Systematic Testing for Detecting Concurrency Errors in Erlang Programs

Maria Christakis*, Alkis Gotovos* and Konstantinos Sagonas[†][‡]
* *Department of Computer Science, ETH Zurich, Switzerland*
[†] *Department of Information Technology, Uppsala University, Sweden*
[‡] *School of Electrical and Computer Engineering, National Technical University of Athens, Greece*
maria.christakis@inf.ethz.ch   alkisg@ethz.ch   kostis@it.uu.se

***Abstract*—We present the techniques used in Concuerror, a systematic testing tool able to find and reproduce a wide class of concurrency errors in Erlang programs. We describe how we take advantage of the characteristics of Erlang's actor model of concurrency to selectively instrument the program under test and how we subsequently employ a stateless search strategy to systematically explore the state space of process interleaving sequences triggered by unit tests. To ameliorate the problem of combinatorial explosion, we propose a novel technique for avoiding process blocks and describe how we can effectively combine it with preemption bounding, a heuristic algorithm for reducing the number of explored interleaving sequences. We also briefly discuss issues related to soundness, completeness and effectiveness of techniques used by Concuerror.**

I. Introduction

Concurrent programming is a necessity nowadays but such programs are still notoriously difficult to get right. Specific interleaving sequences that occur only rarely can trigger unexpected errors and result in crashes that are surprising even to expert programmers. This is as true today as it was thirty years ago. The difference is that thirty years ago it was only expert programmers who wrote concurrent programs. Today it is the average programmer. As a result, program crashes due to concurrency errors are not only more likely to surprise more programmers, but are also more likely to occur.

Despite advances in model checking and program verification, testing seems to still be the predominant technique for locating software errors. In fact, many programmers are test conscious and programs often come with extensive test suites, commonly based on some unit testing framework. Unfortunately, this kind of testing is not powerful enough to uncover concurrency errors, let alone give guarantees for their absence. Independently of language, detection of concurrency errors requires systematic exploration of interleaving sequences, which is both time consuming and difficult to get right.

Fortunately, compared to thirty years ago, there have been significant advances both in programming language and testing technology. Some programming languages adopt concurrency models, such as the *actor model*, that restrict certain kinds of concurrency errors to a small subset of the language. Advances in systematic testing include *stateless model checking* tools like CHESS [1], an inspiration for our work, which allows finding important concurrency errors even with a small number of preemptions between processes.

This paper presents the techniques used in Concuerror, a stateless model checking tool, that, given an Erlang program and its (existing) test suite, systematically explores process interleaving and presents detailed interleaving information about errors (such as abnormal process exits, stuck processes and assertion violations) that occur during the execution of these tests. Concuerror is extremely easy to use and can support the test-driven development of Erlang programs [2].

The focus of this paper is on Concuerror's implementation technology. More specifically, we describe how we take advantage of the fact that, in an actor language such as Erlang, the majority of program statements access process-local data and cannot possibly result in concurrency errors. This, in turn, allows for a selective instrumentation of the program that enables Concuerror's scheduling algorithm to effectively explore the space of all interleaving sequences of a test executed against the program and identify concurrency errors that occur during its execution. More importantly, we propose a novel technique called *blocking avoidance* that we effectively combine with *iterative context bounding* [3] to significantly reduce the number of explored interleaving sequences. The techniques used in Concuerror, besides being effective in identifying erroneous process interleaving sequences and giving precise explanations about them, can in principle also verify the absence of the kinds of concurrency errors that the tool detects, when the entire interleaving space has been explored.

Outline. In Section II we briefly present Erlang and an example program on which we will illustrate our approach. Section III gives an overview on the techniques used in Concuerror, and Sections IV and V present them in detail. Section Section VI describes two efficiency improvements whose effectiveness is evaluated in Section VII. We review related work in Section VIII and conclude.

II. Preliminaries

Erlang is an industrially relevant programming language based on the actor model of concurrency. In Erlang, actors are realized by language-level processes that, by default, share no memory and communicate with each other via asynchronous message passing. Erlang processes are very lightweight as they are implemented by the runtime system of the language rather than by OS threads, and typical applications often create several thousands of them. Processes get created using the `spawn` family of functions. A `spawn` call creates a new process P having its own private memory area (stack, heap and mailbox) and returns a *process identifier* (PID) for it. Optionally, P can be `linked` to another process, typically its parent, or `registered` under a specific name in a global table, so that other processes can refer to P using its name instead of its PID when sending messages to it. Messages are sent *asynchronously* using the `!/2` expression, which takes two arguments and is a convenient shorthand for the `send/2` function. A process can then consume messages using selective pattern matching in `receive` expressions[1], which are *blocking* operations in case a process mailbox does not contain a matching message. Of course, blocking the execution of a process until a specific kind of message from another process arrives can lead to stuck processes.

Stuck processes, however, are not the only kinds of concurrency errors that are possible in Erlang. Although the majority of memory that programs access is process-local, the language comes with various built-in functions (BIFs), implemented in C, that manipulate data structures at the level of the virtual machine (VM) which are shared between all processes. Interleaving sequences of calls to these BIFs can lead to data races or result in abnormal process exits. The latter may in turn result in abnormal termination of other processes. Testing for absence of concurrency errors due to unfortunate process interleaving is complicated by the fact that many errors are hard to come across and expose by conventional unit testing. Part of the difficulty lies in that the scheduling of processes is done by the Erlang VM and is mostly deterministic. It is currently based on the notion of *reduction steps*: roughly, each process gets to execute for a certain number of "reductions" (currently $2,000$ function calls) before it has to yield back to its scheduler which then picks another process to execute. (A process also yields if it gets blocked on a `receive`.) Consequently, multiple runs of the same unit test are most likely to exhibit the same behavior with respect to process interleaving as such tests are too small for scheduling non-determinism to take effect.

