View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Kent Academic Repository

Kent Academic Repository

Full text document (pdf)

Citation for published version

Christakis, Maria and Sagonas, Konstantinos (2011) Detection of Asynchronous Message Passing Errors Using Static Analysis. In: Practical Aspects of Declarative Languages. Lecture Notes in Computer Science. pp. 5-18. ISBN 978-3-642-18377-5.

DOI

http://doi.org/10.1007/978-3-642-18378-2_3

Link to record in KAR

http://kar.kent.ac.uk/58950/

Document Version

Author's Accepted Manuscript

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

Detection of Asynchronous Message Passing Errors Using Static Analysis

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. Concurrent programming is hard and prone to subtle errors. In this paper we present 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. Our analysis is completely automatic, fast, and strikes a proper balance between soundness and completeness: it is effective in detecting errors and avoids false alarms by computing a close approximation of the interprocess communication topology of programs. We have integrated our analysis in dialyzer, a widely used tool for detecting software defects in Erlang programs, and demonstrate its effectiveness on libraries and applications of considerable size. Despite the fact that these applications have been developed over a long period of time and are reasonably well-tested, our analysis has managed to detect a significant number of previously unknown message passing errors in their code.

1 Introduction

Concurrent execution of programs is more or less a necessity these days. To cater for this need, most programming languages come with built-in support for creating processes or threads. Depending on the concurrency model of the language, interprocess communication takes place through synchronized shared structures (as in C/Pthreads, Java and Haskell), synchronous message passing on typed channels (as in Concurrent ML), or asynchronous message passing (as in Erlang). Even though certain problems associated with concurrent execution of programs are completely avoided in some of these models, each of them comes with its own set of gotchas and possibilities for programming errors. Independently of the concurrency model which is employed by the language, concurrent programming is fundamentally more difficult than its sequential counterpart.

Tools that detect software errors early in the development cycle can help in making concurrent programming more robust and easier for programmers. In particular, tools based on static analysis seem promising as they are completely automatic and in principle scale better than, for example, those based on model checking. Unfortunately, designing and implementing an effective static analysis for a concurrent language which has not been designed with analysis in mind is a

R. Rocha and J. Launchbury (Eds.): PADL 2011, LNCS 6539, pp. 5–18, 2011.
© Springer-Verlag Berlin Heidelberg 2011

challenging task. For example, in a language based on processes communicating using asynchronous message passing such as Erlang, it is possible to create an unbounded number of processes, send any term as a message, communicate with processes located on any machine, local or remote, selectively retrieve messages from a process' mailbox using pattern matching, monitor other processes and register to receive their messages when they die, etc. On top of all that, the language is dynamically typed and higher-order, which makes the task of computing precise type and control-flow information very difficult, if not impossible.

In the context of such a real-world language, we aim to statically detect errors that arise from the use of asynchronous message passing. To do so, we have designed an effective analysis that determines the interprocess communication topology of Erlang programs, discovers which occurrences of the sending primitives match which occurrences of the receiving primitives, and emits warnings accordingly. 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 and requires no guidance from its user; 2) strikes a proper balance between soundness and completeness in order to be: 3) fast and scalable. As we will soon see, we have achieved these goals.

The contributions of our work are as follows:

- we document some of the most important kinds of errors associated with concurrency via asynchronous message passing;
- we present an effective and scalable analysis that detects these errors, and
- we demonstrate the effectiveness of our analysis on a set of widely used and reasonably well-tested libraries and open source applications by reporting a number of previously unknown message passing errors in their code bases.

The next section overviews the Erlang language and the defect detection tool which is the implementation platform for our work. Sect. 3 describes commonly occurring kinds of message passing errors in Erlang programs, followed by Sect. 4 which presents in detail the analysis we use to detect them. The effectiveness and performance of our analysis is evaluated in Sect. 5 and 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. The main implementation of the language, the Erlang/OTP (Open Telecom Platform) system from Ericsson, has been open source since 1998 and 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 most other programming languages out there 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 unresponsive processes when things go wrong.

Since 2007, the Erlang/OTP distribution includes a static analysis tool, called dialyzer [2,3], for finding software defects (such as type errors, exception-raising code, code which has become unreachable due to some logical error, etc.) in single Erlang modules or entire applications. 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. In 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]. Before we describe how we extended dialyzer's analyses to also detect message passing errors, let us first see how concurrency with asynchronous message passing works and the kinds of related defects that may exist in Erlang programs.

