Static Detection of Deadlocks in Erlang

Maria Christakis[1] and Konstantinos Sagonas[1,2]

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

Abstract. We address the problem of detecting two different kinds of possible deadlocks in Erlang programs using static analysis. Our analysis is completely automatic, fast and effective in detecting deadlocks while avoiding most false alarms. We have integrated our analysis in dialyzer, a widely used tool for detecting software defects in Erlang programs, and demonstrate its effectiveness on open-source libraries and applications of considerable size. Despite the fact that most of these applications have been used over a long period of time and are reasonably well-tested, our analysis has detected a number of previously unknown deadlocks in their code that may have devastating effects at runtime.

1 Introduction

A deadlock is an unintended condition under which a number of actions are stuck on some synchronization primitive waiting for each other to proceed. Although deadlocks can have detrimental effects on a running system, they are likely to occur even in applications that have been used and tested for a long period of time. Concurrent programs that employ a wide range of synchronization primitives to control thread interactions and avoid other concurrency errors, such as data races, are usually particularly prone to deadlocks. On top of that, the shift to multi-core machines makes many more thread interactions possible, thus making deadlocks more likely.

Erlang [1] is a functional programming language that has been designed with the aim of simplifying concurrent programming. The language avoids the complicated management of threads and the error-prone use of explicit locking operations on shared variables, since its concurrency model is based on light-weight, user-level processes that communicate using asynchronous message passing. However, Erlang does not avoid all problems associated with concurrent execution. In particular, it provides for interactions and communication patterns between processes that may be stuck in their execution until some conditions are met, hence allowing certain kinds of deadlocks in programs. In addition, as Erlang's primary application area is in large-scale, reactive control systems that create an unbounded number of processes, such as server applications, these process interactions become complex and extremely hard to predict. Not only that, but also deadlocks may remain unexposed during testing and when revealed, it is quite distressing to reproduce them, let alone find their cause.

To ameliorate this situation, we have identified two kinds of possible deadlocks in concurrent Erlang programs, and have designed an effective analysis that detects them and brings them to the attention of the programmers. Besides tailoring the analysis to the characteristics of the language, the main challenges for our work have been to develop an analysis that: 1) is completely automatic requiring no guidance from its user, and 2) strikes a proper balance between soundness and completeness for efficiency reasons. As we will soon see, we have achieved these goals.

The contributions of our work are as follows:

– we document the most important kinds of deadlocks in Erlang programs;
– we present an effective analysis that detects these errors, and
– we demonstrate the effectiveness of our analysis by running it on a set of widely used and reasonably well-tested libraries and open source applications and reporting a number of previously unknown deadlocks in their code bases.

The next section gives a brief overview of the Erlang language and the defect detection tool that is the implementation platform for our work. Sect. 3 describes two kinds of possible deadlocks in Erlang programs, followed by Sect. 4 that presents in detail the analysis we use to detect them. The effectiveness of our analysis is evaluated in Sect. 5. The paper ends with a review of related work (Sect. 6) and some final remarks.

2 Erlang and Dialyzer

Erlang [1] is a strict, dynamically typed, functional programming language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly code reloading, automatic memory management and support for multiple platforms. Erlang's primary application area has been in large-scale embedded control systems developed by the telecom industry, but its uses have expanded to application areas such as web services, online commerce, gaming, banking, etc. The main implementation of the language, the Erlang/OTP (Open Telecom Platform) system from Ericsson, has been used quite successfully both by Ericsson and by other companies around the world to develop software for large commercial applications. Nowadays, applications written in the language are significant, both in number and in code size, making Erlang one of the most industrially relevant declarative languages.

Erlang's main strength is that it has been built from the ground up to support concurrency. Its concurrency model differs from that of most other programming languages as it is not based on shared memory but on asynchronous message passing between extremely light-weight processes (lighter than OS threads). Erlang comes with a `spawn` family of primitives to create new processes, and with `!` (send) and `receive` primitives for interprocess communication via message passing. Any data can be sent as a message and processes may be located on any machine. Each process has a *mailbox*, essentially a message queue, where each message sent to the process will arrive. Message selection from the mailbox occurs through pattern matching. To support robust systems, a process can register to receive a message if another one terminates. Erlang provides mechanisms for allowing a process to timeout while waiting for messages, a `try/catch`-style exception mechanism for error handling, and ways to organize processes in supervision hierarchies to restart or take over the duties of dead or irresponsive processes.