Running example. A simple Erlang program involving two processes is shown in Figure 1. The `pong/0` function, which is exported and may be called from outside the `ping_pong` module, spawns a process that will execute the code of function `ping/1` (line 6). The spawned process, which is registered under the same name as the module (line 6), sends a `ping` message to the parent process (line 10), which, in turn, is expected to receive this message and return `ok` (line 7). This code has a concurrency error. Its execution will raise a runtime exception if the spawned process terminates before the parent process attempts to register its PID, which would not exist after the process terminates. As a result of this exception, the process executing function `pong/0` will crash and exit abnormally. This error is so subtle that many Erlang programmers are not even aware of its possibility. Still, such errors compromise the robustness of applications.

```
1  -module(ping_pong).
2  -export([pong/0]).
3
4  pong() ->
5    Self = self(),
6    register(ping_pong, spawn(fun () -> ping(Self) end)),
7    receive ping -> ok end.
8
9  ping(PongPID) ->
10   PongPID ! ping.
```

Figure 1: Simple example program involving two processes and a concurrency error.

The core of the problem here is that the sequence of calls that `spawns` the new process and `registers` its PID needs to run atomically. In the current implementation of Erlang/OTP, the probability of the parent process being scheduled out between these two consecutive calls is small[2]. However, even if the calls were further apart, which would increase the likelihood that the first process is scheduled out somewhere in between, the error cannot easily be provoked with unit testing because the scheduler of Erlang/OTP is deterministic. To expose it, one would have to abandon unit for system testing or employ a randomized scheduler like PULSE [4]. In any case, both styles of testing would have to rely on luck to provoke and reproduce the error. In contrast, Concuerror is able to find it immediately with systematic testing.

III. Concuerror in a Nutshell

To detect concurrency-related runtime errors such as the above abnormal process exit, Concuerror, given a program and its test suite, systematically explores process interleaving and presents detailed interleaving information on any errors that occur during the execution of the tests. In addition to abnormal process exits, Concuerror detects assertion violations and stuck processes.

Approach. To detect these kinds of errors, Concuerror effectively explores all interleaving sequences of the processes that participate in a test execution using a *stateless search strategy*, i.e. a search strategy that does not capture the shared state of the program. Specifically, recording an interleaving sequence involves storing information only

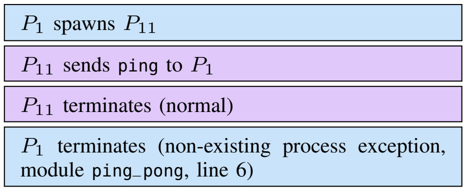
[1] The general form of receive expressions is `receive...after`. What comes after the `after` keyword is a *timeout* value: either an integer or the special value `infinity`, in which case the behavior is that of a `receive` expression without an `after` clause.

[2] To be precise the probability that reductions are exhausted at the `spawn` call is $1/2,000$.

```
-module(test).
-export([pong_test/0]).
-include_lib("eunit/include/eunit.hrl").

pong_test() ->
  ?assertEqual(ok, ping_pong:pong()).
```

(a) Test module for the running example



P_1 spawns P_{11}

P_{11} sends ping to P_1

P_{11} terminates (normal)

P_1 terminates (non-existing process exception, module ping_pong, line 6)

(b) Erroneous interleaving sequence as reported by Concuerror

Figure 2: Erroneous interleaving sequence found by running the test in Concuerror.

about context switches, while enforcing the execution of all such sequences consists in efficiently controlling when the participating processes yield or resume execution.

The delegation of control over process execution from the Erlang scheduler to Concuerror is achieved through source-to-source instrumentation of the program under test and execution in a totally unmodified Erlang/OTP runtime system. The alternative would have been to modify the runtime system, but that approach would run the risk of slightly altering the runtime semantics, besides being more difficult to implement and maintain across different versions of Erlang/OTP. More concretely, the program undergoes a parse transformation that inserts *preemption points* in the code, i.e. points where a context switch is allowed to occur, without altering its semantics. In practice, a context switch may occur at any function call during the execution of a process in the Erlang VM. However, to avoid generating redundant interleaving sequences that lead to the same shared state, instrumentation in Concuerror inserts preemption points only at process actions that interact with (i.e. inspect or update) this shared state, which are very few in Erlang. We call such actions *preemptive*. As a result, as long as the semantics of the program under test is not altered and preemption points are inserted at all preemptive actions, our approach is both sound and complete in terms of exploring all valid interleaving sequences and detecting all the concurrency errors it targets.

Even though Erlang programs have very few points where interaction with shared state occurs, the approach we have described so far may quickly become infeasible as the number of interleaving sequences grows exponentially in the number of preemption points. To further reduce redundancy, Concuerror employs a novel technique for avoiding sequences that involve processes blocking on `receive` expressions. In addition, it adapts to the Erlang setting a heuristic prioritization technique, called *iterative context-bounding* [3]—*preemption bounding* for short—that restricts the number of explored interleaving sequences with respect to a user-specified parameter.

