Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications

Konstantinos Sagonas

Department of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Abstract

We describe some of our experiences from developing the Dialyzer defect detection tool and overseeing its use in large-scale commercial applications of the telecommunications industry written in Erlang. In particular, we mention design choices that in our opinion have contributed to Dialyzer's acceptance in its user community, things that have so far worked quite well in its setting, the occasional few that have not, and the lessons we learned from interacting with a wide, and often quite diverse, variety of users.

1. Introduction

Programmers occasionally make mistakes, even functional programmers do. This latter species is by choice immune to some of the more typical kinds of software defects such as buffer overruns or accessing memory which has been freed, but cannot escape many other kinds of programming errors, even the more mundane ones such as simple typos. To catch some of these errors early in the development phase, many functional programmers prefer using statically typed languages such as ML or Haskell. These languages impose a static type discipline on programs and report obvious type violations during compilation. Static typing is not a panacea and does have some drawbacks. First of all, the errors that are caught are limited by the power of the currently employed type system; for example, none of the employed type systems statically catches division by zero errors or using a negative integer as an array index. Another drawback is that static typing often imposes quite stringent rules on what is considered a type-correct program (for example, type systems often require that each variable has a type which can be uniquely determined by constructors) and forces a program development model with a fair amount of constraints (e.g., ML requires that the module structure of an application is hierarchical and that there are no calls to functions with unknown type signatures).

Mainly due to reasons such as those described above, some programmers feel more at ease practicing a different religion. They adopt a more *laissez-faire* style of programming and choose to program in dynamically typed functional languages, like Lisp or Scheme, instead. Erlang [1] is such a language. In fact, it is not only dynamically typed but it also extends pattern matching by allowing type guards in function heads and in case statements. It is also a concurrent language which is used by large companies in the telecommunications industry to develop large-scale (several hundred thousand lines of code) commercial applications.

The Erlang/OTP development environment Since defect detection tools are only additional weapons in the war against software bugs, we briefly describe the surroundings of our tool. The development environment of the Erlang/OTP system from Ericsson[1] strongly encourages rapid prototyping and performing unit testing early on in the development cycle. Like many functional language implementations, the Erlang/OTP system comes with an interactive shell where Erlang modules can be loaded and the functions in them can easily be tested on an individual basis by simply issuing calls to them. If an exception occurs at any point, it is caught and presented to the user together with a stack trace which shows the sequence of calls leading to the exception. Many errors are eliminated in this way. Of course, testing of multi-thousand (and often million) LOC commercial applications such as e.g. the software of telecom switches is not limited to unit testing to catch exceptions but is much more thorough and systematic; for one such example see [7]. However, testing, no matter how thorough, cannot of course detect all software defects. Tools that complement testing, such as static analyzers, have their place in software development regardless of language. Erlang is no exception to this.

2. Dialyzer: A brief overview

The Dialyzer [5] is a lightweight static analysis tool that identifies some software defects such as obvious type errors, unreachable code, redundant tests, virtual machine bytecode which is unsafe, etc. in single Erlang modules or entire applications. Because not all of defects identified by Dialyzer are software bugs, we thereafter collectively refer to them as code *discrepancies*.[2]

Dialyzer starts its analysis either from Erlang source code or from the virtual machine bytecode that the Erlang/OTP compiler has produced and reports to its user the functions where the discrepancies occur and an indication of what each discrepancy is about.

Characteristics Notable characteristics of Dialyzer are:

- Currently Dialyzer is *a push-button technology and completely automatic*. In particular it accepts Erlang code "as is" and does not require any annotations from the user, it is very easy to customize, and supports various modes of operation (GUI vs. command-line, module-local vs. application-global analysis, using analyses of different power, focusing on certain types of discrepancies only, etc.)
- Dialyzer is a *complete* defect detector — though of course not guaranteed to find all errors — in the sense that it does not report any false positives; more on this below.
- Its basic analysis is typically *quite fast*. On a 2GHz Pentium 4 laptop it "dialyzes" about 800 lines of Erlang code per second.