3 Message Passing in Erlang

As described in Sect. 2, Erlang's concurrency primitives `spawn`, `!` (send) and `receive` allow a process to spawn new processes and communicate with others through asynchronous message passing. Let's see these primitives in 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 arguments of the `spawn`. In the example program shown in Fig. 1, a process is spawned that will execute the code of the function closure `Fun`.

```
-export([hello_world/0]).

hello_world() ->
  Fun = fun() -> world(self()) end,
  Pid = spawn(Fun),
  register(world, Pid),
  world ! hello.

world(Parent) ->
  receive
    hello -> Parent ! hi
  end.
```

Fig. 1. Simple example program

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. Besides addressing a process by using its pid, there is also a mechanism, called the *process registry*, which acts as a node-local name server, for registering a process under a certain name so that messages can be sent to the process using that name. Names of processes are currently restricted to atoms. In our example program, the spawned process is registered under the name `world` which is then used to send the message `hello` to the process.

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

Misuse of these concurrency and communication primitives may lead to the following kinds of message passing errors in Erlang programs:

Receive with no messages (RN). A `receive` statement in the code executed by some process whose mailbox will be empty. This defect could reveal the occurrence of possible deadlocks in the patterns of interprocess communication — processes mutually waiting for messages from other processes.

Receive of the wrong kind (RW). A `receive` statement in the code of some process whose mailbox will contain messages of different kinds than the ones expected by the `receive`. Currently, such a defect can have devastating effects on a running system, overflowing the mailbox of some process and bringing the node down. To avoid being bitten by this, many Erlang programs adopt a defensive programming style and include a catch-all clause in `receive`s whose only purpose is to consume any unwanted messages.

This practice is not ideal because it might hide real communication problems. Additionally, it makes this kind of message passing errors hard to find.

Receive with unnecessary patterns (RU). A `receive` with clauses containing patterns that will never match messages sent to the process executing that code. This problem may be harmless (i.e., just some unreachable code in the `receive`) or, in conjunction with the existence of a catch-all pattern which consumes all messages as the last clause of the statement, may hide a serious functionality error.

Send nowhere received (SR). A send operation to a process whose code does not contain any (matching) `receive`s. This defect can also result in the overflow of some mailbox and bring a node down.

Being able to statically detect such types of concurrency defects is crucial in safety-critical systems such as those developed in the telecommunications sector.

4 The Analysis

In a higher-order language with unlimited dynamic process creation, the kinds of message passing errors we described in the previous section are not simple to detect. In order to detect which message emissions match which receptions, it is necessary to determine the communication topology of processes, which will then be used as a basis for detecting these errors. We have designed and implemented such an analysis and describe it in this section.

Conceptually, our analysis 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 message passing errors are detected. For efficiency reasons, the actual implementation blurs the lines separating these phases and employs some optimizations. False alarms are avoided by taking language characteristics and messages generated by the runtime system into account. Let's see all these in detail.

4.1 Collecting Information

We have integrated our analysis in dialyzer because many of the components that it relies upon were already available or could be easily extended to provide the information that the analysis needs. The analysis starts 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 [5], 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 which makes the binding occurrence and scope of all variables explicit.

As the source code is translated to Core Erlang, dialyzer constructs the *control-flow graph* (CFG) of each function and function closure and then uses the escape analysis of Carlsson et al. [6] to determine values, in particular closures, that escape their defining function. For example, for the code of Fig. 1 the escape

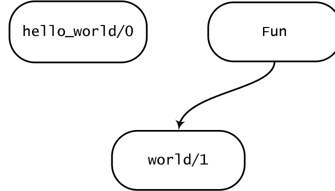

Fig. 2. Call graph of example program

analysis will determine that function `hello_world` defines a function closure that escapes from this function as it is passed as argument to a `spawn`. 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. For the example in the same figure, the call graph will contain three nodes for functions whose definitions appear in the code (functions `hello_world`, `world`, and the closure) and an edge from the node of the function closure to that of `world`, as shown in Fig. 2.

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 data item 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. For example, different atoms $a_1, \ldots, a_n$ are represented as different *singleton types* in the type domain and their union $a_1 | \ldots | a_n$ is mapped to the supertype *atom*() only when the size of the union exceeds a relatively high limit [7]. We will see how this information is used by the message analysis in Sect. 4.2 and 4.3.