Note that, as Concuerror does not keep track of the shared state, executing the same interleaving sequence more than once may result in different such states in case there are interactions with uninstrumented processes. In principle, this problem could be solved by instrumenting all processes participating in a test execution, including Erlang system and library processes. In practice, however, this approach may quickly become infeasible again. For this reason, Concuerror allows the user to choose which modules of the program under test to instrument and opt for testing process interactions only at the user-level ignoring any interactions with system or library processes.

Usage of Concuerror. The typical usage of Concuerror is as follows. The user opens the tool's *graphical user interface* (GUI), loads a number of Erlang modules and chooses a test to run. Note that Concuerror can run already existing test functions and requires no modifications to them or to the program under test. In addition, since its testing approach is systematic, programmers need not spawn huge numbers of processes in their tests to increase the chances of detecting any concurrency errors—the minimum number of processes that is necessary to cause the error will do. As an example, let us assume that the user loads module `ping_pong` of Figure 1 and a module containing a test for this program shown in Figure 2a. This test uses an EUnit [5] assertion to check whether the return value of `ping_pong:pong/0` is `ok`.

As a first step, Concuerror's *instrumenter* applies an automatic parse transformation to the loaded modules in order to insert preemption points into the code. In our running example, preemption points are inserted at all process interactions with the shared state, i.e. process creation (`spawn/1` on line 6), process registration (`register/2` on line 6) and message passing (`receive` on line 7 and `!/2` on line 10). Even though in this simple example preemption points are inserted at most process actions, in real programs the actions that interact with the shared state constitute only a small portion of the code, thus allowing our approach to handle large programs.

After the transformed modules have been compiled and loaded, the user chooses an exported function to execute under Concuerror, like the `pong/0` function of Figure 1 or the `pong_test/0` function of Figure 2a. The tool's *scheduler* executes all possible interleaving sequences of the selected test function—up to the current preemption bound—and reports detailed interleaving information on any errors encountered, like the exception described in the previous section. The error that Concuerror finds is shown in Figure 2b.

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, the preemption bound can be increased for a more thorough exploration, depending on the program's complexity and the user's time constraints. If the exploration 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. In this case, Concuerror functions not only as a testing, but also as a verification tool.

IV. Program Instrumentation

Concuerror instruments the code of the program under test at the granularity of modules. The translation is source-to-source and, as a result, processes yield and resume execution at preemption points with a simple `receive` expression:

```
pause() ->
  receive scheduler_prompt -> continue end.
```

By calling Concuerror's `pause/0` function, a process blocks on the `receive` expression until a prompt from the scheduler is received. In this section, we mainly focus on the instrumentation of built-in function calls and `receive` expressions. However, our approach allows the user to also insert preemption points at any function call that interacts with the shared state. This gives the alternative of instrumenting the code at a higher level than that of Erlang primitives.

Since, as described in Section II, process exits might affect the execution of other processes, we have chosen to place preemption points *after* any interaction with the shared state to conveniently separate the last such interaction from a process exit. This implies that a context switch also needs to occur before a newly spawned process starts executing the user code, otherwise the process would only yield execution after executing the first interaction with the shared state.

A. Built-In Function Calls

The instrumentation of built-in function calls that interact with the shared state consists in their substitution with calls to appropriate wrapper functions provided by Concuerror.

The signature of a wrapper function is identical to the signature of the BIF it is replacing, i.e. it accepts the same arguments and returns the same values. For instance, in our running example, the `erlang:spawn/1` call is transformed into a call to the wrapper function `concuerror:spawn_wrapper/1` with the same arguments, and the `!/2` expression is transformed into a `concuerror:send_wrapper/2` call.

Internally, all wrapper functions have the same structure: (1) the original BIF is called, (2) the scheduler is notified of the process action, (3) the process yields execution (via a call to `concuerror:pause/0`) until the scheduler prompts it to continue, and (4) when execution resumes, the result of the original BIF call is returned. As an example, consider the declaration of `concuerror:spawn_wrapper/1`, which is shown in Figure 3. Note that the newly spawned process yields execution immediately after its creation. In Figure 4, the declaration of `concuerror:send_wrapper/2` is shown, which uses a macro to instrument the message that is being sent to the destination process so that it contains more information about the sender. This information includes the sender's PID and a unique identifier for distinguishing it from processes that have not been instrumented by Concuerror (e.g. processes pertaining to the Erlang VM).

```
spawn_wrapper(F) ->
  Fun = fun () -> pause(), F() end,
  PID = spawn(Fun),
  notify_scheduler(spawn, PID),
  pause(),
  PID.
```

Figure 3: Concuerror's `spawn/1` wrapper.

```
send_wrapper(Dest, Msg) ->
  Pair = {Dest, Msg},
  Dest ! ?INSTR_MSG(Msg),
  notify_scheduler(send, Pair),
  pause(),
  Msg.
```

Figure 4: Concuerror's `send/2` wrapper.

B. `receive` Expressions

The instrumentation of `receive` expressions is more complex than that of BIF calls, as `receive`s are language expressions that are more difficult to intercept and whose semantics dictates that processes might block while waiting for a matching message to be received.