Basic functionality explained using an example The simplest way of using Dialyzer is via the command line. The command:

```
dialyzer --src -r dir
```

[1] OTP stands for Open Telecom Platform; see `www.erlang.org`.

[2] DIALYZER stands for DIscrepancy AnaLYZer of ERlang programs.

will find, recursively, all `.erl` Erlang source files under `dir` and will collectively analyze them for discrepancies. (The `--src` option is needed because, for historical reasons, analysis starts from virtual machine bytecode by default. The command with the `--src` option omitted, will analyze all `.beam` bytecode files under `dir`.)

We illustrate some of the kinds of discrepancies that Dialyzer is capable of identifying by the following, quite factitious, example. Assume that we are analyzing a bunch of modules, among them `m1` and `m2`, and that `m2` contains a function `bar` (denoted `m2:bar`) called by function `foo` in module `m1`. Because Dialyzer constructs the inter-modular function dependency graph, function `m2:bar` will be analyzed first. In doing so, let us assume that type inference determines that function `m2:bar`, when not throwing an exception, returns a result of the following type:

```
'gazonk' | {'ok', 0 | 42 | 'aaa' | [{'ok',_}]}
```

i.e., its result is either the atom `'gazonk'` or a pair (i.e., a 2-tuple) whose first element is the atom `'ok'` and its second element is either the integer `0`, or the integer `42`, or the atom `'aaa'`, or a list of pairs whose first element is the atom `'ok'`. (The use of an underscore in the second element denotes the universal type *any*.)

First of all, note that this is a type which will not be derived by the inferencers that statically typed language commonly employ. Type inferencers like those of e.g. ML would typically collapse the integers `0` and `42` to the built-in *integer* type and would not allow mixing primitive types such as integers and atoms without having them wrapped in appropriate constructors. More importantly, they would never derive an unconstrained type (such as the type *any*) at some position.

So, how come Dialyzer's type inferencer comes up with the *any* type for the second element of pairs in the list? This can happen for various reasons:

- The most common reason is that the source code of function `m2:bar` does not contain enough information to determine the type of the second element of these pairs.[3] It may indeed be the case that the function is polymorphic in this position, or more likely that this position is not constrained by information derived by or supplied to the analysis. The latter can happen if the second element of these pairs is manipulated by a function in some other module `m3` that Dialyzer knows nothing about because `m3` was not included in the analysis. (Type analyzers for statically typed languages would never tolerate this situation and simply give up here.)
- The analysis has decided to over-approximate, through *widening*, the inferred type. This can happen either to ensure termination or for performance reasons.

Given the return type for `m2:bar` shown above, when analyzing the code of function `m1:foo` — shown with numbered discrepancies as Program 1 — Dialyzer will report the following:

1. The call to the built-in function `list_to_atom`, if reached, will raise a runtime exception since it is called with an argument which is an atom rather than a list. (The programmer is obviously confused here; for example, perhaps the intention was to use the function `atom_to_list` instead.) In a similar manner, type clashes in calling other analyzed functions which are not necessarily language built-ins (e.g. `m2:bar`) will be identified. This is the kind of type errors that any static type analyzer would also be able to detect.
2. The case clause guarded by `is_integer(Num), Num < 0` will never succeed because its guard will never evaluate to true. The complete case clause is thus redundant. This is something that most static type analyzers would not be able to catch, for reasons explained above. Strictly, this is not an error but having redundant code like that scattered in the program is a strong indication for programmer confusion — programmers rarely fancy writing redundant code; see also [8]. Our experience is that such discrepancies quite often indicate places where bugs may creep, or may be the remains of obsolete interfaces; perhaps some time ago `m2:bar` was returning pairs with negative integers in their second element but not anymore. Such discrepancies indicate code that can be eliminated, which often simplifies the interface between functions. Note that in this case all the callers of `m1:foo` would also have to handle the 2-tuple where the first element is `'error'` besides `'answer'`. In any case, reporting to the user that this case clause will never succeed is a true statement prompting the user for some action. In our opinion, it cannot be considered a false positive; it is not a side-effect of inaccuracy in the analysis due to e.g. over-approximation or path-insensitivity.
3. The guard `size(List)` will fail since its argument is a list and in Erlang the `size` function only accepts tuples and binaries as its argument, not lists. (The corresponding function for lists is called `length` and this is a common programming mistake.) The problem here is that this defect will remain undetected at runtime because, for good reasons, all exceptions in guard contexts are intercepted and silenced; the semantics of `when` guards in Erlang dictates this. Testing has therefore very little chance of discovering this error. (The only method is to find out that the ... code in the body of the case clause never executes).
4. The case clause with pattern `{'EXIT', R}` will never match. This may seem obvious given the return type of `m2:bar`, but it indicates another common Erlang programming error. The programmer probably intended to handle exceptions here, but forgot to protect the call to `m2:bar` with a `catch` construct; i.e., write the case statement as:

