution of the operations in the test—that is, a possible interleaving of the parallel processes in the test.

Of course, testing for atomic behavior is just a special case, and in general we may need to test other properties of concurrent code too—but we believe that this is a very important special case. Indeed, Herlihy and Wing claim that their notion of *linearizability* "focuses exclusively on a subset of concurrent computations that we believe to be the most interesting and useful"; we agree. In particular, atomicity is of great interest for the process registry.

One great advantage of this approach is that we can reuse the *same* specification of the sequential behavior of an API, to test its behavior when invocations take place in parallel. We need only find the right linearization of the API calls in the test, and then use the sequential specification to determine whether or not the test has passed. We have implemented this idea in a new QuickCheck module, `eqc_par_statem`, which takes the *same* state-machine specifications as `eqc_statem`, but tests the API in parallel instead. While state machine specifications require some investment to produce in real situations, this means that we can test for race conditions *with no further investment* in developing a parallel specification. It also means that, as the code under test evolves, we can switch freely to-and-fro between sequential testing to ensure the basic behavior still works, and race condition testing using `eqc_par_statem`.

The difficulty with this approach is that, when we run a test, then there is no way to *observe* the sequence in which the API operations take effect. (For example, a server is under no obligation to service requests in the order in which they are made, so observing this order would tell us nothing.) In general, the only way to tell whether there is a possible sequentialization of a test case which can explain the observed test results, is to *enumerate* all possible sequentializations. This is prohibitively expensive unless care is taken when test cases are generated.

4.2 Generating Parallel Test Cases

Our first approach to parallel test case generation was to use the standard Quviq QuickCheck library `eqc_statem` to generate sequential test cases, then execute all the calls in the test case in parallel, constrained only by the data dependencies between them (which arise from symbolic variables, bound in one command, being used in a later one). This generates a great deal of parallelism, but sadly also an enormous number of possible serializations—in the worst case in which there are no data dependencies, a sequence of n commands generates $n!$ possible serializations. It is not practically feasible to implement a test oracle for parallel tests of this sort.

Instead, we decided to generate parallel test cases of a more restricted form. They consist of an initial sequential prefix, to put the system under test into a random state, followed by exactly *two* sequences of calls which are performed in parallel. Thus the possible serializations consist of the initial prefix, followed by an interleaving of the two parallel sequences. (Lu et al. (2008) gives clear evidence that it is possible to discover a large fraction of the concurrency related bugs by using only two parallel threads/processes.) We generate parallel test cases by parallelizing a suffix of an `eqc_statem` test case, separating it into two lists of commands of roughly equal length, with no mutual data dependencies, which are *non-interfering* according to the sequential specification. By non-interference, we mean that all command preconditions are satisfied in any interleaving of the two lists, which is necessary to prevent tests from failing because a precondition was unsatisfied—not an interesting failure. We avoid parallelizing too long a suffix (longer than 16 commands), to keep the number of possible interleavings feasible to enumerate (about 10,000 in the worst case). Finally, we run tests by first running the prefix, then spawning two processes to run the two command-lists in parallel, and collecting their results, which will be non-deterministic depending on the actual parallel scheduling of events.

We decide whether a test has passed, by attempting to construct a sequentialization of the test case which explains the results observed. We begin with the sequential prefix of the test case, and use the `next_state` function of the `eqc_statem` model to compute the test case state after this prefix is completed. Then we try to *extend* the sequential prefix, one command at a time, by choosing the first command from one of the parallel branches, and moving it into the prefix. This is allowed only if the `postcondition` specified in the `eqc_statem` model accepts the actual result returned by the command when we ran the test. If so, we use the `next_state` function to compute the state after this command, and continue. If the first commands of *both* branches fulfilled their postconditions, then we cannot yet determine which command took effect first, and we must explore both possibilities further. If we succeed in moving *all* commands from the parallel branches into the sequential prefix, such that all postconditions are satisfied, then we have found a possible sequentialization of the test case explaining the results we observed. If our search fails, then there is *no* such sequence, and the test failed.

This is a greedy algorithm: as soon as a postcondition fails, then we can discard all potential sequentializations with the failing command as the next one in the sequence. This happens often enough to make the search reasonably fast in practice. As a further optimization, we memoize the search function on the remaining parallel branches and the test case state. This is useful, for example, when searching for a sequentialization of $[A, B]$ and $[C, D]$, if both $[A, C]$ and $[C, A]$ are possible prefixes, and they lead to the same test state—for then we need only try to sequentialize $[B]$ and $[D]$ once. We memoize the non-interference test in a similar way, and these optimizations give an appreciable, though not dramatic, speed-up in our experiments—of about 20%. With these optimizations, generating and running parallel tests is acceptably fast.

4.3 Shrinking Parallel Test Cases

When a test fails, QuickCheck attempts to *shrink* the failing test by searching for a similar, but smaller test case which also fails. QuickCheck can often report minimal failing examples, which is a great help in fault diagnosis. `eqc_statem` already has built-in shrinking methods, of which the most important tries to delete unnecessary commands from the test case, and `eqc_par_statem` inherits these methods. But we also implement an additional shrinking method for parallel test cases: if it is possible to move a command from one of the parallel suffixes into the sequential prefix, then we do so. Thus the minimal test cases we find are "minimally parallel"—we know that the parallel branches in the failing tests reported really do race, because everything that can be made sequential, is sequential. This also assists fault diagnosis.

4.4 Testing `proc_reg` for Race Conditions

To test the process registry using `eqc_par_statem`, it is only necessary to modify the property in Section 2 to use `eqc_par_statem` rather than `eqc_statem` to generate and run test cases.

```
prop_proc_reg_parallel() ->
  ?FORALL(Cmds,eqc_par_statem:commands(?MODULE),
     begin
       {ok,ETSTabs} = proc_reg_tabs:start_link(),
       {ok,Server} = proc_reg:start_link(),
       {H,{A,B},Res} =
         eqc_par_statem:run_commands(?MODULE,Cmds),
       cleanup(ETSTabs,Server),
       Res == ok
     end).
```

The type returned by `run_commands` is slightly different (`A` and `B` are lists of the calls made in each parallel branch, paired with the results returned), but otherwise no change to the property is needed.

When this property is tested on a single-core processor, all tests pass. However, as soon as it is tested on a dual-core, tests begin to fail. Interestingly, just running on two cores gives us enough fine-grain interleaving of concurrent processes to demonstrate the presence of race conditions, something we had to achieve by instrumenting the code with calls to `yield()` to control the scheduler when we first tested this code in 2005. However, just as in 2005, the reported failing test cases are large, and do not shrink to small examples. This makes the race condition very hard indeed to diagnose.