Figure 5 shows the instrumentation of the `receive` expression in the running example. The call to function `concuerror:receive_check/1`, whose definition is shown in Figure 6, ensures that the process executing the `receive` does not block forever in case no matching message ever arrives. More specifically, on line 3 of Figure 6, the anonymous function is used to look for matching messages in the process mailbox without receiving them, i.e. without removing them from the message queue. Note that in the `case` expression of the anonymous function there are additional clauses both for uninstrumented matching messages—sent by processes executing uninstrumented code—and for instrumented or uninstrumented messages that do not match. If the mailbox contains a matching message (line 4 of Figure 6), then function `concuerror:receive_check/1` returns, the message is consumed by the `receive` expression of Figure 5, the scheduler is notified, and the process yields execution at a preemption point (via a call to `concuerror:receive_notify/2`). If, however, the mailbox contains no matching messages, the process notifies the tool's scheduler that it is blocked and enters a busy-wait loop checking for the arrival of a matching message (line 5). As

```
concuerror:receive_check(
  fun (Message) ->
    case Message of
      {?UNIQUE, _, ping} -> match;
      ping -> match;
      _ -> no_match
    end
  end),
receive
  {?UNIQUE, PID, ping} ->
    concuerror:receive_notify(PID, ping), ok;
  ping ->
    concuerror:receive_notify(uninstrumented, ping),
    ok
end
```

Figure 5: Instrumentation of the `receive` expression in the running example. The general instrumentation of `receive`s is similar but needs to ensure that all variables it introduces (e.g. `PID` in this code) are fresh.

```
1  receive_check(F) ->
2    {messages, Mailbox} = process_info(self(), messages),
3    case match(F, Mailbox) of
4      match -> continue;
5      no_match -> notify_scheduler(block, self()),
6                  loop(F)
7    end.
8
9  loop(F) ->
10   {messages, Mailbox} = process_info(self(), messages),
11   case match(F, Mailbox) of
12     match -> notify_scheduler(unblock, self()),
13              pause();
14     no_match -> loop(F)
15   end.
16
17 match(F, []) -> no_match;
18 match(F, [Msg|Msgs]) ->
19   case F(Msg) of
20     match -> match;
21     no_match -> match(F, Msgs)
22   end.
```

Figure 6: The `concuerror:receive_check/1` function.

soon as such a message arrives, the process requests to be unblocked (line 11) and when the scheduler prompts it to continue, the message is received.

C. Timeouts

Concuerror's instrumenter eliminates any integer timeouts or delays in the code under test by setting them to zero. The rationale behind this design decision is twofold: (1) it is not good programming practice for the robustness of an application to depend on real-time constraints, and (2) since our tool soundly explores process interleaving, it always explores interleaving sequences in which a process action is scheduled after the execution of other process actions, for instance due to a delay.

As an example, consider a `receive` expression with patterns and an `after` clause. A timeout value of `infinity` means that the process should wait indefinitely for a matching message, which is equivalent to the same `receive` expression without the `after` clause. In such cases, the expression is instrumented as discussed in Section IV-B. If, however, the timeout value evaluates to an integer, then it is set to zero. Consequently, the process will never block on the `receive` and it is not necessary to check whether there is a matching message in the process mailbox, i.e. the `concuerror:receive_check/1` call of Figure 5 is not needed. This decision to gracefully ignore timeouts sacrifices Concuerror's completeness. However, given that delays in Erlang are not exact but instead lower bounds (i.e. at least as long as requested), we decided to take the risk of producing interleaving sequences that are rare or even impossible in practice rather than not handle programs with real-time constraints at all. For instance, if we assume that there is a significant delay in the `ping/1` function of the running example before message `ping` is sent, then the process executing the code of function `pong/0` is very unlikely to throw an exception. Nonetheless, Concuerror ignores the delay and produces the erroneous interleaving sequence, which is practically rare but still possible according to the runtime semantics of the language.

V. Process Scheduling

After having described the syntactic transformations through which Concuerror controls interleaving sequences, let us now present our strategy for systematically exploring the space of such sequences and discuss the details of enforcing each of them.

A. Search Strategy

For representing interleaving sequences, each process must be identified in a way that is both unique and constant across repeated executions of a test function. For this reason, Concuerror assigns to each process a logical identifier (LID), i.e. a string that uniquely identifies the process in the process tree hierarchy. More concretely, the initial process executing the code of the test function is identified as P_1, the first two processes spawned (at runtime) by P_1 as P_{11} and P_{12}, the first process spawned by P_{12} as P_{121}, and so on. With this definition, an interleaving sequence is represented as a sequence of LIDs. Note, however, that not every LID sequence is a valid interleaving sequence; for example, the sequence $P_{11}P_1$ is not valid, because P_{11}, which is spawned by P_1, can never be the first process to run.

Concuerror explores the space of valid interleaving sequences in a depth-first way. An iteration of the search consists in running one process at a time to enforce a specific interleaving sequence. Algorithm 1 shows the search algorithm in pseudo-code. On lines 2–4, an empty interleaving sequence prefix is pushed to an empty stack that stores unexplored prefixes. On line 5, a list of erroneous interleaving sequences is initialized to the empty list. Each iteration of