   ```
   case catch m2:bar(...) of
   ```

   which *would* then match the 2-tuple `{'EXIT', R}` in the case when `m2:bar` threw an exception. (Catch-based exceptions in Erlang are wrapped in a 2-tuple whose first element is the atom `'EXIT'` and the second element is a symbolic representation of the reason, which typically includes a detailed stack trace.)

Program 1 Code snippet which is full of discrepancies.

```
-module(m1).
⋮
foo(...) ->
  case (m2:bar(...) = Bar) of
    Atom when is_atom(Atom) ->
      ..., List = list_to_atom(Atom) 1, ...;
    {'ok', 42} ->
      {'answer', 42};
    {'ok', Num} when is_integer(Num), Num < 0 2 ->
      {'error', "Negative integer - not handled yet"};
    {'ok', [H|T] = List} when size(List) 3 > 1 ->
      ...;
    {'ok', [Bar|T]} 5 ->
      ...;
    {'EXIT', R} 4 ->
      io:format("Caught exception; reason: ~p~n", [R])
  end, % end of the case statement
⋮
```

[3] Erlang programs contain no type declarations or any explicit type information.

5. This one is subtle. The pattern P = {'ok', [Bar|T]} is actually type-correct, but the pattern matching of the term returned by `m2:bar` and assigned to the variable `Bar` will never match with the pattern P, with `Bar` being a sub-term of it. We are not aware of any type checker that would catch this error.

On the other hand, notice that Dialyzer will *not* warn that the case statement has no catch-all clause (similar to C's `default`) or that the {'ok', 0} return from `m2:bar` is not handled. This is done so as to minimize irrelevant 'noise' from the tool. In this respect, Dialyzer differs from tools such as `lint` [4].

As mentioned, the example code we just presented is factitious, albeit only slightly so. Dialyzer has yet to find a single function that contains all these discrepancies in its code at the same time, but all of them, even 5, are examples of discrepancies that Dialyzer actually found in well-tested, commonly-used code of commercial products. Besides these, Dialyzer is also capable of identifying some other software defects in Erlang programs (e.g., possibly unsafe bytecode generated by older versions of the Erlang/OTP compiler and misuses of certain language constructs), but their illustration is beyond the scope of this short experience paper.

Dialyzer is of course not guaranteed to report all software defects — not even all type errors — that an Erlang application might contain. In fact, because Dialyzer's analysis is not path-sensitive, Dialyzer currently suppresses all discrepancies that might be due to losing information when joining the analysis results from different paths. This is in contrast to other defect detection tools that do report false positives due to inaccuracies or heuristics in analyses they employ.

Dialyzer comes with an extensive set of options that allow its user to focus on certain types of discrepancies only and employ analyses of varying precision vs. time complexity trade-offs. It also comes with a graphical user interface in which the user is able to inspect the information that led to the identification of some discrepancy of interest.