The problem is that the test outcome is not determined solely by the test case: depending on the actual interleaving of memory operations on the dual core, the same test may sometimes pass and sometimes fail. This is devastating for QuickCheck's shrinking, which works by repeatedly replacing the failed test case by a smaller one which still fails. If the smaller test happens to succeed—by sheer chance, as a result of non-deterministic execution—then the shrinking process stops. This leads QuickCheck to report failed tests which are far from minimal.

Our solution to this is almost embarrassing in its simplicity: instead of running each test only once, we run it many times, and consider a test case to have passed only if it passes every time we run it. We express this concisely using a new form of QuickCheck property, `?ALWAYS(N,Prop)`, which passes if `Prop` passes `N` times in a row[4]. Now, provided the race condition we are looking for is reasonably likely to be provoked by test cases in which it is present, then `?ALWAYS(10,...)` is very likely to provoke it—and so tests are unlikely to succeed "by chance" during the shrinking process. This dramatically improves the effectiveness of shrinking, even for quite small values of `N`. While we do not *always* obtain minimal failing tests with this approach, we find we can usually obtain a minimal example by running QuickCheck a few times.

When testing the `proc_reg` property above, we find the following simple counterexample:

```
{[{set,{var,5},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,9},{call,proc_reg_eqc,kill,[{var,5}]}},
  {set,{var,15},{call,proc_reg_eqc,reg,[a,{var,5}]}}],
 {[{set,{var,19},{call,proc_reg_eqc,reg,[a,{var,5}]}}],
  [{set,{var,18},{call,proc_reg_eqc,reg,[a,{var,5}]}}]}}
```

This test case first creates and kills a process, then tries to register it (which should have no effect, because it is already dead), and finally tries to register it again twice, in parallel. Printing the diagnostic output from `run_commands`, we see:

```
Sequential:
  [{{state,[],[],[]},<0.5576.2>},
   {{state,[<0.5576.2>],[],[]},ok},
   {{state,[<0.5576.2>],[],[<0.5576.2>]},true}]
Parallel:
  {[{{call,proc_reg_eqc,reg,[a,<0.5576.2>]},
     {'EXIT',{badarg,[{proc_reg,reg,2},...]}}}],
   [{{call,proc_reg_eqc,reg,[a,<0.5576.2>]},true}]}
Res: no_possible_interleaving
```

(where the ellipses replace an uninteresting stack trace). The values displayed under "`Parallel:`" are the results `A` and `B` from the two parallel branches—they reveal that one of the parallel calls to `reg` raised an exception, even though trying to register a dead process should always just return `true`! How this happened, though, is still quite mysterious—but will be explained in the following sections.

[4] In fact we need only repeat tests during shrinking.

5. PULSE: A User-level Scheduler

At this point, we have found a simple test case that fails, but we do not know *why* it failed—we need to debug it. A natural next step would be to turn on Erlang's tracing features and rerun the test. But when the bug is caused by a race condition, then turning on tracing is likely to change the timing properties of the code, and thus interfere with the test failure! Even simply repeating the test may lead to a different result, because of the non-determinism inherent in running on a multi-core. This is devastating for debugging.

What we need is to be able to *repeat* the test as many times as we like, with deterministic results, and to *observe* what happens during the test, so that we can analyze how the race condition was provoked. With this in mind, we have implemented a new Erlang module that can control the execution of designated Erlang processes and records a trace of all relevant events. Our module can be thought of as a *user-level scheduler*, sitting on top of the normal Erlang scheduler. Its aim is to take control over all sources of non-determinism in Erlang programs, and instead take those scheduling decisions randomly. This means that we can repeat a test using exactly the same schedule by supplying the same random number seed: this makes tests repeatable. We have named the module PULSE, short for *ProTest* User-Level Scheduler for Erlang.

The Erlang virtual machine (VM) runs processes for relatively long time-slices, in order to minimize the time spent on context switching—but as a result, it is very unlikely to provoke race conditions in small tests. It is possible to tune the VM to perform more context switches, but even then the scheduling decisions are entierly deterministic. This is one reason why tricky concurrency bugs are rarely found during unit testing; it is not until later stages of a project, when many components are tested together, that the standard scheduler begins to preempt processes and trigger race conditions. In the worst case, bugs don't appear until the system is put under heavy load in production! In these later stages, such errors are expensive to debug. One other advantage (apart from repeatability) of PULSE is that it generates much more fine-grain interleaving than the built-in scheduler in the Erlang virtual machine (VM), because it randomly chooses the next process to run at each point. Therefore, we can provoke race conditions even in very small tests.

Erlang's scheduler is built into its virtual machine—and we did *not* want to modify the virtual machine itself. Not only would this be difficult—it is a low-level, fairly large and complex C program—but we would need to repeat the modifications every time a new version of the virtual machine was released. We decided, therefore, to implement PULSE in Erlang, as a user-level scheduler, and to *instrument* the code of the processes that it controls so that they cooperate with it. As a consequence, PULSE can even be used in conjunction with legacy or customized versions of the Erlang VM (which are used in some products). The user level scheduler also allows us to restrict our debugging effort to a few processes, whereas we are guaranteed that the rest of the processes are executed normally.

5.1 Overall Design

The central idea behind developing PULSE was to provide absolute control over the order of relevant events. The first natural question that arises is: What are the relevant events? We define a *side-effect* to be any interaction of a process with its environment. Of particular interest in Erlang is the way processes interact by message passing, which is asynchronous. Message channels, containing messages that have been sent but not yet delivered, are thus part of the environment and explicitly modelled as such in PULSE. It makes sense to separate side-effects into two kinds: *outward* side-effects, that influence only the environment (such as sending a message over a channel, which does not block and cannot fail, or printing a message), and *inward* side-effects, that allow the environment to influence the behavior of the process (such as receiving a message from a channel, or asking for the system time).

We do not want to take control over purely functional code, or side-effecting code that only influences processes locally. PULSE takes control over some basic features of the Erlang RTS (such as spawning processes, message sending, linking, etc.), but it knows very little about standard library functions – it would be too much work to deal with each of these separately! Therefore, the user of PULSE can specify which library functions should be dealt with as (inward) side-effecting functions, and PULSE has a generic way of dealing with these (see subsection 5.3).