Algorithm 1 Depth-first search in process interleaving space

```
1  function SEARCH()
2    $unexploredPrefixes \leftarrow$ empty stack
3    $emptyPrefix \leftarrow$ empty list
4    PUSH($emptyPrefix$, $unexploredPrefixes$)
5    $erroneousPrefixes \leftarrow$ empty list
6    while not ISEMPTY($unexploredPrefixes$) do
7      $currentPrefix \leftarrow$ POP($unexploredPrefixes$)
8      REPLAY($currentPrefix$)
9      while not PROCESSTERMINATION() and not ERROR() do
10       $activeProcesses \leftarrow$ GETACTIVEPROCS()
11       $nextProcess \leftarrow$ POP($activeProcesses$)
12       foreach $process$ in $activeProcesses$
13         $unexploredPrefix \leftarrow$ COPY($currentPrefix$)
14         APPEND($process$, $unexploredPrefix$)
15         PUSH($unexploredPrefix$, $unexploredPrefixes$)
16       EXECUTE($nextProcess$)
17       APPEND($nextProcess$, $currentPrefix$)
18     if ERROR() then
19       APPEND($currentPrefix$, $erroneousPrefixes$)
20   return $erroneousPrefixes$
```

the main loop pops a prefix from $unexploredPrefixes$ and calls function REPLAY to schedule the participating processes according to this prefix (lines 7–8). Note that REPLAY executes a prefix by first spawning the initial process P_1. After $currentPrefix$ has been replayed, the inner loop is entered: an active process is chosen for execution (lines 10–11), the alternative interleaving sequence prefixes are stored for later exploration (lines 12–15), the chosen process is executed until its next preemption point (line 16), and the current prefix is updated (line 17). This inner loop is executed until either all processes terminate successfully or an error occurs (line 9). In the latter case, the erroneous interleaving sequence is added to the error list (lines 18–19), which is returned by the function (line 20) when the search terminates, i.e. when $unexploredPrefixes$ becomes empty (line 6), in which case there are no more unexplored interleaving sequences[3].

To evaluate the complexity of the search, observe that for a program with n processes and k_i preemption points per process $i = 1, \ldots, n$ ($k_i \leq k$ for some k), the number of different interleaving sequences may be as large as the number of distinct k_i-multiset permutations:

$$\binom{\sum_{i=1}^{n} k_i}{k_1, \ldots, k_n} = \frac{(\sum_{i=1}^{n} k_i)!}{\prod_{i=1}^{n} k_i!} \leq \frac{(nk)!}{(k!)^n}$$

Therefore, the time complexity of Algorithm 1 depends on the number of processes and preemption points in a program (not on its size) and is in the worst case exponential in both n and k. Furthermore, since at most $n-1$ interleaving sequence prefixes are stored at each of the (at most) nk preemption points, the space complexity is $O(n^2k)$.

```
1 handle_spawn(ParentLID, ChildPID, Context) ->
2   link(ChildPID),
3   ChildLID = newLID(ChildPID, ParentLID),
4   logAction({spawn, ParentLID, ChildLID}),
5   NewActive = sets:add_element(ChildLID,
6                                 Context#context.active),
7   NewPrefix = Context#context.prefix ++ [ParentLID],
8   Context#context{active=NewActive, prefix=NewPrefix}.
```

Figure 7: Scheduler handler for `spawn/1` calls.

B. *Low-level Scheduling*

The instrumentation interface to the user code that we described in Section IV is not enough to enforce the execution of specific interleaving sequences. As shown in Algorithm 1, some additional information (e.g. $currentPrefix$, $activeProcesses$) is also required. This information is stored in a structure, called scheduler *context*, and includes two sets of active and blocked processes, the current interleaving sequence prefix and the currently running process.

In the general case, updating the context merely involves extending the current interleaving sequence prefix with the process executed last (lines 16–17 of Algorithm 1) and moving this process to either the active or blocked set depending on its last preemptive action. However, there are cases that require special handling. As an example, let us describe how the scheduler updates the context after a `spawn/1` call (see Figure 7). Function `handle_spawn/2` takes as arguments the LID of the parent process that called `spawn/1`, the PID of the newly spawned process and the current context. On line 2, the scheduler links to the new process; it is essential that the scheduler be linked to all processes under test so that it is notified of their exits (normal or abnormal) with appropriate `'EXIT'` messages. On line 3, the spawned process is assigned a new LID and, on line 4, the spawning action is recorded. The remaining lines of `handle_spawn/2` update the context by adding the child LID to the active set—the parent LID is already active—and extending the current interleaving sequence prefix with the LID of the parent process.

The scheduler detects the three types of errors that Concuerror targets as follows. A stuck process error is reported whenever all alive processes are blocked, which means that the program under test cannot make progress. Such an error might signify the existence of a deadlock in case the stuck processes circularly depend on each other to continue execution. Exceptions or assertion violations are reported whenever a user process exits abnormally. The distinction between these two kinds of errors is made depending on the details of the process exit information.

[3] The actual implementation reports each error as soon as it is found by the algorithm.

VI. Efficiency Improvements

The exponential time complexity of Algorithm 1 suggests that the search becomes quickly infeasible as the number of preemption points increases. In the following, we present two techniques that significantly improve the efficiency of the search. The first is a simple partial-order reduction technique that avoids redundant interleaving sequences involving process blocks on `receive` expressions. The second is a heuristic method, called preemption bounding, that bounds the number of allowed context switches and drastically reduces the number of explored interleaving sequences.

A. Blocking Avoidance

As we have already mentioned, when a process executes a `receive` expression without any matching messages in its mailbox, it blocks. The scheduler then chooses an active process (if any) for execution. As explained in Section IV-B, the blocked process only becomes active again as soon as a matching message arrives. More importantly, although the action of blocking on a `receive` expression inspects the shared state (by checking the process mailbox), it does not update it. In addition, the process cannot resume execution and interact with the shared state again right after the block. Therefore, a context switch is guaranteed to occur at that point, and the inspection of the shared state does not affect the execution of any process, i.e. is not really a preemptive action. Consequently, interleaving sequences containing process blocks are redundant and can be soundly ignored.