For more up-to-date information, see also Dialyzer's homepage: `www.it.uu.se/research/group/hipe/dialyzer/`

3. Dialyzer's usage so far

Even before its first public release, Dialyzer was applied to relatively large code bases, both by us and more commonly by Erlang application developers. We have been working closely with developers of the AXD301 and GPRS[4] projects at Ericsson, and with a T-Mobile team in the U.K. which also uses Erlang for some of its product development. An early account of the effectiveness of an internal and significantly less powerful version of the tool appears in [5].

The first releases of Dialyzer, versions 1.0.*, featured analysis starting from virtual machine bytecode only and the tool only had a GUI mode. We were somewhat (positively) surprised to receive numerous user requests to develop a command-line version of the tool so that Dialyzer becomes more easily integrated to the purely `make`-based build environment of some projects.[5] Once this happened, we even received extensions to the tool's functionality contributed by users that made it into the next release. For example, the code that supports the `-r` (add files recursively) option was a user contribution; before that, users were forced to manually specify all files and directories to include in the analysis.

At the time of this writing, some of the code bases analyzed by Dialyzer are open-source community programs (e.g., the code of the Wings 3D subdivision graphics modeler,[6] of the Yaws web server,[7] and of the `esdl` graphical user interface library[8]). However, the majority of Dialyzer's uses are large commercial applications from the telecommunications domain. Among them is the code base of the AXD301 ATM switch consisting of about 2,000,000 lines of Erlang code, where by now Dialyzer has identified a significant number (many hundreds) of software defects that have gone unnoticed after years of extensive testing. It is also continuously being used in the Erlang/OTP R10 (release 10) system to eliminate numerous bugs that previous releases contained in some of its standard libraries. We also know that Dialyzer is being used on the code of some of Nortel's products, but we do not have any further information on it.

At least in the commercial projects, Dialyzer is typically run as part of a centralized (often nightly) build. Perhaps because of this, many Dialyzer users typically complain that Dialyzer's identification of discrepancies is not as clear and concise as the messages they are used to getting from the Erlang/OTP compiler. Although it is indeed the case that currently there is plenty of room for improvement in Dialyzer's presentation of the identified discrepancies, it is clear that for many of the discrepancies simple one-line explanations of the form `"line 42: unused variable X"` will never be possible. Some of the discrepancies identified are complex, involve interactions of functions from various modules, and it is not always clear how to assign blame. As a simple example, note discrepancy 4 of Program 1: Dialyzer will currently complain that:

```
The clause matching involving 'EXIT' will never match;
argument is of type 'ok'
```

while the culprit is probably a missing `catch` construct after the `case`. As a more involved example, for discrepancy 2, Dialyzer will report that the guard will always fail. But perhaps function `m2:bar` should return negative integers in this tuple position after all.[9] Finding why inter-modular type inference determines that `m2:bar` only returns the numbers `0` and `42` in that tuple position might not be at all trivial — especially to users who are not familiar with type systems.

4. Experiences and lessons learned

Requests for better explanations of identified discrepancies aside, Dialyzer has been extremely successful. It has managed to identify a significant number of software defects that have remained undetected after a long period of extensive testing. For example, because of the high level of reliability required from telecom switches, the developers of AXD301, a team consisting of about 200 people, has over a period of more than eight years spent a considerable percentage of their effort on testing. Still Dialyzer managed to identify many discrepancies and often serious bugs. As another example, certain bugs in Erlang/OTP standard libraries managed to survive over many releases of the system, despite being in commonly used modules of the system. Although this may sound a bit contradictory, it has a simple explanation. Many of the bugs were in error-handling code or code paths of the library modules which were not executed frequently enough.

[4] GPRS: General Packet Radio Service.

[5] We thought that when academic projects got *real* users, it was supposed to be the other way around: the users would demand a GUI! More seriously, this is somewhat contrary to experience reported in literature. For example [2] reports that a significant portion of the effort is spent in explaining defects in a user-friendly way (presumably aided by a GUI). This probably does tell something about the habits of the Erlang programmer community, which is mostly Unix-centered, but we will not try to analyze it further.