A process is only under the control of PULSE if its code has been properly instrumented. All other processes run as normal. In instrumentation, occurrences of side-effecting actions are replaced by indirections that communicate with PULSE instead. In particular, outward side-effects (such as sending a message to another process) are replaced by simply sending a message to PULSE with the details of the side-effect, and inward side-effects (such as receiving a message) are replaced by sending a request to PULSE for performing that side-effect, and subsequently waiting for permission. To ease the instrumentation process, we provide an automatic instrumenter, described in subsection 5.4.

5.2 Inner Workings

The PULSE scheduler controls its processes by allowing only one of them to run at a time. It employs a cooperative scheduling method: At each decision point, PULSE randomly picks one of its waiting processes to proceed, and wakes it up. The process may now perform a number of outward side-effects, which are all recorded and taken care of by PULSE, until the process wants to perform an inward side-effect. At this point, the process is put back into the set of waiting processes, and a new decision point is reached.

The (multi-node) Erlang semantics (Svensson and Fredlund 2007) provides only one guarantee for message delivery order: that messages between a **pair** of processes arrive in the same order as they were sent. So as to adhere to this, PULSE's state also maintains a message queue between each pair of processes. When process P performs an outward side-effect by sending a message M to the process Q, then M is added to the queue $\langle P, Q\rangle$. When PULSE wants to wake up a waiting process Q, it does so by randomly picking a non-empty queue $\langle P', Q\rangle$ with Q as its destination, and delivering the first message in that queue to Q. Special care needs to be taken for the Erlang construct `receive` ... `after` n `->` ... `end`, which allows a receiving process to only wait for an incoming message for n milliseconds before continuing, but the details of this are beyond the scope of this paper.

As an additional benefit, this design allows PULSE to detect deadlocks when it sees that all processes are blocked, and there exist no message queues with the blocked processes as destination.

As a clarification, the message queues maintained by PULSE for each pair of processes should not be confused with the internal mailbox that each process in Erlang has. In our model, sending a message M from P to Q goes in four steps: (1) P asynchronously sends off M, (2) M is on its way to Q, (3) M is delivered to Q's mailbox, (4) Q performs a `receive` statement and M is selected and removed from the mailbox. The only two events in this process that we consider side-effects are (1) P sending of M, and (3) delivering M to Q's mailbox. In what order a process decides to process the messages in its mailbox is not considered a side-effect, because no interaction with the environment takes place.

5.3 External Side-effects

In addition to sending and receiving messages between themselves, the processes under test can also interact with uninstrumented code. PULSE needs to be able to control the order in which those interactions take place. Since we are not interested in controlling the order in which pure functions are called we allow the programmer to specify which external functions have side-effects. Each call of a side-effecting function is then instrumented with code that `yields` before performing the real call and PULSE is free to run another process at that point.

Side-effecting functions are treated as atomic which is also an important feature that aids in testing systems built of multiple components. Once we establish that a component contains no race conditions we can remove the instrumentation from it and mark its operations as atomic side-effects. We will then be able to test other components that use it and each operation marked as side-effecting will show up as a single event in a trace. Therefore, it is possible to test a component for race conditions independently of the components that it relies on.

5.4 Instrumentation

The program under test has to cooperate with PULSE, and the relevant processes should use PULSE's API to send and receive messages, spawn processes, etc., instead of Erlang's built-in functionality. Manually altering an Erlang program so that it does this is tedious and error-prone, so we developed an instrumenting compiler that does this automatically. The instrumenter is used in exactly the same way as the normal compiler, which makes it easy to switch between PULSE and the normal Erlang scheduler. It's possible to instrument and load a module at runtime by typing in a single command at the Erlang shell.

Let us show the instrumentation of the four most important constructs: sending a message, yielding, spawning a process, and receiving a message.

5.4.1 Sending

If a process wants to send a message, the instrumenter will redirect this as a request to the PULSE scheduler. Thus, `Pid ! Msg` is replaced by

```
scheduler ! {send, Pid, Msg},
Msg
```

The result value of sending a message is always the message that was sent. Since we want the instrumented *send* to yield the same result value as the original one, we add the second line.

5.4.2 Yielding

A process yields when it wants to give up control to the scheduler. Yields are also introduced just before each user-specified side-effecting external function.

After instrumentation, a yielding process will instead give up control to PULSE. This is done by telling it that the process yields, and waiting for permission to continue. Thus, `yield()` is replaced by

```
scheduler ! yield,
receive
  {scheduler, go} -> ok
end
```

In other words, the process notifies PULSE and then waits for the message go from the scheduler before it continues. All control messages sent by PULSE to the controlled processes are tagged with `{scheduler, _}` in order to avoid mixing them up with "real" messages.

5.4.3 Spawning

A process P spawning a process Q is considered an outward side-effect for P, and thus P does not have to block. However, PULSE must be informed of the existence of the new process Q, and Q

needs to be brought under its control. The spawned process Q must therefore wait for PULSE to allow it to run. Thus, `spawn(Fun)` is replaced by

```
Pid = spawn(fun() -> receive
                         {scheduler, go} -> Fun()
                     end
          end),
scheduler ! {spawned, Pid},
Pid
```

In other words, the process spawns an altered process that waits for the message go from the scheduler before it does anything. The scheduler is then informed of the existence of the spawned process, and we continue.

5.4.4 Receiving

Receiving in Erlang works by pattern matching on the messages in the process' mailbox. When a process is ready to receive a new message, it will have to ask PULSE for permission. However, it is possible that an appropriate message already exists in its mailbox, and receiving this message would not be a side-effect. Therefore, an instrumented process will first check if it is possible to receive a message with the desired pattern, and proceed if this is possible. If not, it will tell the scheduler that it expects a new message in its mailbox, and blocks. When woken up again on the delivery of a new message, this whole process is repeated if necessary.

We need a helper function that implements this checking-waiting-checking loop. It is called `receiving`:

```
receiving(Receiver) ->
  Receiver(fun() ->
    scheduler ! block,
    receive
      {scheduler, go} -> receiving(Receiver)
    end
  end).
```

`receiving` gets a receiver function as an argument. A receiver function is a function that checks if a certain message is in its mailbox, and if not, executes its argument function. The function `receiving` turns this into a loop that only terminates once PULSE has delivered the right message. When the receiver function fails, PULSE is notified by the `block` message, and the process waits for permission to try again.

Code of the form

```
receive Pat -> Exp end
```

is then replaced by

```
receiving(fun (Failed) ->
            receive
              Pat     -> Exp
              after 0 -> Failed()
            end
          end)
```

In the above, we use the standard Erlang idiom (`receive ... after 0 -> ... end`) for checking if a message of a certain type exists. It is easy to see how receive statements with more than one pattern can be adapted to work with the above scheme.