Since 2007, the Erlang/OTP distribution includes a static analysis tool, called dialyzer [2,3], for finding software defects in single Erlang modules or entire applications. These defects include type errors, exception-raising code, code that has become unreachable due to some logical error, etc. Nowadays, dialyzer is used extensively in the Erlang programming community and is often integrated in the build environment of many applications. The tool is totally automatic, easy to use and supports various modes of operation: command-line vs. GUI, starting the analysis from source vs. byte code, focussing on some kind of defects only, etc. For sequential programs, notable characteristics of dialyzer's core analysis are that it is *sound for defect detection* (i.e., it produces no false alarms), *fast* and *scalable*. Its core analyses that detect defects are supported by various components for creating and manipulating function call graphs for a higher-order language, control-flow analyses, efficient representations of sets of values, data structures optimized for computing fixpoints, etc. Since November 2009, dialyzer's analysis has been enhanced with a component that automatically detects *data races* in Erlang programs [4]. Recently, we have also presented a static analysis that is able to detect some commonly occurring kinds of *message passing errors* in languages with dynamic process creation and communication based on asynchronous message passing [5]. This analysis is integrated in the development version of dialyzer. Before we describe how we extended dialyzer's analyses to detect another class of concurrency errors, namely deadlocks, let us first see the kinds of deadlocks that may exist in Erlang programs.

3 Deadlocks in Erlang

In this paper, we consider a deadlock to be the condition under which the progress of a number of processes is prevented due to some dependence either on each other or on other processes. Based on this definition, we focus on two kinds of deadlocks, namely the *communication* and the *behaviour deadlocks*. We present them in the following two subsections.

3.1 Communication Deadlocks

Let us examine Erlang's main concurrency primitives in more detail:

Spawn The `spawn` primitive creates a process and returns a *process identifier* (pid) for addressing the newly spawned process. The new process executes the code of the function denoted in the argument of the `spawn`. In the example program shown in Fig. 1, two processes are spawned that will execute the

```
-module(ping_pong).

-export([play/0]).

play() ->
  Ping = spawn(fun ping/0),
  spawn(fun() -> pong(Ping) end).

ping() ->
  receive
    pong -> ok
  end.

pong(Ping) ->
  Ping ! pong,
  receive
    ping -> ok
  end.
```

Fig. 1. Example program with a communication deadlock

code of the `ping/0` function and the function closure, that just calls `pong/1`. We will refer to these processes as the *ping* and *pong* processes respectively.

Send The expression `Pid ! Msg` sends the message `Msg`, that may refer to any valid Erlang term, to the process with pid `Pid` in a non-blocking operation. In our example program, the *pong* process sends the message `pong` to the *ping* process.

Receive Messages are received with the `receive` construct. Each process has its own mailbox for messages it receives. A mailbox functions as a queue in the sense that any new messages are placed at the end of the mailbox. When a process executes a `receive`, the first message in the mailbox is matched against the patterns of the `receive` in sequential order. If the message matches some pattern, it is removed from the mailbox and the actions corresponding to the matching pattern are executed. However, if the message does not match, it is kept in the mailbox and the next message is tried instead. If this matches any pattern, it is removed from the mailbox while keeping the previous and any other messages in the mailbox. In case the end of the mailbox is reached and no messages have been matched, the process blocks (i.e., stops execution) and waits to be rescheduled to repeat this procedure. In the example program of Fig. 1, the *ping* process will receive a `pong` message from the *pong* process but the latter will block as its mailbox will be empty.

Misuse of these concurrency and communication primitives may lead to communication deadlocks. In such deadlocks, messages are the resources for which processes wait. We therefore define a *communication deadlock* as the condition under which one or more processes block on some `receive` statement. More specifically, a communication deadlock occurs in the following cases:

No messages A `receive` statement in the code executed by some process blocks because the process mailbox is always empty. This defect could reveal a set of processes mutually waiting for messages from each other without any process in the set ever sending a message.

Messages of the wrong kind A `receive` statement in the code of some process blocks because the process mailbox always contains messages of different kinds than those expected by the `receive`. Such a defect, apart from blocking the process execution, can have devastating effects on a running system, overflowing the mailbox of some process and bringing the node down.