[6] See `www.wings3d.com/`.

[7] Yaws: Yet Another Web Server; see `yaws.hyber.org/`.

[8] Erlang OpenGL/SDL API and utilities; see `sourceforge.net/projects/esdl`.

[9] Worse yet, there might even be a comment in its code to the effect that `m2:bar` possibly returns a negative integer. Some programmers currently trust comments more than output from static analyzers... but we are working on slowly changing this!

```
remote_dirty_select(Tab, [{HeadPat,_,_}] = Spec, [H|T]) when tuple(HeadPat), size(HeadPat) > 2, H =< size(Spec) ->
    ... % code for the body of this clause
remote_dirty_select(...) ->
    :
    :  % code for this and other clauses below handling select queries with general specifications
```

Figure 1. Code from the `mnesia` database with a guard that will always fail making the body of the first clause unreachable.

Observations Some qualitative observations can be made:

- The vast majority of (at least non-trivial) defects identified by Dialyzer are due to the interaction between multiple functions; a significant number of them span across module boundaries. This is mainly due to the fact that Erlang is great for testing functions on an individual basis (or in small sets), but currently provides little support for specifying and ensuring proper use of functions in other modules.
- Probably due to the reason described above, we did not observe the usual inverse correlation between the age of some piece of code and the number of discrepancies in it, at least not directly. The problem is that callers of some function may be significantly older than the callee and as the callee's interface evolves, the callers possibly remain unchanged. Having redundant clauses handling return values that were perhaps returned long ago but not anymore, is an extremely common discrepancy. As mentioned, such code is typically harmless but often desperately in need for cleanups. Doing so, simplifies the code where the discrepancy occurs and exposes opportunities for further simplifications elsewhere, often significant ones.
- Even in dynamically typed languages such as Erlang, the code that is frequently executed does not have type errors. As a result, Dialyzer tends to find most discrepancies off the commonly executed paths. Commonly executed paths are often reasonably well-tested and most discrepancies have already been eliminated. On the other hand, exception- or error-handling code, code that handles timeouts in concurrent programs, etc. does not always have this property.
- Quite often, fixing even simple discrepancies in some particular piece of code exposes more serious ones in code which lies in close proximity to the code which is fixed.

Most of these observations are not very surprising and in line with those of other researchers in the area.

Myths On the other hand, our experience so far has made us seriously doubt the validity of the following common beliefs:

1. *Software defects identified by a static analysis tool are shallow.* Occasionally one might see such a statement, especially in comparisons between static analysis and model checking techniques; see e.g. [3]. It is of course very hard to dispute such a statement, as its validity depends on what one considers as a "shallow" defect, but we will try to do so anyway.

   Figure 1 shows a small code segment from the code of Mnesia [6], a database management system distributed as part of Erlang/OTP. It also shows an actual discrepancy identified by Dialyzer, which is now fixed. On the surface, the indicated discrepancy is indeed shallow; a simple misuse of a library predicate (using `size` on a list rather than `length`). Viewed from this prism, there is indeed nothing "deep" here: the programmer made a silly mistake. Because the function `remote_dirty_select` is quite commonly used, what is surprising in this case is that this mistake managed to remain unnoticed over many Erlang/OTP releases. The subtlety of the problem was actually in the other clauses for the function. This bug was not identified because this clause was there for optimization purposes (in order to handle the common case of a single-element specification list fast). The subsequent clauses also provided the functionality of the first clause, but using a more general (handling specification lists of any size) and thus more expensive mechanism. It is very difficult to identify such performance-related software defects by means of (even exhaustive) testing. It is not clear to us that such software defects, for which the correctness criterion cannot be specified using a simple formula whose validity can be checked by model-checking techniques, are of the "shallow" kind.
2. *Software defects, once identified, are soon fixed.* Coming from academia, one is a bit shocked to discover that the "real world" is somehow different. Fixing bugs, no matter how serious, is not always a developer's top-priority, because program development in the real world follows a different model than that of open-source projects managed by small teams of individuals. Software evolution in big commercial projects goes hand-in-hand with filling bug reports, sending them to the developer who is responsible for the maintainance of the piece of code containing the bug, caring about backwards compatibility even when the functionality is crippled, and often having to invest a non-trivial amount of effort in order to modify or extend existing regression test suites, which are typically not maintained by programmers but by a separate testing team. (Our experience here is actually in line with that of other researchers; see e.g. [2, 3].)