5.5 Testing `proc_reg` with PULSE

To test the `proc_reg` module using both QuickCheck and PULSE, we need to make a few modifications to the QuickCheck property in Section 4.4.

```
prop_proc_reg_scheduled() ->
  ?FORALL(Cmds,eqc_par_statem:commands(?MODULE),
   ?ALWAYS(10,?FORALL(Seed,seed(),
     begin
      SRes =
        scheduler:start([{seed,Seed}],
          fun() ->
            {ok,ETSTabs} = proc_reg_tabs:start_link(),
            {ok,Server} = proc_reg:start_link(),
            eqc_par_statem:run_commands(?MODULE,Cmds),
            cleanup(ETSTabs,Server),
          end),
      {H,AB,Res} = scheduler:get_result(SRes),
      Res == ok
    end))).
```

PULSE uses a random seed, generated by `seed()`. It also takes a function as an argument, so we create a lambda-function which initializes and runs the tests. The result of running the scheduler is a list of things, thus we need to call `scheduler:get_result` to retrieve the actual result from `run_commands`. We should also remember to *instrument* rather than compile all the involved modules. Note that we still use `?ALWAYS` in order to run the same test data with different random seeds, which helps the shrinking process in finding smaller failing test cases that would otherwise be less likely to fail.

When testing this modified property, we find the following counterexample, which is in fact simpler than the one we found in Section 4.4:

```
{[{set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,10},{call,proc_reg_eqc,kill,[{var,9}]}}],
 {[{set,{var,15},{call,proc_reg_eqc,reg,[c,{var,9}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[c,{var,9}]}}]}}
```

When prompted, PULSE provides quite a lot of information about the test case run and the scheduling decisions taken. Below we show an example of such information. However, it is still not easy to explain the counterexample; in the next section we present a method that makes it easier to understand the scheduler output.

```
 -> <'start_link.Pid1'> calls
    scheduler:process_flag [priority,high]
    returning normal.
 -> <'start_link.Pid1'> sends
    '{call,{attach,<0.31626.0>},
          <0.31626.0>,#Ref<0.0.0.13087>}'
    to <'start_link.Pid'>.
 -> <'start_link.Pid1'> blocks.
*** unblocking <'start_link.Pid'>
    by delivering '{call,{attach,<0.31626.0>},
                         <0.31626.0>,
                         #Ref<0.0.0.13087>}'
    sent by <'start_link.Pid1'>.
...
```

6. Visualizing Traces

PULSE records a complete trace of the interesting events during test execution, but these traces are long, and tedious to understand. To help us interpret them, we have, utilizing the popular GraphViz package (Gansner and North 1999), built a *trace visualizer* that draws the trace as a graph. For example, Figure 1 shows the graph drawn for one possible trace of the following program:

```
procA() ->
  PidB = spawn(fun procB/0),
  PidB ! a,
  process_flag(trap_exit, true),
  link(PidB),
  receive
    {'EXIT',_,Why} -> Why
  end.

procB() ->
  receive
    a -> exit(kill)
  end.
```

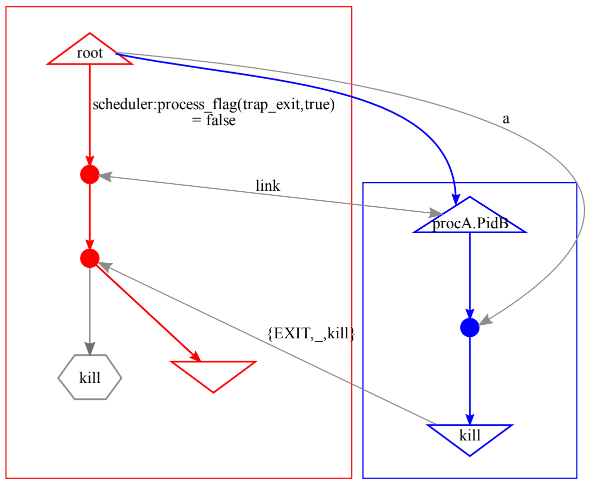

Figure 1. A simple trace visualization.

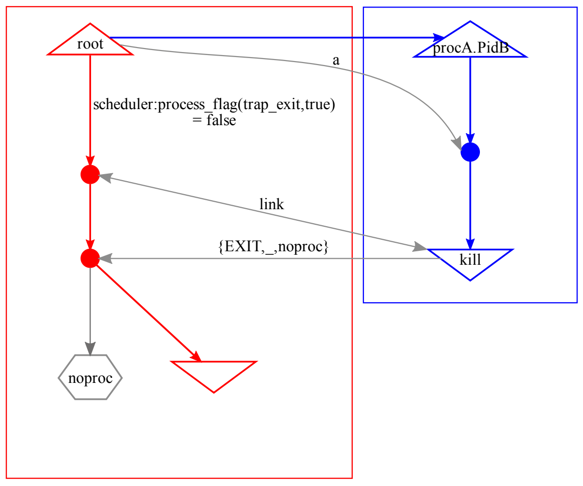

Figure 2. An alternative possible execution.

The function `procA` starts by spawning a process, and subsequently sends it a message `a`. Later, `procA` *links* to the process it spawned, which means that it will get notified when that process dies. The default behavior of a process when such a notification happens is to also die (in this way, one can build hierarchies of processes). Setting the process flag *trap_exit* to true changes this behaviour, and the notification is delivered as a regular message of the form `{EXIT,_,_}` instead.

In the figure, each process is drawn as a sequence of state transitions, from a start state drawn as a triangle, to a final state drawn as an inverted triangle, all enclosed in a box and assigned a unique color. (Since the printed version of the diagrams may lack these colors, we reference diagram details by location and not by color. However, the diagrams are even more clear in color.) The diagram shows the two processes, `procA` (called `root`) which is shown to the left (in red), and `procB` (called `procA.PidB`, a name automatically derived by PULSE from the point at which it was spawned) shown to the right (in blue). Message delivery is shown by gray arrows, as is the return of a result by the `root` process. As explained in the previous section, processes make transitions when receiving a message[5], or when calling a function that the instrumenter knows has a side-effect. From the figure, we can see that the `root` process spawned `PidB` and sent the message `a` to it, but before the message was delivered then the `root` process managed to set its `trap_exit` process flag, and linked to `PidB`. `PidB` then received its message, and killed itself, terminating with reason `kill`. A message was sent back to `root`, which then returned the exit reason as its result.