Such types of concurrency defects might have disastrous consequences in any system, let alone the safety-critical systems developed in the telecommunications sector. But these are not the only kind of deadlocks possible in Erlang programs.

3.2 Behaviour Deadlocks

Erlang/OTP comes with some commonly employed concurrency design patterns, called *behaviours*. For example, the client-server model, comprising a central server and an arbitrary number of clients, is frequently used for resource management, i.e., the clients share a common resource managed by the server. Usually, the clients and servers in each instance of the client-server model share similar structure patterns, and behaviours are generic implementations of these patterns. The standard Erlang/OTP behaviours include the implementation of servers in client-server relations, finite state machines, event handlers and supervisors in supervision trees. User-defined behaviours may also be implemented but are beyond the scope of this paper.

When a behaviour is used in the implementation of a process, the code is divided into a generic and a specific part, called the *behaviour* and *callback modules* respectively. For built-in behaviours, the behaviour module is part of Erlang/OTP while the callback module is implemented by the user. For instance, for the creation of a server process, the user must write a callback module that exports a predefined set of callback functions.

In the example program of Fig. 2, `gen_server` and `server` are the behaviour and callback modules, as declared in the `behaviour` and `module` attributes respectively. The callback module provides an interface to the server for manipulating a counter, a set of callback functions and two macro definitions. The server interface includes functions `make_counter/1`, `count_down/1` and `set/2` that, in the order mentioned, create a counter initialized to zero, perform a countdown to zero and set the counter to a given value. The state of the counter is stored in an Erlang Term Storage (ETS) table that is public to all processes. The callback functions `init/1`, `handle_call/3` and `handle_cast/2` are implicitly called by the following functions of the behaviour module:

`gen_server:start_link(ServerName,Module,Args,Options)` which creates a generic server process that calls function `Module:init/1` with arguments `Args` to initialize. The server is registered, either locally or globally, under a name specified in `ServerName`.

```
-module(server).

-behaviour(gen_server).

-export([make_counter/1, count_down/1, set/2]).

-export([init/1, handle_call/3, handle_cast/2]).

-define(S, server).
-define(T, table).

make_counter(?T) ->
  gen_server:start_link({local, ?S}, ?S, ?T, []).

count_down(?T) ->
  gen_server:call(?S, ?T).

set(?T, N) ->
  gen_server:cast(?S, {?T, N}).
```

```
init(?T) ->
  ets:new(?T, [named_table, public]),
  ets:insert(?T, {counter, 0}),
  {ok, feeling_good}.

handle_call(?T, _From, St) ->
  [{counter, N}] = ets:lookup(?T, counter),
  case N of
    0 ->
      ok;
    _ ->
      ets:insert(?T, {counter, N - 1}),
      gen_server:call(?S, ?T, infinity)
  end,
  {reply, 0, St}.

handle_cast({?T, N}, St) ->
  ets:insert(?T, {counter, N}),
  {noreply, St}.
```

Fig. 2. Example program with behaviour deadlock

`gen_server:call(ServerRef,Request)` which makes a synchronous call to the server with reference `ServerRef` by sending a request and waiting until a reply arrives or a timeout occurs. The default value for the timeout is 5000 ms. If no reply is received within this specified time, the call fails. The server calls function `Module:handle_call/3` to handle the request. Here, the reference `ServerRef` is determined by the registered name `ServerName` passed to `gen_server:start_link/4` and `Module` refers to the callback module also passed to the same behaviour function.

`gen_server:call(ServerRef,Request,Timeout)` which has the same functionality as `gen_server:call/2`. The only difference is that this call allows the user to specify how long to wait for a reply. `Timeout` may be either a positive integer specifying the number of milliseconds to wait or the atom `infinity` meaning that the call will wait indefinitely for a reply.

`gen_server:cast(ServerRef,Request)` which makes an asynchronous call to the server with reference `ServerRef` by sending a request and returning immediately. The server calls function `Module:handle_cast/2` to handle the request.