4.2 Constructing the Communication Graph

The second phase of the analysis determines the interprocess communication topology in the form of a graph.

Each vertex of the graph represents an escaping function 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. The code of any function that is either a root node in the call graph or an argument to a `spawn` is assumed to be executed by a separate process. For our example program, the communication graph will contain two nodes, one for function `hello_world` and one 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.

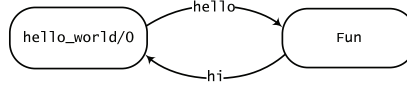

Fig. 3. Communication graph of example program

To this end, we start by traversing the CFGs of the functions corresponding to the vertices in the communication graph using depth-first search. The depth-first search starts by identifying program points containing a call to a pid-yielding primitive (i.e., the `self` primitive, that returns the pid of the calling process, and the `spawn` family of primitives), and then tries to find program points "deeper" in the graph containing send operations to the process with this pid. In case the search encounters either a call to or a `spawn` of some other statically known function, the traversal continues by examining its CFG, otherwise it is ignored. Built-ins for registering a pid under a certain name require special attention since the registered name may then be used to send a message to the process. A pre-processing step associates names with their registered pids so that the analysis can use this information to replace all name occurrences. Finally, if the traversal finds a send operation to some pid, the analysis takes variable sharing into account to determine whether this pid refers to the same process as the pid yielded by the primitive that initially triggered the depth-first search. If this is the case, an edge is added to the communication graph emanating from the vertex of the process whose CFG is traversed and incident on the vertex of the process identified by the pid, otherwise nothing is done. An annotation is added to the new edge indicating the type information of the message. If such an edge already exists in the graph, then the analysis simply updates its annotation to also include the type information of the new message. In the end, this traversal creates the complete set of edges in the communication graph.

For the code of Fig. 1, the communication graph will have two edges, one from vertex `hello_world` to the closure and one from the closure to `hello_world`. The annotations for these edges will be `hello` and `hi`, respectively. The communication graph for the example program is illustrated in Fig. 3.

4.3 Detecting Message Passing Errors

At this stage of the analysis, the CFG of each function that corresponds to a vertex in the communication graph is traversed anew to detect any message passing errors.

Each vertex in the communication graph has an in-degree that is either equal to or greater than zero. A vertex 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 does not expect any messages (i.e., there are no `receive`s in the CFG), a warning is emitted for each sent message. In case the process expects to receive messages,

the analysis takes into account the type information of the messages and the `receive` patterns in order to decide whether they match. A message S matches a `receive` pattern R if the *infimum* (i.e., the greatest lower bound) of their type information is a non-empty subtype of R. Note that S is the annotation of the edge in the communication graph, while R is found in the CFG. As an example consider a sent tuple message with type $S :: \{\texttt{gazonk}, \mathit{integer}()\}$ and a `receive` pattern with type $R :: \{\mathit{atom}(), \texttt{42}\}$. The analysis computes the infimum of these types, $\{\texttt{gazonk}, \texttt{42}\}$, which is a subtype of R in this case. Actually, this message will only be received if the second element of the tuple is `42`, but the analysis, aiming at being sound for defect detection, will flag this as an error only if it can statically determine that the second element of the message is a term other than the integer `42`. In short, at the end of the CFG traversal, warnings are emitted for `receive` patterns or entire constructs that do not match any messages and for messages that do not match any `receive` patterns.

Loops in the communication graph indicate that messages are sent and received by the same process and require special treatment. If no messages sent by other processes match a `receive` pattern or construct, then messages sent to the process by itself at program points "higher" in the CFG should match, otherwise a warning is emitted.

Note that the traversal that searches for `receive`s, unlike the traversal that searches for send operations described in the previous section, ignores any `spawn`s of statically known functions since spawned processes cannot receive messages in place of the process being analyzed, although they may send messages to it.

For the example program, the analysis inspects the CFG of the `hello_world` vertex, which has in-degree one, and finds that there is no `receive` in the code executed by the process. Consequently, it emits a warning with the filename and line number of the send operation of the `hi` message reporting that this message will be nowhere received.