If before resuming the execution of a process (line 16 of Algorithm 1) we knew that it would eventually block, we could avoid exploring such a redundant interleaving sequence by choosing another process for execution instead. Although we do not have this information available in advance, in the instrumented version of the code it is possible to check whether the already executing process will block on a subsequent `receive` expression, which still allows us to avoid these redundant sequences. In particular, when the currently running process is about to block, the scheduler moves it to the blocked set but does not extend the current interleaving sequence prefix. This means that the search continues without taking into account the execution of the blocked process. Moreover, when replaying an interleaving sequence prefix, if the last action of the prefix is a process block, then the search iteration is soundly aborted since the prefix is redundant and any alternatives will have already been added to the stack of unexplored prefixes (lines 12–15 of Algorithm 1).

Despite the simplicity of blocking avoidance, Section VII shows that it significantly reduces the search space and, therefore, dramatically improves performance on programs that make heavy use of message passing.

B. Preemption Bounding

To further improve efficiency, we apply a heuristic technique, called preemption bounding, which was first proposed by Musuvathi and Qadeer [3] and builds on the hypothesis that most concurrency errors are revealed with a small number of context switches. The main idea is to impose an upper limit on the number of context switches that the scheduler is allowed to enforce during each search iteration. However, certain context switches are necessary and leave no choice to the scheduler, namely those that occur because of process blocks or exits. These context switches are dictated by the program semantics and are called non-preemptive. In contrast, the context switches that are optionally enforced by the scheduler are called preemptions and their number may be bounded in order to reduce the explored state space. Consequently, even if there is a bound on the number of allowed context switches, the program under test is still executed from start to finish unlike in other heuristic techniques, such as iterative deepening.

The preemption bounding algorithm proposed by Musuvathi and Qadeer starts with a preemption bound of zero, which is iteratively increased until the chosen value is reached. Instead of a single stack for storing unexplored interleaving sequence prefixes ($unexploredPrefixes$ in Algorithm 1), two stacks are used, one for storing prefixes within the current preemption bound ($S_{current}$), and one for storing prefixes that exceed this bound (S_{next}). We developed a modified version of this algorithm in order to incorporate blocking avoidance to it. The need for such an adjustment is shown in the following execution scenario.

Let the program under test consist of three processes, P_1, P_{11} and P_{12}, and assume that the current preemption bound is $c = 0$. Also, assume that the current interleaving sequence prefix is RP_1, where $R \in \{P_1, P_{11}, P_{12}\}^*$, i.e. P_1 is the process executed last, and all three processes are active. Due to the value of the current bound, no preemptions are allowed, interleaving sequence prefixes RP_1P_{11} and RP_1P_{12} are pushed to S_{next}, and execution continues with P_1. Assume that this time P_1 blocks on a `receive` expression. If we employ the blocking avoidance technique presented in Section VI-A, P_1 is moved to the blocked set and the current interleaving sequence prefix is not updated. At this point, execution continues with an active process, say P_{11}. As a result, the context switch from P_1 to P_{11}, previously considered preemptive, is found to be non-preemptive. Therefore, in our version of preemption bounding, RP_1P_{11} is removed from S_{next} because it is the current interleaving sequence prefix, and RP_1P_{12} is moved from S_{next} to $S_{current}$ for exploration within the current preemption bound. Thus, we adapt the preemption bounding algorithm as it was first proposed to also take advantage of blocking avoidance.

For a program with n processes, $k_i \leq k$ preemption

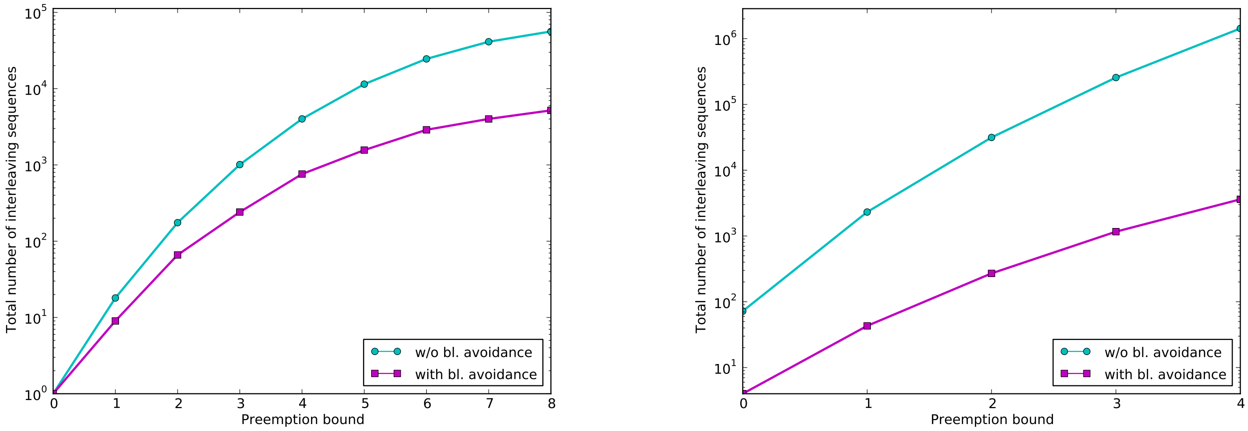


Figure 8: Number of interleaving sequences with and without blocking avoidance for the `pidigits8_test/0` (left) and `attach_noping_test/0` (right) tests.

Table I: Number of processes (n) and maximum number of preemption points per process ($\max k_i$) for the benchmarks.