The astute reader has already noticed that the `count_down/1` function of Fig. 2 will fail due to a deadlock: It calls function `gen_server:call/2` and waits for a reply. However, the `handle_call/3` function, that handles this request, instead of calling itself recursively in order to successively reduce the value of the counter, sends another synchronous request to the server with the `gen_server:call/3` call. If the example were not made up, we would assume that the programmer considered the `gen_server:call/3` and `handle_call/3` functions equivalent, forgetting that the former is the synchronous version of the latter. As a result, the `gen_server:call/2` function becomes synchronously recursive, deadlocks and fails when the default timeout occurs. Taking the example program a step further, it is possible for more than one servers to be involved in such a deadlock in case a server's synchronous request issues additional such requests to other servers that analogously issue synchronous requests to the server that initially triggered them. We therefore define a *behaviour deadlock* as the condition under which two or more synchronous behaviour calls are mutually waiting for each other. Usually, when such a deadlock occurs, a timeout comes to the rescue.

Behaviour deadlocks may be caused by any synchronous behaviour functions of Erlang/OTP since, apart from being synchronous, they require the interference of the user for the implementation of the callback functions and are thus prone to errors. Having presented these two kinds of deadlocks in Erlang, that are also the categories of deadlocks that our analysis detects, let us now present the details of the analysis.

4 The Analysis

Statically detecting the deadlocks we described in the previous section is not trivial. In order to detect communication deadlocks in a higher-order language with unlimited process creation and asynchronous message passing such as Erlang, the *communication topology* of processes needs to be determined in a fairly precise way. On the other hand, the behaviour deadlock detection requires a quite different approach. Concrete hard-coded information about the Erlang behaviours and their functionality must be provided to the analysis, that will use this information for the creation of a *wait-for graph*, the basis for detecting these errors. We have designed and implemented such analyses and describe them in this section. Although we describe them as being distinct, the actual implementation blurs the lines of this distinction in order to be efficient.

We have integrated our analyses in dialyzer both because it is a widely used tool in the Erlang community [6] and because many of the components that it relies upon were already available or could be easily extended to provide the information that the analyses need. The analyses start with the user specifying a set of directories/files to be analyzed. Rather than operating directly on Erlang source, all of dialyzer's passes operate at the level of Core Erlang [7], the language used internally by the Erlang compiler. Core Erlang significantly eases the analysis of Erlang programs by removing syntactic sugar and by introducing a `let` construct that makes the binding occurrence and scope of all variables explicit.

4.1 Detection of Communication Deadlocks

Conceptually, the analysis for the detection of communication deadlocks has three distinct phases: an initial phase that scans the code to collect information needed by the subsequent phases, a phase where a communication graph is constructed, and a phase where communication deadlocks are detected.

In the first phase of the analysis, as the source code is translated to Core Erlang, dialyzer constructs the *control-flow graph* (CFG) of each function and function closure that will later be traversed in search of concurrency primitives.

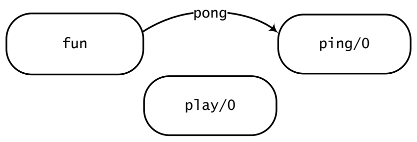

Fig. 3. Communication graph of example program with communication deadlock

Dialyzer then uses the escape analysis of Carlsson et al. [8] to determine values, in particular closures, that escape their defining function. Given this information, dialyzer also constructs the *inter-modular call graph* of all functions and closures, so that subsequent analyses can use this information to speed up their fixpoint computations. Based on both the escape analysis and the call graph, the analysis identifies the processes that might be created at runtime. Besides control-flow, the analysis also needs data-flow information and more specifically it needs information on whether variables can possibly refer to the same process identifier or not. This information is computed and explicitly maintained by the *sharing/alias analysis* component in dialyzer's race analysis [4]. In addition, our analysis exploits the fact that dialyzer computes type information at a very fine-grained level [9] to decide whether messages match their receiving patterns.