Figure 2 shows an alternative trace, in which `PidB` dies *before* `root` creates a link to it, which generates an exit message with a different exit reason. The existence of these two different traces indicates a race condition when using `spawn` and `link` separately (which is the reason for the existence of an atomic `spawn_link` function in Erlang).

The diagrams help us to understand traces by gathering together all the events that affect one process into one box; in the original traces, these events may be scattered throughout the entire trace. But notice that the diagrams also *abstract away* from irrelevant information—specifically, the order in which messages are delivered to *different* processes, which is insignificant in Erlang. This abstraction is one strong reason why the diagrams are easier to understand than the traces they are generated from.

[5] If messages are consumed from a process mailbox out-of-order, then we show the delivery of a message to the mailbox, and its later consumption, as separate transitions.

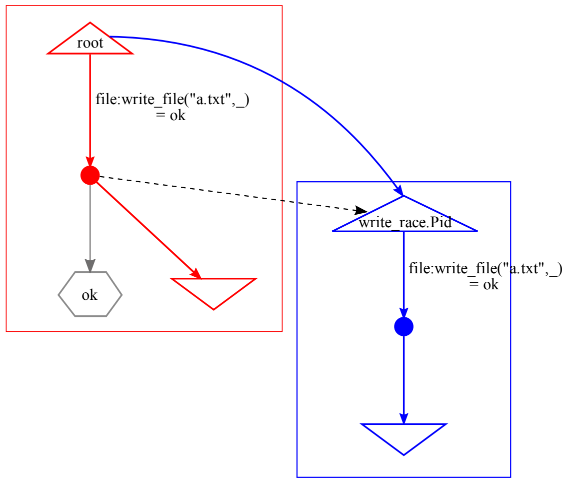

Figure 3. A race between two side-effects.

However, we *do* need to know the order in which calls to functions with side-effects occur, even if they are made in different processes. To make this order visible, we add dotted black arrows to our diagrams, from one side-effecting call to the next. Figure 3 illustrates one possible execution of this program, in which two processes race to write to the same file:

```
write_race() ->
  Pid = spawn(fun() ->
                file:write_file("a.txt","a")
              end),
  file:write_file("a.txt","b").
```

In this diagram, we can see that the `write_file` in the `root` process preceded the one in the spawned process `write_race.Pid`.

If we draw these arrows between *every* side-effect and its successor, then our diagrams rapidly become very cluttered. However, it is only necessary to indicate the sequencing of side-effects explicitly *if their sequence is not already determined.* For each pair of successive side-effect transitions, we thus compute Lamport's "happens before" relation (Lamport 1978) between them, and if this already implies that the first precedes the second, then we draw no arrow in the diagram. Interestingly, in our examples then this eliminates the majority of such arrows, and those that remain tend to surround possible race conditions—where the message passing (synchronization) does not enforce a particular order of side-effects. Thus black dotted arrows are often a sign of trouble.

6.1 Analyzing the `proc_reg` race conditions

Interestingly, as we saw in Section 5.5, when we instrumented `proc_reg` and tested it using PULSE and QuickCheck, we obtained a different—even simpler—minimal failing test case, than the one we had previously discovered using QuickCheck with the built-in Erlang scheduler. Since we need to use PULSE in order to obtain a trace to analyze, then we must fix this bug first, and see whether that also fixes the first problem we discovered. The failing test we find using PULSE is this one:

```
{[{set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,10},{call,proc_reg_eqc,kill,[{var,9}]}}],
 {[{set,{var,15},{call,proc_reg_eqc,reg,[c,{var,9}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[c,{var,9}]}}]}}
```

In this test case, we simply create a dead process (by spawning a process and then immediately killing it), and try to register it twice in parallel, and as it happens the first call to `reg` raises an exception. The diagram we generate is too large to include in full, but in Figure 4 we reproduce the part showing the problem.

In this diagram fragment, the processes are, from left to right, the `proc_reg` server, the second parallel fork (`BPid`), and the first parallel fork (`APid`). We can see that `BPid` first inserted its argument into the ETS table, recording that the name `c` is now taken, then sent an asynchronous message to the server (`{cast,{..}}`) to inform it of the new entry. Thereafter `APid` tried to insert an ETS entry with the same name—but failed. After discovering that the process being registered is actually dead, `APid` sent a message to the server asking it to "audit" its entry (`{call,{..},_,_}`)—that is, clean up the table by deleting the entry for a dead process. *But this message was delivered before the message from* `BPid`*!* As a result, the server could not find the dead process in its table, and failed to delete the entry created by `BPid`, leading `APid`'s second attempt to create an ETS entry to fail also—which is not expected to happen. When `BPid`'s message is finally received and processed by the server, it is already too late.

The problem arises because, while the clients create "forward" ETS entries linking the registered name to a pid, it is the server which creates a "reverse" entry linking the pid to its monitoring reference (created by the server). It is this reverse entry that is used by the server when asked to remove a dead process from its tables. We corrected the bug by letting clients (atomically) insert *two* ETS entries into the same table: the usual forward entry, and a dummy reverse entry (lacking a monitoring reference) that is later overwritten by the server. This dummy reverse entry enables the server to find and delete both entries in the test case above, thus solving the problem.

In fact, the current Erlang virtual machine happens to deliver messages to local mailboxes instantaneously, which means that one message cannot actually overtake another message sent earlier—the cause of the problem in this case. This is why this minimal failing test was not discovered when we ran tests on a multi-core, using the built-in scheduler. However, this behavior is not guaranteed by the language definition, and indeed, messages between nodes in a distributed system *can* overtake each other in this way. It is expected that future versions of the virtual machine may allow message overtaking even on a single "many-core" processor; thus we consider it an advantage that our scheduler allows this behavior, and can provoke race conditions that it causes.

It should be noted that exactly the same scenario can be triggered in an alternative way (without parallel processes and multi-core!); namely if the `BPid` above is preempted between its call to `ets:insert_new` and sending the `cast`-message. However, the likelihood for this is almost negligible, since the Erlang scheduler prefers running processes for relatively long time-slices. Using PULSE does not help triggering the scenario in this way either. PULSE is not in control at any point between `ets:insert_new` and sending the `cast`-message, meaning that only the Erlang scheduler controls the execution. Therefore, the only feasible way to repeatedly trigger this faulty scenario is by delaying the `cast`-message by using PULSE (or a similar tool).