Test function	n	$\max k_i$
`pidigits8_test/0`	2	10
`attach_noping_test/0`	3	14
`attach_test/0`	3	22
`dialyzer_test/0`	14	78

points per process, $b_i \leq b$ non-preemptive context switches per process and a preemption bound of c, the number of interleaving sequences is exponential in n, b and c, but is now polynomial in k. As k is usually the most significant factor in determining the number of interleaving sequences, preemption bounding allows for efficiently testing large programs as long as the bound c is chosen sufficiently small.

VII. Evaluation and Experience

We first evaluate the effectiveness of the techniques presented in the previous section, blocking avoidance and preemption bounding, on two small Erlang programs. Note that, as shown in the preceding sections, the complexity of the programs under test should not be judged in terms of program size (e.g. number of LOC), but rather in terms of the number of participating processes and the number of preemption points per process. Table I contains the values of these parameters for the test functions described below.

The first program[4] uses message passing to calculate and print the first N digits of π. The main process of the program calculates the digits and spawns a new process for printing them. Test function `pidigits8_test/0` calculates and prints the first eight π digits. The second program[5], which we have presented in detail in a previous publication [2], involves the creation of a generic registration server, which could, for instance, be used to manage limited system resources. Processes may attach to and detach from the server, but only a limited number of processes are allowed to be attached at any moment. Test function `attach_noping_test/0` involves a server process and two other processes that concurrently attempt to attach themselves to it. Test function `attach_test/0` is similar to `attach_noping_test/0`, but the two processes also ping the server to confirm that they have been correctly attached.

First, we assess the effect of blocking avoidance on the number of explored interleaving sequences. Figure 8 shows the total number of interleaving sequences for increasing values of the preemption bound with and without blocking avoidance. In both tests, our technique significantly reduces the number of generated interleaving sequences. In particular, in `pidigits8_test/0` the total number of sequences is reduced by one and in `attach_noping_test/0` by almost three orders of magnitude. Observe that, as the preemption bound increases, the efficiency improvement also increases.

We also evaluate how preemption bounding affects the execution time for test functions `attach_noping_test/0` and `attach_test/0`. The results are shown in Table II. Note that for `attach_noping_test/0` the execution times are negligible even for large preemption bounds. For `attach_test/0`, which is more complex, execution time increases significantly with the preemption bound. This suggests that lower preemption bounds may be used in fast regression (unit) test suites, while larger bounds are perhaps better suited for nightly testing of more intricate interleaving sequences. As a side note, a preemption bound of two (approximately 2 minutes of execution time for 14 test functions) was enough to reveal all concurrency errors encountered during the development of the registration server [2].

Besides applying Concuerror on small benchmarks, we

[4] http://shootout.alioth.debian.org/u32q/program.php?test=pidigits\&lang=hipe

[5] https://github.com/mariachris/Concuerror/tree/master/resources/tdd

Table II: Testing time vs. preemption bound for the two registration server tests.

Preemption bound	≤2	3	4	5	6	7
`attach_noping_test/0`	0m0s	0m0s	0m1s	0m2s	0m5s	0m7s
`attach_test/0`	0m1s	0m6s	0m26s	1m26s	3m59s	9m23s

have applied it on large Erlang code bases. We report on one such experience: applying Concuerror to the code base of Dialyzer, a widely-used static analysis tool which is included in the Erlang/OTP distribution. Dialyzer's code base is about 28,000 LOC and its parallel version uses Erlang's concurrency primitives extensively [6]. Using one of the tests in Dialyzer's regression suite and the default preemption bound of two, Concuerror was able to find, in less than a minute, various interleaving sequences that resulted in a stuck server process for one of Dialyzer's components. This particular concurrency error had not been detected during several months of Dialyzer's development. With a small change we were able to correct this error and include its fix just in time for the release of Erlang/OTP R15B02 (September 2012).

It should be noted that Dialyzer is a very complex piece of software. The particular unit test, which revealed the concurrency error, takes about half a minute to complete if run in parallel on a machine with four physical cores (eight threads). The test spawns 14 Erlang processes and the maximum number of preemption points per process is 78 (see Table I). In the code base with the error present, Concuerror detects various interleaving sequences where the server process is stuck. After correcting the error and running Concuerror on the same test and with the same preemption bound, Concuerror does not report any other concurrency errors after exploring a significant number of interleaving sequences: 24,900 within 42 minutes and 712,000 within 700 minutes. While this does not mean that Dialyzer's code base is free from concurrency errors, systematically exploring this number of interleaving sequences increases the developers' confidence significantly.

VIII. Related Work

The problem of effectively exploring process interleaving to detect concurrency errors in programs is well-studied by now. In the literature, one can find various approaches addressing this problem in the fields of software testing and model checking. Still, our work demonstrates how to systematically test and replay process interleaving in an actor programming language and how to design an effective and efficient tool that takes advantage of the characteristics of these languages.

The work most closely related to ours is the work on CHESS [1], a tool that enables systematic testing and debugging of three classes of multithreaded programs—user-mode Win32, .NET and Singularity programs—by introducing a thin wrapper layer between the program under test and the concurrency API. In contrast to our approach, CHESS targets imperative programs with significantly more shared state than Erlang, and gains control over thread interleaving through platform-dependent wrappers instead of syntactic transformations of the source code. Both tools, however, are similar in that they employ model checking techniques to effectively explore all process interleaving sequences.