The second phase of the analysis determines the interprocess communication topology in the form of a graph. Each vertex of the graph represents a function or closure whose code may be run by a separate process at runtime. This information is computed by a pre-processing step during the construction of the call and control-flow graphs. Functions that correspond to root nodes in the call graph as well as functions or escaping closures that are passed as arguments to `spawn` calls are assumed to be executed by separate processes. For our example program, the communication graph will contain three nodes, for functions `play/0` and `ping/0` and for the closure. Every edge of the communication graph is directed and corresponds to a communication channel between two processes. Naturally, its direction of communication is from the source to the target process, meaning that messages are sent in that direction. Each edge is annotated with the type information of the messages that are sent through the channel. In order to determine the graph edges, we need to inspect every possible execution path of the program for messages that are passed between processes. To this end, the analysis traverses the CFGs of the functions corresponding to the vertices in the communication graph using depth-first search. For each vertex, if the traversal finds a send operation to some pid, the analysis takes variable sharing into account to determine the recipient process that this pid refers to, thus identifying the target vertices of each edge. In the end, this traversal creates the complete set of edges in the communication graph. For the code of Fig. 1, the communication graph has one edge from the closure to vertex `ping/0` annotated with `pong` since a `pong` message is sent from the process executing the code of the closure to the *ping* process. The communication graph for this example program is illustrated in Fig. 3.

At the final stage of the analysis, the CFG of each function that corresponds to a vertex in the communication graph is traversed anew to detect any communication deadlocks. A vertex in the communication graph with in-degree equal to zero indicates that no messages are sent to the process it represents. Hence, the traversal of the CFG emits a warning for each `receive` construct it encounters. A vertex with in-degree greater than zero indicates that messages are sent to the process and the analysis determines whether these messages will be received. In case the process expects to receive messages (i.e., there are `receive`s in the CFG), the analysis takes into account the type information of both the messages and the `receive` patterns in order to decide whether they match. In short, at the end of the CFG traversal, warnings are emitted for `receive` constructs that do not have matching patterns for any messages. For the example program, the analysis inspects the CFG of the closure, that has in-degree zero, and finds that there is a `receive` in the code executed by the process, namely in `pong/1`. Consequently, it emits a warning with the filename and line number of the `receive` reporting a communication deadlock.

Note that this part of the analysis reuses components for the detection of message passing errors, as we consider any message passing errors that involve a blocking `receive` to be communication deadlocks. These components employ optimization ideas and techniques to avoid false alarms in case the available static information is too limited to construct the exact interprocess communication topology [5].

4.2 Detection of Behaviour Deadlocks

The behaviour deadlock detection analysis also has three phases: an initial phase that scans the code and collects information, a phase where a wait-for graph is constructed, and a phase where behaviour deadlocks are detected.

The first phase of the analysis collects information that will be used for the construction of the wait-for graph in the next phase, such as dialyzer's type information, the CFGs of each function and function closure and the intermodular call graph. Besides this, during the source code translation to Core Erlang, information is obtained on which behaviours, if any, are implemented by the user. For the example of Fig. 2, the analysis finds that the generic server (`gen_server`) design pattern is implemented since it is declared in the `behaviour` attribute of the module. In addition, calls to any behaviour functions need to be identified. To compute this information, the analysis first refers to a hard-coded set of calls for each implemented behaviour. Then, during the construction of the function CFGs, it collects a set of program points containing any of these calls. The behaviour set, i.e., the set of behaviour calls, for the example program contains the `gen_server:start_link/4`, `gen_server:call/2,3` and `gen_server:cast/2` calls. Finally, the analysis refers once more to its hard-coded information to generate a subset of the behaviour set, containing only the synchronous calls. The `gen_server` behaviour offers two families of synchronous calls, the `gen_server:call/2,3` and `gen_server:multi_call/2,3,4` calls, that are all handled by the same callback function, the `handle_call/3` function.

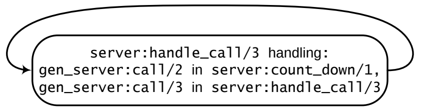
server:handle_call/3 handling:
gen_server:call/2 in server:count_down/1,
gen_server:call/3 in server:handle_call/3

Fig. 4. Wait-for graph of example program with behaviour deadlock

Based on this information, the analysis filters the behaviour set to create the synchronous subset containing the `gen_server:call/2,3` calls.