4.4 Some Optimizations

Although we have described the second and third phases of the analysis as being distinct, our implementation blurs this distinction, thereby avoiding redundant searches and speeding up the analysis. In addition, we also employ the following optimizations:

Control-flow graph minimization. The CFGs that dialyzer constructs by default contain the complete Core Erlang code of functions. This makes sense as most of its analyses, including the type and sharing analyses, need this information. However, note that the path traversal procedure of Sect. 4.2 and 4.3 requires only part of this information. For example, in the program illustrated on the left box of Fig. 4, the `io:format` call is irrelevant both for determining the complete set of edges in the communication graph and for detecting any message passing errors. Our analysis takes advantage of this by a pre-processing step that removes all this code from the CFGs and by recursively removing CFGs of *leaf* functions that do not contain any concurrency primitives either directly or indirectly.

Avoiding repeated traversals. After the CFGs are minimized as described above, the depth-first traversal starts from some vertex in the communication graph. The traversal of all paths from this vertex often encounters a split in the CFG (e.g., a point where a `case` statement begins) which is followed by a CFG join (the point where the `case` statement ends). All the straight-line code which lies between the join point and the next split, including any straight-line code in the CFGs of functions called there, does not need to be repeatedly traversed if it is found to contain no concurrency primitives during the traversal of its first depth-first search path. This optimization effectively prunes common sub-paths by condensing them to a single program point.

Avoiding redundant traversals. Another optimization is to collect, during the construction of the CFGs of functions, a set of program points containing send operations and another set of program points containing `receive` constructs. The first set is used in the construction of the communication graph to determine whether the CFG of a statically known function that is either called or spawned needs to be inspected. If no program point in the set is reachable from the function directly or indirectly (i.e., via some call or `spawn`), then the CFG is not traversed. The elements of the second set act like pointers and replace the vertices of the communication graph in the error detection phase of the analysis, thereby avoiding unnecessary traversals of the control flow graphs.

4.5 False Alarms and Their Avoidance

The analysis we have described so far may produce false alarms in case the available static information is too limited to construct the exact interprocess communication graph. To this effect, we employ techniques for completely avoiding false alarms, thus making the analysis sound for defect detection.

A factor that could limit the precision of our analysis is lack of precise knowledge about the behaviour of built-in functions (BIFs). For example, Erlang/OTP comes with BIFs, implemented in C, that create messages inside the VM and send them to processes in a non-transparent way. The left box of Fig. 4 shows a function from the code of the ibrowse application (file `ibrowse_test.erl`). On this code, a naïve implementation of the analysis would warn that the {`'DOWN', ...`} pattern of the `receive` statement is unused because no such messages are ever constructed in the entire application, let alone in this module (whose code is only partly shown in the figure). However, such messages are created by the `spawn_monitor` BIF inside the VM and are placed in the message queue of the monitoring process when the spawned process dies. We have taken special care to provide our analysis with precise information about such BIFs and their behaviour. This information is fairly complete at this point so we entirely avoid this kind of false alarms. For similar reasons, the analysis can either have *a priori* knowledge about the behaviour of heavily used Erlang/OTP libraries, or pre-compute this information so as to avoid having to re-analyze these libraries in each run.

Another limiting factor is dialyzer's sharing/alias analysis component. Since the computation of variables referring to the same data is static, it may not

```
unit_tests(Opts) ->
  Opts1 = Options ++ [{connect_timeout, 5000}],
  {Pid, Ref} = spawn_monitor(?MODULE, ut1, [self(), Opts1]),
  receive
    {done, Pid} -> ok;
    {'DOWN', Ref, _, _, Info} ->
      io:format("Crashed: ~p~n", [Info])
  after 60000 ->
    ...
  end.

ut1(Parent, Opts) ->
  lists:foreach(...),
  Parent ! {done, self()}.  % the only send operation
```

```
-export([start/0]).

start() ->
  Pid = spawn(fun pong/0),
  ping(Pid).

ping(Pid) ->
  Pid ! {self(), ping},
  receive pong -> pang end.

pong() ->
  receive
    {Pid, ping} -> Pid ! pong
  end.
```

Fig. 4. Programs susceptible to false alarms

always be possible to find the complete sets of these variables. The right box of Fig. 4 shows a made up example of Erlang code for which the first implementations of our analysis incorrectly warned that the `receive` statement in the `ping` function would block. This false alarm was emitted because the sharing/alias analysis is unable to statically determine whether the {`self(), ping`} message will actually be received by the `pong` process. The analysis was therefore unable to conclude that the `Pid` variable in the received message is the pid of the `start` process. This was also the case when the analysis lost track of terms because data was stored in data structures (usually records, lists or ETS tables) and then retrieved from them. Again, we have taken special care to avoid these false alarms by acknowledging that the sharing/alias component has lost the data item — specifically the pid — assigned to a variable and thereby suppressing any warnings that would be emitted as a result of this inaccuracy.