Other related work in the software testing community is ExitBlock-RW [7], the first dynamic partial-order reduction technique for effectively enumerating and testing valid interleaving sequences in multithreaded Java programs. This technique is based on the observation that context switches need only be placed at synchronization points, an idea that both CHESS and Concuerror borrow. More recent approaches in this area include dynamic partial-oder reduction [8], reachability testing [9], which uses on-the-fly partial-order reduction techniques, concolic testing [10], which also handles programs with inputs, and coverage-guided testing [11], which uses dynamically-learned ordering constraints over shared state interactions to select only high-risk interleaving sequences for execution.

Even though several model checkers target the verification of concurrent programs, most of them attempt to capture the program state at the cost of space explosion. Notable examples of such stateful model checkers are Bogor [12], CMC [13], Java PathFinder [14], and Basset [15], which is built on top of Java PathFinder and provides a framework for testing actor programs compiled to Java bytecode. Our approach, however, is more similar to the stateless enumeration of process interleaving sequences, as in VeriSoft [16]. Furthermore, the idea of bounding the number of context switches for efficiency reasons is also present in context-bounded model checking [17].

Testing tools for Erlang. Although there exist several testing tools for Erlang, the majority of them currently fails to be effective in Erlang's concurrent setting.

EUnit [5] is the Erlang implementation of the popular xUnit testing framework and is reportedly the testing tool used most by Erlang developers. Despite its ease of use, EUnit executes each test under a single interleaving sequence and is inadequate for detecting concurrency errors.

QuviQ QuickCheck is a property-based 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, which provides no correctness guarantees, the user is required to write properties the program should satisfy using a special notation. This requires the user's familiarity with the non-trivial task of writing properties and, additionally, excludes the use of existing unit tests without any modifications.

McErlang [18] is a model checker that utilizes a custom runtime simulator of Erlang's concurrency semantics to explore the program state space. In principle, the ability to deploy monitors using linear temporal logic specifications 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 the abnormal process exit of our running example, often requires the manual insertion of unobvious code, a task that is tedious and, more importantly, requires altering the original code. Due to problems such as these, McErlang has so far failed to build a strong user community among Erlang developers. We have good reasons to believe that the situation will be very different for Concuerror.

IX. Conclusion

We presented a framework for the systematic testing of Erlang programs in order to effectively detect and reproduce a wide class of concurrency errors. We showed how this is accomplished by first instrumenting the program under test and, subsequently, exploring the state space of interleaving sequences using an unmodified VM. To ameliorate the problem of state space explosion, we proposed and employed a novel technique for avoiding process blocks in receives and combined it with preemption bounding, a heuristic algorithm for reducing the number of explored interleaving sequences.

The main future work is developing and employing more powerful dynamic partial-order techniques that further reduce state space redundancy. We also intend to investigate whether the techniques used by Concuerror can be extended to handle the distribution primitives of the language. On a more practical level, we want to look into the issue of automatic instrumentation of library functions and investigate how property-based testing tools like PropEr [19] can be used to automatically generate test cases for Concuerror.

Acknowledgments

Most of this work was performed when all authors were associated with the National Technical University of Athens in Greece. During 2012, the research engagement of the third author in this work was partially supported by UPMARC and EU project RELEASE. We also thank Ilias Tsitsimpis and Valentin Wüstholz for their helpful feedback and comments.

References

[1] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *OSDI*. USENIX, 2008.

[2] A. Gotovos, M. Christakis, and K. Sagonas, "Test-driven development of concurrent programs using Concuerror," in *Erlang WS*. ACM, 2011.

[3] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *PLDI*. ACM, 2007.

[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 *ICFP*. ACM, 2009.

[5] R. Carlsson and M. Rémond, "EUnit: A lightweight unit testing framework for Erlang," in *Erlang WS*. ACM, 2006.

[6] S. Aronis and K. Sagonas, "On using Erlang for parallelization: Experience from parallelizing Dialyzer," in *TFP*, ser. LNCS. Springer, 2013.

[7] D. L. Bruening, "Systematic testing of multithreaded Java programs," Master's thesis, MIT, 1999.

[8] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*. ACM, 2005.

[9] Y. Lei and R. H. Carver, "Reachability testing of concurrent programs," *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, 2006.

[10] K. Sen and G. Agha, "Concolic testing of multithreaded programs and its application to testing security protocols," University of Illinois at Urbana Champaign, Tech. Rep. UIUCDCS-R-2006-2676, 2006.

[11] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *ICSE*. ACM, 2011.

[12] Robby, M. B. Dwyer, and J. Hatcliff, "Bogor: An extensible and highly-modular software model checking framework," in *ESEC/FSE*. ACM, 2003.

[13] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A pragmatic approach to model checking real code," in *OSDI*. USENIX, 2002.

[14] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *ASE*. IEEE Computer Society, 2000.

[15] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha, "A framework for state-space exploration of Java-based actor programs," in *ASE*. IEEE Computer Society, 2009.

[16] P. Godefroid, "Model checking for programming languages using Verisoft," in *POPL*. ACM, 1997.

[17] A. Lal, T. Touili, N. Kidd, and T. W. Reps, "Interprocedural analysis of concurrent programs under a context bound," in *TACAS*, ser. LNCS. Springer, 2008.

[18] L.-Å. Fredlund and H. Svensson, "McErlang: A model checker for a distributed functional programming language," in *ICFP*. ACM, 2007.

[19] M. Papadakis and K. Sagonas, "A PropEr integration of types and function specifications with property-based testing," in *Erlang WS*. ACM, 2011.