The second phase of the analysis constructs a wait-for graph, that is essentially a call graph of synchronous calls. Every vertex of the graph represents a callback function that handles the requests of a number of synchronous calls. As we have already mentioned, both of the calls in the synchronous set are handled by the same callback function. Thus the wait-for graph for our example program has only one vertex annotated with the callback function and the program points containing the synchronous calls that it handles. However, the analysis does not have any information about the callback module, i.e., the module where the `handle_call/3` function is defined. To identify this module, the analysis first uses its hard-coded information to establish which argument of the synchronous calls refers to the server and then exploits dialyzer's type information to extract the server name. In Fig. 2, the `gen_server:call/2,3` calls refer to the server `server`, an atom defined in a macro definition. Now, the analysis uses the behaviour set to look for any calls that might register the server under this name. In this case, `gen_server:start_link/4` is the only call that could register the server, as described in Sect. 3.2, and the type information of its first argument confirms the name. From the type information of its second argument, it is established that the `server` module is the callback module for this behaviour. Thus the analysis infers that the requests of the `gen_server:call/2,3` calls are handled by the `server:handle_call/3` function and the vertex of the wait-for graph is created. Note that if the type information for the behaviour calls were not precise enough to correctly identify the callback module, the analysis would not proceed to avoid emitting any false alarms. In the wait-for graph, there is a directed edge from vertex V_1 to vertex V_2 if there exists a synchronous call whose request is handled by the callback function of V_1 that must wait for a synchronous call handled by the callback function of V_2 to return. The edges are determined by checking whether there is a path in the inter-modular call graph from the callback function in each vertex of the wait-for graph to any caller function of a synchronous call in the same or any other vertex of the graph. If such a path exists, an edge is added from the vertex of the callback function to the vertex of the synchronous call. The path for the example program is trivial since the callback function `handle_call/3` is also the caller function of `gen_server:call/3`. The wait-for graph for this program is shown in Fig 4.

The final phase of the analysis uses the wait-for graph in order to detect behaviour deadlocks. A program might deadlock if and only if there is a directed

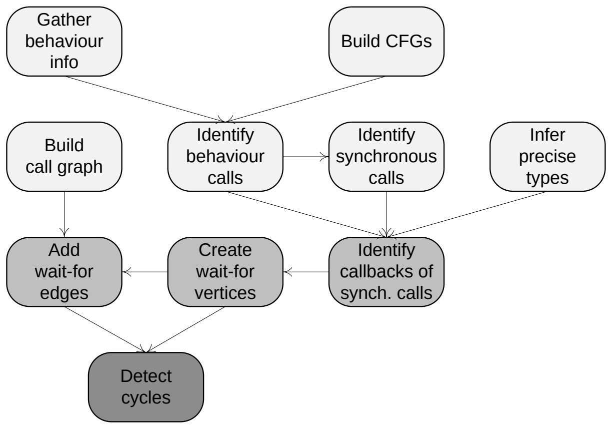

Fig. 5. Dependency graph for behaviour deadlock analysis

cycle in its wait-for graph. Consequently, the analysis searches the graph for the existence of cycles. If such cycles exist, it reports the synchronous calls that are mutually waiting for each other. As expected, the wait-for graph of Fig. 4 has a cyclic wait that involves the `gen_server:call/2,3` calls.

A dependency graph for the behaviour deadlock analysis is shown in Fig. 5. The phases of the analysis are indicated with different colors. Let us now evaluate the effectiveness of these techniques on a suite of large, widely used Erlang applications.

5 Experimental Evaluation

The analysis we described in the previous section has been fully implemented and incorporated in the development version of dialyzer. We have paid special attention to integrate it smoothly with the existing analyses, reuse as much of the underlying infrastructure as possible, and fine-tune the analysis so that it incurs relatively little additional overhead to dialyzer's default mode of use. The core of the deadlock analysis is about 2,500 lines of Erlang code and the user can turn it on either via a GUI button or a command-line option.

We have measured the effectiveness and performance of the analysis by applying it on a corpus of Erlang code bases of significant size; in total more than a million lines of code.[3] As these code bases have been developed and tested over a long period of time, it is perhaps not surprising that our analysis did not find deadlocks in most of them. In fact, many Erlang developers admit to having run into and corrected such errors. An indicative example is that of Reia[4], a hybrid

[3] The source of Erlang/OTP distribution alone is about 800k lines of code.

[4] `http://james-iry.blogspot.com/2009/04/erlang-style-actors-are-all-about_16.html`

object/actor language for the Erlang VM: at some point during its implementation, there was a behaviour deadlock involving two generic servers synchronously calling each other that was later eliminated. Still, there are Erlang/OTP libraries and applications for which the analysis has detected possible deadlocks currently present in their code. A short description of these code bases appears in Table 1; most are heavily used and reasonably well-tested. For open source applications, we used the code from their public repositories at the end of March 2011.