6.2 A second race condition in `proc_reg`

Having corrected the bug in `proc_reg` we repeated the QuickCheck test. The property still fails, with the same minimal failing case that we first discovered (which is not so surprising since the problem that we fixed in the previous section cannot actually occur with today's VM). However, we were now able to reproduce the failure with PULSE, as well as the built-in scheduler. As a result, we could now analyze and debug the race condition. The failing case is:

```
{[{set,{var,4},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,7},{call,proc_reg_eqc,kill,[{var,4}]}},
  {set,{var,12},{call,proc_reg_eqc,reg,[b,{var,4}]}}],
 {[{set,{var,18},{call,proc_reg_eqc,reg,[b,{var,4}]}}],
  [{set,{var,21},{call,proc_reg_eqc,reg,[b,{var,4}]}}]}}
```

In this test case we also create a dead process, but we try to register it once in the sequential prefix, before trying to register it twice in parallel. Once again, one of the calls to `reg` in the parallel branches raised an exception.

Turning again to the generated diagram, which is not included in the paper for space reasons, we observed that both parallel branches (`APid` and `BPid`) fail to insert `b` into the ETS table. They fail since the name `b` was already registered in the sequential part of the test case, and the server has not yet processed the `DOWN` message generated by the monitor. Both processes then call `where(b)` to see if `b` is *really* registered, which returns `undefined` since the process is dead. Both `APid` and `BPid` then request an "audit" by the server, to clean out the dead process. After the audit, both processes assume that it is now ok to register `b`, there is a race condition between the two processes, and one of the registrations fails. Since this is not expected, an exception is raised. (Note that if `b` were alive then this would be a perfectly valid race condition, where one of the two processes successfully registers the name and the other fails, but the specification says that the registration should always return `true` for dead processes).

This far into our analysis of the error it became clear that it is an altogether rather unwise idea ever to insert a dead process into the process registry. To fix the error we added a simple check (`is_process_alive(Pid)`) before inserting into the registry. The effect of this change on the performance turned out to be negligible, because `is_process_alive` is very efficient for local processes. After this change the module passed 20 000 tests, and we were satisfied.

7. Discussion and Related Work

Actually, the "fix" just described does not really remove all possible race conditions. Since the diagrams made us understand the algorithm much better, we can spot another possible race condition: If `APid` and `BPid` try to register the same pid at the same time, and that process dies *just after* `APid` and `BPid` have checked that it is alive, then the same problem we have just fixed, will arise. The rea-

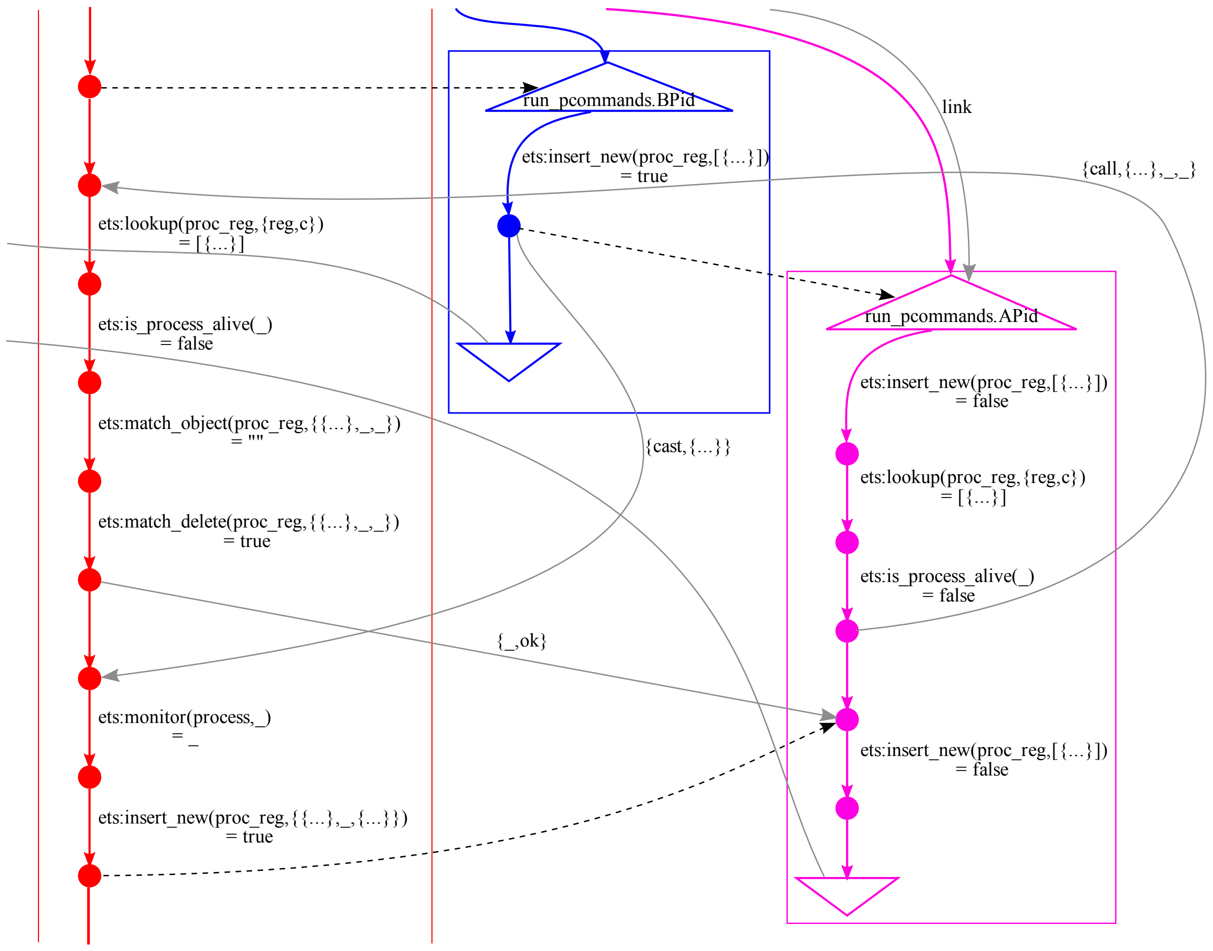
run_pcommands.BPid
link
ets:insert_new(proc_reg,[{...}]) = true
{call,{...},_,_}
ets:lookup(proc_reg,{reg,c}) = [{...}]
run_pcommands.APid
ets:is_process_alive(_) = false
ets:insert_new(proc_reg,[{...}]) = false
ets:match_object(proc_reg,{{...},_,_}) = ""
{cast,{...}}
ets:lookup(proc_reg,{reg,c}) = [{...}]
ets:match_delete(proc_reg,{{...},_,_}) = true
ets:is_process_alive(_) = false
{_,ok}
ets:monitor(process,_) = _
ets:insert_new(proc_reg,[{...}]) = false
ets:insert_new(proc_reg,{{...},_,{...}}) = true

Figure 4. A problem caused by message overtaking.

son that our tests succeeded even so, is that a test must contain *three* parallel branches to provoke the race condition in its new form—two processes making simultaneous attempts to register, and a third process to kill the pid concerned at the right moment. Because our parallel test cases only run *two* concurrent branches, then they can never provoke this behavior.