Clearly, the optimization ideas and the techniques to avoid false alarms have a heavy impact on the effectiveness, performance and precision of our method. Let us therefore evaluate it 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 message analysis is about 2,000 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.[1] 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

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

Table 1. Applications for which the analysis detected message passing errors

Application libraries from the Erlang/OTP R14A distribution	
inets	A set of Internet clients and servers
observer	Tools for tracing and investigating distributed systems
Open source Erlang applications	
disco	A map/reduce framework for distributed computing
dynomite	A Dynamo clone
effigy	A mocking library for testing
eldap	An LDAP API
enet	A network stack
erlang_js	A driver for SpiderMonkey (the Mozilla JavaScript engine)
etap	A TAP (Test Anything Protocol) client library
iserve	An HTTP server
log_roller	A distributed logging system
natter	An XMPP client
pgsql	A PostgreSQL driver
stoplight	A mutex server based on the SIGMA algorithm
ubf	Universal binary format

find errors in most of them. Still, there are Erlang/OTP libraries and applications for which the analysis has detected concurrency errors in their code. The rest of this section focusses on these code bases. A short description of them 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 October 2010.

The left part of Table 2 shows the lines of code (LOC) for each application and the number of message passing problems identified by the analysis. These are shown categorized as in Sect. 3: namely, as related to a `receive` that will block either because no messages are sent to the process (RN) or because the messages sent there are of the wrong kind (RW), as related to a `receive` with unnecessary patterns (RU), or related to a send operation to a process without a `receive` (SR). The right part of the table shows the elapsed wall clock time (in seconds), and memory requirements (in MB) for running dialyzer without and with the analysis component that detects message passing errors in these programs.[2] The evaluation was conducted on a machine with an Intel Core2 Quad CPU @ 2.66GHz with 3GB of RAM, running Linux. (But currently the analysis utilizes only one core.)

As can be seen in the table, the analysis detects a number of message passing problems, some of which can be detrimental to the functionality and

[2] The relatively high memory requirements of the enet application are due to an (automatically generated?) file containing just two functions of about 10,000 LOC each. When excluding this file, dialyzer needs 2.0 secs and 74MB in its default mode and 2.4 secs and 75MB with the analysis that detects message passing errors.

Table 2. Effectiveness and performance of the message analysis

Application	LOC	Errors				Time (secs)		Space (MB)	
		RN	RW	RU	SR	w/o msg	w msg	w/o msg	w msg
inets	29,389	-	-	2	-	26.1	60.1	89	119
observer	6,644	-	-	1	-	23.6	35.1	78	88
disco	11,846	-	-	2	-	17.3	20.8	85	126
dynomite	19,384	1	-	-	-	7.2	7.9	72	74
effigy	568	1	-	-	-	0.8	0.9	21	21
eldap	5,148	-	-	1	-	9.9	11.5	110	112
enet	23,028	-	-	1	-	15.0	15.8	765	766
erlang_js	1,720	-	-	1	-	11.0	12.2	73	80
etap	665	-	-	-	1	0.4	0.4	17	19
iserve	788	-	-	2	2	1.3	1.4	30	30
log_roller	2,539	-	1	-	-	3.1	3.5	33	46
natter	1,494	-	-	2	-	1.7	1.9	30	32
pgsql	1,253	-	-	1	-	1.6	3.0	31	41
stoplight	1,462	1	-	-	-	1.6	1.6	39	40
ubf	7,052	-	-	1	-	18.7	23.7	70	81

robustness of these applications. We have manually examined the source code of these applications and all these problems are genuine bugs.

Regarding performance, in most cases, the additional time and memory overhead of the message passing error detection component of the analysis is too small to care about. The only exception is inets on which the analysis takes about twice as much time to complete. Still the analysis times are reasonable. Given that the analysis is totally automatic and smoothly integrated in a defect detection tool which is widely used by the community, we see very little reason not to use it regularly when developing Erlang programs.