Table 1. Applications for which the analysis detected deadlocks

Application libraries from the Erlang/OTP R14B02 distribution	
snmp	Simple Network Management Protocol
Open source Erlang applications	
dynomite	A Dynamo clone
effigy	A mocking library for testing
log_roller	A distributed logging system
yatsy	Yet Another Test Server — Yaws compatible
zotonic	A content management system

Table 2. Effectiveness of the deadlock analysis

		Errors	
Application	LOC	CD	BD
snmp	56,728	-	1
dynomite	19,381	1	-
effigy	1,288	1	-
log_roller	2,539	1	-
yatsy	2,356	-	23
zotonic	68,462	-	3

Table 2 shows the lines of code (LOC) for each application and the number of blocking program points identified by the analysis. These are shown categorized as in Sect. 3: namely, as related to a communication deadlock (CD) because they involve a blocking `receive`, or a behaviour deadlock (BD) because there is a synchronous call that will either timeout or wait forever. As can be seen in the table, the analysis detects a number of errors, that may be detrimental to the functionality and robustness of these applications. We have manually examined the source code of these applications and all these problems are genuine bugs under certain runtime conditions that are dependent on the program input and the chosen execution paths. More details about these errors may be found on dialyzer's website: `http://www.softlab.ntua.gr/dialyzer/`.

Regarding performance, we deliberately did not include measurements of the additional time and memory overhead of the deadlock detection component of the analysis as it is too small to care about. In fact, for Erlang applications consisting of hundreds of thousands of lines of code, time is typically a matter of a few minutes and space is usually less than a gigabyte. Given that the analysis is totally automatic and smoothly integrated in a defect detection tool that is widely used by the community, we see very little reason not to use it regularly when developing Erlang programs.

6 Related Work

The problem of detecting deadlocks in concurrent programs is fundamental and well studied. In the literature one can find various approaches either for shared-memory, distributed [10] or database systems [11]. Work on deadlock detection for the former systems includes many static approaches. Boyapati et al. [12] have presented a *type based* approach that allows programmers to specify a partial order among locks and guarantees that well-typed programs are free of data races and deadlocks. A number of *data-flow analyses* have also been proposed. Among them, approaches by von Praun [13], Williams et al. [14] and Engler and Ashcraft [15] rely on the computation of a static lock-order graph and report cycles in the graph as possible deadlocks. Similarly, our analysis for behaviour deadlock detection constructs a static graph and searches it for cycles. In distributed and database systems, most approaches are dynamic but also involve cycle detection in wait-for graphs. In these approaches however, the main points of interest are the efficiency of the cycle detection algorithms and the methods employed for the construction and maintenance of the wait-for graph.

Regarding communication, some researchers have proposed using *effect-based type systems* to analyze the communication behaviour of message passing programs; an early such work is the analysis by Nielson and Nielson for detecting when programs written in CML have a finite topology [16]. There has also been a number of *abstract interpretation based* analyses that are closer in spirit to the analysis we employ for the detection of communication deadlocks. Mercouroff designed and implemented an analysis for CSP programs with a static structure based on an approximation of the number of messages sent between processes [17]. Colby's abstract interpretation based, whole program analysis uses control paths to identify threads, possibly created at the same spawn point, and construct the interprocess communication topology of the program [18]. A more precise, but also more complex and less scalable, control-flow analysis was proposed by Martel and Gengler [19]. Contrary to what we do, in their work the accuracy of the analysis is enhanced by building finite automata that eliminate some impossible communication channels and aid in computing the possibly matching emissions for each reception, and thus the possibly received values. An interesting future direction for our analysis is to see how we can use some of these ideas to enhance the precision of our analysis without sacrificing its performance.

In 2009, Claessen et al. proposed a method to detect race conditions in Erlang programs by employing *property-based testing* using QuickCheck and a special purpose randomizing user-level scheduler for Erlang called PULSE [20]. The PULSE scheduler controls its processes by randomly picking only one of them to run at a time. As an additional benefit, this design allows PULSE to detect communication deadlocks when all of its randomly interleaved processes are blocked waiting on some `receive` and no messages are being sent to any of the blocked processes. To the best of our knowledge, this has been the only attempt to detect deadlocks in Erlang programs so far. While we prefer our method because it is more scalable and analyzes the entire program instead of random process schedules, QuickCheck and PULSE may find deadlocks that our tool would suppress for fear of emitting false alarms in case the available static information were not precise enough.