   In fact, we even seriously doubt that an automatic classification of the seriousness of defects would help here. As a concrete example, we reported 18 discrepancies that Dialyzer identified in some library of Erlang/OTP to the library's maintainer, which incidentally was not the original author. Most of them were fixed pretty soon, but one in particular — which was the most serious — was not. In fact, it remained unfixed for quite some time. The reason for this was that it would involve serious redesign of the code and this might significantly change the behaviour of the library.
3. *Only programs written in low-level languages such as C seriously benefit from defect detection tools.* Not many serious developers believe this statement anyway, but often one of the arguments used in favour of high-level languages is that these languages avoid common programmer errors. Although this is a very true statement in some contexts (for example, one does not have the possibility to free memory once, let alone twice, in a garbage-collected language), it often fails to point out the fact that any language, no matter how high a level of abstraction it offers, has silly pitfalls and traps for developers, and these are often directly connected, and difficult to separate from, the language's strengths. We hold that software defect detection tools, especially lightweight ones, have their place in sofware development independently of the programming environment and language which is employed.

Final remarks Dialyzer is a static analysis tool identifying discrepancies — out of which some of them are serious bugs — in Erlang applications. We believe that the following, perhaps unique characteristics, have played a crucial role in Dialyzer's acceptance by its user community:

- The tool is extremely lightweight and requires absolutely no code changes or user-supplied annotations in order to be useful.
- The tool has so far tried its best to keep down the level of 'noise' which it generates, often at the expense of failing to report actual bugs. For the first versions of Dialyzer, one desired feature was to never issue a warning that could be perceived as misleading or be such that the user would find it extremely difficult to interpret. For example, we noticed that when analyzing virtual machine bytecode which has been generated using aggressive inlining, it would probably be difficult for naïve users to interpret the discrepancies. The approach we took was to simply suppress all discrepancies found in inline-compiled bytecode.

Although some might no doubt find this approach a bit extreme, we felt it was important for Dialyzer to succeed in gaining the developers' trust and be integrated in a non-disruptive way in the development process (i.e., without requiring any methodological changes from the users). Of course, this is only step number one. Once the developers' attitude and expectation level has been raised sufficiently, we intend to provide options that lift some of these restrictions.

Acknowledgments

Dialyzer's implementation would not have been possible without Tobias Lindahl. Thanks also go to all Dialyzer's users for useful contributions, feedback, thought-provoking requests for extensions of its functionality, and often working code for these changes.

The research of the author has been supported in part by grant #621-2003-3442 from the Swedish Research Council and by the Vinnova ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson AB and T-Mobile U.K.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience*, 30(7):775–802, June 2000.

[3] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation. Proceedings of the 5th International Conference*, volume 2937 of *LNCS*, pages 191–210. Springer, Jan. 2004.

[4] S. C. Johnson. Lint, a C program checker. Technical report, Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ, Dec. 1977.

[5] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[6] H. Mattsson, H. Nilsson, and C. Wikström. Mnesia - a distributed robust DBMS for telecommunications applications. In G. Gupta, editor, *Practical Applications of Declarative Languages: Proceedings of the PADL'1999 Symposium*, volume 1551 of *LNCS*, pages 152–163, Berlin, Germany, Jan. 1999. Springer.

[7] U. Wiger, G. Ask, and K. Boortz. World-class product certification using Erlang. *SIGPLAN Notices*, 37(12):25–34, Dec. 2002.

[8] Y. Xie and D. Engler. Using redundancies to find errors. *IEEE Trans. Software Eng.*, 29(10):915–928, Oct. 2003.