6 Related Work

Static analysis [8] is a fundamental and well studied technique for discovering program properties and reasoning about program behaviour, independently of language. Besides being the basis for most compiler optimizations, in recent years static analysis has been extensively used to detect software errors in programs, both sequential and concurrent.

In the context of higher-order functional languages, and starting with the work of Shivers [9], control-flow analyses aim to approximate which functions may be applied at runtime as a result of some computation. When concurrency comes into the picture, processes are dynamically created and functions are passed between processes and executed by any receiving process, the task becomes more complicated as a piece of code in the source program may be executed by any process and the control-flow analysis may need to be infinitary [10].

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 finite topology [11]. There has also been a number of *abstract interpretation based* analyses that are closer in spirit to the analysis we employ. 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 [12] and Martel and Gengler an analysis that statically determines an approximation of the communication topology of a CML program [13]. The abstract interpretation based whole program analysis of Colby uses control paths to identify threads [14]. Unlike earlier work which collapsed multiple threads created at the same spawn point to a single approximate thread, control paths are able to distinguish multiple threads created at the same spawn point and thus compute a more precise interprocess communication topology of a program. Still, the precision problem was not completely solved.

A more precise, but also more complex and less scalable, control-flow analysis was proposed by Martel and Gengler [13]. Contrary to what we do, in their work the accuracy of the analysis is enhanced by building finite automata. More specifically, the analysis orders the synchronization primitives of any sequential processes in the system by building an automaton for each process. It then approximates how the different processes may interact with each other by building a reduced product automaton from the process automata. As a result, the analysis eliminates some impossible communication channels, computes 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 soundness for defect detection (i.e., its "no false alarms" property) or its scalability.

7 Concluding Remarks and Future Work

We have presented a new analysis for identifying some commonly occurring kinds of concurrency errors that arise from the use of asynchronous message passing in a higher-order language with unlimited process creation, message queues and selective message reception based on pattern matching. By computing a close approximation of the interprocess communication topology of programs and by effectively matching occurrences of send with `receive` primitives, our analysis manages to achieve a good balance between precision and scalability. As shown in the experimental evaluation section of the paper, the analysis has managed to detect a significant number of message passing errors in widely used and reasonably well-tested applications written in Erlang.

The implementation of our analysis is fast and robust; we expect that it will be included in some upcoming release of Erlang/OTP. Still, there are some engineering issues to address. Among them, the most challenging one is to design and implement a framework for the explanation of message passing errors, perhaps by maintaining more information in the analysis and designing a component to

visualize the communication topology of an application. But this is a general problem for static analyses that detect program errors: the more sophisticated the errors that an analysis detects are, the more difficult it is for programmers to trust the analysis results and, more importantly, to reason about the program change that will correct the error. Tools that help them in this task are needed.

References

1. Armstrong, J.: Programming Erlang: Software for a Concurrent World. The Pragmatic Bookshelf, Raleigh (2007)
2. Lindahl, T., Sagonas, K.: Detecting software defects in telecom applications through lightweight static analysis: A war story. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 91–106. Springer, Heidelberg (2004)
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.) PADL 2010. LNCS, vol. 5937, pp. 119–133. Springer, Heidelberg (2010)
5. Carlsson, R.: An introduction to Core Erlang. In: Proceedings of the PLI 2001 Workshop on Erlang (2001)
6. Carlsson, R., Sagonas, K., Wilhelmsson, J.: Message analysis for concurrent programs using message passing. ACM Transactions on Programming Languages and Systems 28(4), 715–746 (2006)
7. 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, pp. 167–178. ACM, New York (2006)
8. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus (1999)
9. Shivers, O.: Control Flow Analysis in Scheme. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 164–174. ACM, New York (1988)
10. Nielson, F., Nielson, H.R.: Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 332–345. ACM, New York (1997)
11. 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, pp. 84–97. ACM, New York (1994)
12. Mercouroff, N.: An Algorithm for Analyzing Communicating Processes. In: Schmidt, D., Main, M.G., Melton, A.C., Mislove, M.W., Brookes, S.D. (eds.) MFPS 1991. LNCS, vol. 598, pp. 312–325. Springer, Heidelberg (1992)
13. Martel, M., Gengler, M.: Communication Topology Analysis for Concurrent Programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 265–286. Springer, Heidelberg (2000)
14. Colby, C.: Analyzing the Communication Topology of Concurrent Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 202–213. ACM, New York (1995)