7 Concluding Remarks

We have showed kinds of deadlocks that Erlang programs can exhibit and have presented a static analysis that detects them. Our analysis is fast, robust and uses effective techniques to achieve a proper balance between precision and performance. By implementing this analysis in a publicly available and commonly used tool for detecting software defects in Erlang programs, we were able to detect a number of previously unknown deadlocks in widely used and reasonably well-tested applications written in Erlang, as shown in the experimental evaluation section of the paper. By identifying kinds of possible deadlocks in Erlang programs, we also contribute in a concrete way to raising the awareness of the Erlang programming community on these errors.

In the future, we hope that Erlang developers will be watching out for these errors when programming. We also plan for our analysis to be included in an upcoming release of Erlang/OTP, thus acquiring its place in the developer's tool suite. In a wider perspective, we are interested in evaluating such an analysis in functional programming languages with similar concurrency features as Erlang.

References

1. Armstrong, J.: Programming Erlang: Software for a Concurrent World. The Pragmatic Bookshelf, Raleigh, NC (2007)
2. Lindahl, T., Sagonas, K.: Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin, W.N., ed.: Programming Languages and Systems: Proceedings of the Second Asian Symposium. Volume 3302 of LNCS., Berlin, Germany, Springer (2004) 91–106
3. Sagonas, K.: Experience from developing the Dialyzer: A static analysis tool detecting defects in Erlang applications. In: Proceedings of the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools. (2005)

4. Christakis, M., Sagonas, K.: Static detection of race conditions in Erlang. In Carro, M., Peña, R., eds.: Practical Aspects of Declarative Languages: Proceedings of the PADL Symposium. Volume 5937 of LNCS., Berlin, Germany, Springer (January 2010) 119–133
5. Christakis, M., Sagonas, K.: Detection of asynchronous message passing errors using static analysis. In Rocha, R., Launchbury, J., eds.: Practical Aspects of Declarative Languages: Proceedings of the PADL Symposium. Volume 6539 of LNCS., Berlin, Germany, Springer (January 2011) 5–18
6. Nagy, T., Nagyné Víg, A.: Erlang testing and tools survey. In: Proceedings of the 7th ACM SIGPLAN Workshop on Erlang, New York, NY, USA, ACM (2008) 21–28
7. Carlsson, R.: An introduction to Core Erlang. In: Proceedings of the PLI'01 Workshop on Erlang. (2001)
8. Carlsson, R., Sagonas, K., Wilhelmsson, J.: Message analysis for concurrent programs using message passing. ACM Transactions on Programming Languages and Systems **28**(4) (July 2006) 715–746
9. Lindahl, T., Sagonas, K.: Practical type inference based on success typings. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, New York, NY, USA, ACM (2006) 167–178
10. Singhal, M.: Deadlock detection in distributed systems. Computer **22** (1989) 37–48
11. Knapp, E.: Deadlock detection in distributed databases. ACM Computing Surveys **19** (December 1987) 303–328
12. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '02, New York, NY, USA, ACM (2002) 211–230
13. von Praun, C.: Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland (February 2004)
14. Williams, A., Thies, W., Ernst, M.D.: Static deadlock detection for Java libraries. In: ECOOP 2005 — Object-Oriented Programming, 19th European Conference, Glasgow, Scotland (July 2005) 602–629
15. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, New York, NY, USA, ACM (2003) 237–252
16. Nielson, F., Nielson, H.R.: Higher-Order Concurrent Programs with Finite Communication Topology. In: Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages, New York, NY, USA, ACM (1994) 84–97
17. Mercouroff, N.: An algorithm for analyzing communicating processes. In: Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics, London, UK, Springer-Verlag (1992) 312–325
18. Colby, C.: Analyzing the Communication Topology of Concurrent Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, New York, NY, USA, ACM (1995) 202–213
19. Martel, M., Gengler, M.: Communication Topology Analysis for Concurrent Programs. In: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. Volume 1885 of LNCS., Heidelberg, Springer (2000) 265–286

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