The best way to fix the last race condition problem in `proc_reg` would seem to be to simplify its API, by restricting `reg` so that a process may only register itself. This, at a stroke, eliminates the risk of two processes trying to register the same process at the same time, and guarantees that we can never try to register a dead process. This simplification was actually made in the production version of the code.

Parallelism in test cases

We could, of course, generate test cases with three, four, or even more concurrent branches, to test for this kind of race condition too. The problem is, as we explained in section 4.2, that the number of possible interleavings grows extremely fast with the number of parallel branches. The number of interleavings of K sequences of length N are as presented in Figure 5.

The practical consequence is that, if we allow more parallel branches in test cases, then we must restrict the length of each branch correspondingly. The bold entries in the table show the last “feasible” entry in each column—with three parallel branches, we would need to restrict each branch to just three commands; with

		K			
		2	3	4	5
	1	2	6	24	**120**
	2	6	90	**2520**	113400
	3	20	**1680**	369600	10^8
N	4	70	34650	6×10^7	3×10^{11}
	5	252	756756	10^{10}	6×10^{14}
	…		…		
	8	**12870**	10^{10}	10^{17}	8×10^{24}

Figure 5. Possible interleavings of parallel branches

four branches, we could only allow two; with five or more branches, we could allow only one command per branch. This is in itself a restriction that will make some race conditions impossible to detect. Moreover, with more parallel branches, there will be even more possible schedules for PULSE to explore, so race conditions depending on a precise schedule will be correspondingly harder to find.

There is thus an engineering trade-off to be made here: allowing greater parallelism in test cases may in theory allow more race conditions to be discovered, but in practice may reduce the probability of finding a bug with each test, while at the same time increasing the cost of each test. We decided to prioritize longer sequences over more parallelism in the test case, and so we chose $K = 2$. However, in the future we plan to experiment with letting QuickCheck randomly choose K and N from the set of feasible combinations. To be clear, note that K only refers to the parallelism in the test case, that is, the number of processes that make calls to the API. The system under test may have hundreds of processes running, many of them controlled by PULSE, independently of K.

The problem of detecting race conditions is well studied and can be divided in runtime detection, also referred to as *dynamic detection*, and analyzing the source code, so called *static detection*. Most results refer to race conditions in which two threads or processes write to shared memory (data race condition), which in Erlang cannot happen. For us, a race condition appears if there are two schedules of occurring side effects (sending a message, writing to a file, trapping exits, linking to a process, etc) such that in one schedule our model of the system is violated and in the other schedule it is not. Of course, writing to a shared ETS table and writing in shared memory is related, but in our example it is allowed that two processes call ETS insert in parallel. By the atomicity of insert, one will succeed, the other will fail. Thus, there is a valid race condition that we do not want to detect, since it does not lead to a failure. Even in this slightly different setting, known results on race conditions still indicate that we are dealing with a hard problem. For example, Netzer and Miller (1990) show for a number of relations on traces of events that ordering these events on 'could have been a valid execution' is an NP-hard problem (for a shared memory model). Klein et al. (2003) show that statically detecting race conditions is NP-complete if more than one semaphore is used.

Thus, restricting `eqc_par_statem` to execute only two processes in parallel is a pragmatic choice. Three processes may be feasible, but real scalability is not in sight. This pragmatic choice is also supported by recent studies (Lu et al. 2008), where it is concluded that: *"Almost all (96%) of the examined concurrency bugs are guaranteed to manifest if certain partial order between 2 threads is enforced."*

Hierarchical approach

Note that our tools support a *hierarchical* approach to testing larger systems. We test `proc_reg` *under the assumption that the underlying ets operations are atomic*; PULSE does not attempt to (indeed, cannot) interleave the executions of single ETS operations, which are implemented by C code in the virtual machine. Once we have established that the `proc_reg` operations behave atomically, then *we can make the same assumption* about them when testing code that makes use of them. When testing for race conditions in modules that use `proc_reg`, then we need not, and do not want to, test for race conditions in `proc_reg` itself. As a result, the PULSE schedules remain short, and the simple random scheduling that we use suffices to find schedules that cause failures.

Model Checking

One could argue that the optimal solution to finding race conditions problem would be to use a model checker to explore all possible interleavings. The usual objections are nevertheless valid, and the rapidly growing state space for concurrent systems makes model checking totally infeasible, even with a model checker optimized for Erlang programs, such as McErlang (Fredlund and Svensson 2007). Further it is not obvious what would be the property to model check, since the atomicity violations that we search for can not be directly translated into an LTL model checking property.

Input non-determinism

PULSE provides deterministic scheduling. However, in order for tests to be repeatable we also need the external functions to behave consistently across repeated runs. While marking them as side-effects will ensure that they are only called serially, PULSE does nothing to guarantee that functions called in the same sequence will return the same values in different runs. The user still has to make sure that the state of the system is reset properly before each run. Note that the same arguments apply to QuickCheck testing; it is crucial for shrinking and re-testing that input is deterministic and thus it works well to combine QuickCheck and PULSE.

False positives

In contrast to many race finding methods, that try to spot common patterns leading to concurrency bugs, our approach does not produce false positives and not even does it show races that result in correct execution of the program. This is because we employ *property-based testing* and classify test cases based on whether the results satisfy correctness properties and report a bug only when a property is violated.

Related tools

Park and Sen (2008) study atomicity in Java. Their approach is similar to ours in that they use a random scheduler both for repeatability and increased probability of finding atomicity violations. However, since Java communication is done with shared objects and locks, the analysis is rather different.

It is quite surprising that our simple randomized scheduler—and even just running tests on a multi-core—coupled with repetition of tests to reduce non-determinism, should work so well for us. After all, this can only work if the probability of provoking the race condition in each test that contains one is reasonably high. In contrast, race conditions are often regarded as very *hard* to provoke because they occur so rarely. For example, Sen used very carefully constructed schedules to provoke race conditions in Java programs (Sen 2008)—so how can we reasonably expect to find them just by running the same test a few times on a multi-core?

We believe two factors make our simple approach to race detection feasible.

- Firstly, Erlang is not Java. While there *is* shared data in Erlang programs, there is much less of it than in a concurrent Java program. Thus there are many fewer potential race conditions, and a simpler approach suffices to find them.
- Secondly, we are searching for race conditions during *unit testing*, where each test runs for a short time using only a relatively small amount of code. During such short runs, there is a fair chance of provoking race conditions with any schedule. Finding race conditions during whole-program testing is a much harder problem.

Chess, developed by Musuvathi et al. (2008), is a system that shares many similarities with PULSE. Its main component is a scheduler capable of running the program deterministically and replaying schedules. The key difference between Chess and PULSE is that the former attempts to do an exhaustive search and enumerate all the possible schedules instead of randomly probing them. Several interesting techniques are employed, including prioritizing schedules that are more likely to trigger bugs, making sure that only fair schedules are enumerated and avoiding exercising schedules that differ insignificantly from already visited ones.

Visualization

Visualization is a common technique used to aid understanding software. Information is extracted statically from source code or dynamically from execution and displayed in graphical form. Of many software visualization tools a number are related to our work. Topol et al. (1995) developed a tool that visualizes executions of parallel programs and shows, among other things, a trace of messages sent between processes indicating the happened-before relationships. Work of Jerding et al. (1997) is able to show dynamic call-graphs of object-oriented programs and interaction patterns between their components. Arts and Fredlund (2002) describe a tool that visualizes traces of Erlang programs in form of abstract state transition diagrams. Artho et al. (2007) develop a notation that extends UML diagrams to also show traces of concurrent executions of threads, Maoz et al. (2007) create event sequence charts that can express which events "must happen" in all possible scenarios.

8. Conclusions

Concurrent code is hard to debug and therefore hard to get correct. In this paper we present an extension to QuickCheck, a user level scheduler for Erlang (PULSE), and a tool for visualizing concurrent executions that together help in debugging concurrent programs. The tools allow us to find concurrency errors on a module testing level, whereas industrial experience is that most of them slip through to system level testing, because the standard scheduler is deterministic, but behaves differently in different timing contexts.

We contributed `eqc_par_statem`, an extension of the state machine library for QuickCheck that enables parallel execution of a sequence of commands. We generate a sequential prefix to bring the system into a certain state and continue with parallel execution of a suffix of independent commands. As a result we can provoke concurrency errors and at the same time get good shrinking behavior from the test cases.

We contributed with PULSE, a user level scheduler that enables scheduling of any concurrent Erlang program in such a way that an execution can be repeated deterministically. By randomly choosing different schedules, we are able to explore more execution paths than without such a scheduler. In combination with QuickCheck we get in addition an even better shrinking behavior, because of the repeatability of test cases.

We contributed with a graph visualization method and tool that enabled us to analyze concurrency faults more easily than when we had to stare at the produced traces. The visualization tool depends on the output produced by PULSE, but the use of computing the "happens before" relation to simplify the graph is a general principle.

We evaluated the tools on a real industrial case study and we detected two race conditions. The first one by only using `eqc_par_statem`; the fault had been noticed before, but now we did not need to instrument the code under test with `yield()` commands. The first and second race condition could easily be provoked by using PULSE. The traces recorded by PULSE were visualized and helped us in clearly identifying the sources of the two race conditions. By analyzing the graphs we could even identify a third possible race condition, which we could provoke if we allowed three instead of two parallel processes in `eqc_par_statem`.

Our contributions help Erlang software developers to get their concurrent code right and enables them to ship technologically more advanced solutions. Products that otherwise might have remained a prototype, because they were neither fully understood nor tested enough, can now make it into production. The tool PULSE and the visualization tool are available under the Simplified BSD License and have a commercially supported version as part of Quviq QuickCheck.

Acknowledgments

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868.

References

Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, July 2007.

Cyrille Artho, Klaus Havelund, and Shinichi Honiden. Visualization of concurrent program executions. In *COMPSAC '07: Proc. of the 31st Annual International Computer Software and Applications Conference*, pages 541–546, Washington, DC, USA, 2007. IEEE Computer Society.

Thomas Arts and Lars-Åke Fredlund. Trace analysis of Erlang programs. *SIGPLAN Notices*, 37(12):18–24, 2002.

Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In *ERLANG '06: Proc. of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, 2006.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proc. of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

Mats Cronqvist. Troubleshooting a large Erlang system. In *ERLANG '04: Proc. of the 2004 ACM SIGPLAN workshop on Erlang*, pages 11–15, New York, NY, USA, 2004. ACM.

Lars-Åke Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, 42(9): 125–136, 2007.

Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications. *Software - Practice and Experience*, 30: 1203–1233, 1999.

M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL '87: Proc. of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Prog. Lang.*, pages 13–26, New York, NY, USA, 1987. ACM.

John Hughes. QuickCheck Testing for Fun and Profit. In *9th Int. Symp. on Practical Aspects of Declarative Languages*. Springer, 2007.

Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *In Proc. of the 19th International Conference on Software Engineering*, pages 360–370, 1997.

Klein, Lu, and Netzer. Detecting race conditions in parallel programs that use semaphores. *Algorithmica*, 35:321–345, 2003.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28 (9):690–691, 1979.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Comput. Archit. News*, 36(1):329–339, 2008.

Shahar Maoz, Asaf Kleinbort, and David Harel. Towards trace visualization and exploration for reactive systems. In *VLHCC '07: Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 153–156, Washington, DC, USA, 2007. IEEE Computer Society.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *In Proc. of the 1990 Int. Conf. on Parallel Processing*, pages 93–97, 1990.

Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16: Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145, New York, NY, USA, 2008. ACM.

Koushik Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43(6):11–21, 2008.

H. Svensson and L.-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Erlang '07: Proc. of the 2007 SIGPLAN Erlang Workshop*, pages 43–54, New York, NY, USA, 2007. ACM.

B. Topol, J.T. Stasko, and V. Sunderam. Integrating visualization support into distributed computing systems. *Proc. of the 15th Int. Conf. on: Distributed Computing Systems*, pages 19–26, May-Jun 1995.

Ulf T. Wiger. Extended process registry for Erlang. In *ERLANG '07: Proc. of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 1–10, New York, NY, USA, 2007. ACM.