

Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story

Tobias Lindahl and Konstantinos Sagonas

Computing Science, Dept. of Information Technology, Uppsala University, Sweden
{Tobias.Lindahl,Konstantinos.Sagonas}@it.uu.se

Abstract. In safety-critical and high-reliability systems, software development and maintenance are costly endeavors. The cost can be reduced if software errors can be identified through automatic tools such as program analyzers and compile-time software checkers. To this effect, this paper describes the architecture and implementation of a software tool that uses lightweight static analysis to detect discrepancies (i.e., software defects such as exception-raising code or hidden failures) in large commercial telecom applications written in Erlang. Our tool, starting from virtual machine bytecode, discovers, tracks, and propagates type information which is often implicit in Erlang programs, and reports warnings when a variety of type errors and other software discrepancies are identified. Since the analysis currently starts from bytecode, it is completely automatic and does not rely on any user annotations. Moreover, it is effective in identifying software defects even in cases where source code is not available, and more specifically in legacy software which is often employed in high-reliability systems in operation, such as telecom switches. We have applied our tool to a handful of real-world applications, each consisting of several hundred thousand lines of code, and describe our experiences and the effectiveness of our techniques.

Keywords: Compile-time program checking, software development, software tools, defect detection, software quality assurance.

1 Introduction

All is fair in love and war, even trying to add a static type system in a dynamically typed programming language. Software development usually starts with love and passion for the process and its outcome, then passes through a long period of caring for (money making) software applications by simply trying to maintain them, but in the end it often becomes a war, the *war against software bugs*, that brings sorrow and pain to developers. In this war, the software defects will use all means available to them to remain in their favorite program. Fortunately, their primary weapon is concealment, and once identified, they are often relatively easy to kill.

In the context of statically typed programming languages, the type system aids the developer in the war against software bugs by automatically identifying type errors at compile time. Unfortunately, the price to pay for this victory is the compiler rejecting all programs that cannot be proved type-correct by the currently employed type system.

This starts another war, the *war against the type system*, which admittedly is a milder one. The only way for programmers to fight back in this war is to rewrite their programs. (Although occasionally the programming language developers help the programmers in fighting this war by designing a bigger weapon, i.e., a more refined type system).

Dynamically typed programming languages avoid getting into this second war. Instead, they adopt a more or less “anything goes” attitude by accepting all programs, and relying on type tests during runtime to prevent defects from fighting back in a fatal way. Sometimes these languages employ a less effective weapon than a static type system, namely a *soft type system*, which provides a limited form of type checking. To be effective, soft type systems often need guidance by manual annotations in the code. Soft typing will not reject any program, but will instead just inform the user that the program could not be proved type-correct. In the context of the dynamically typed programming language ERLANG, attempts have been made to develop such soft type systems, but so far none of them has gained much acceptance in the community. We believe the main reasons for this is the developers’ reluctance to invest time (and money) in altering their already existing code and their habits (or personal preferences). We remark that this is not atypical: just think of other programming language communities like e.g., that of C.

Instead of devising a full-scale type checker that would need extensive code alterations in the form of type annotations to be effective, we pragmatically try to adapt our weapon’s design to the programming style currently adhered to by ERLANG programmers. We have developed a lightweight type-based static analysis for finding *discrepancies* (i.e., software defects such as exception-raising code, hidden failures, or redundancies such as unreachable code) in programs without having to alter their source in any way. The analysis does not even need access to the source, since its starting point is virtual machine bytecode. However, the tool has been developed to be extensible in an incremental way (i.e., with the ability to take source code into account and benefit from various kinds of user annotations), once it has gained acceptance in its current form.

The actual tool, called DIALYZER,¹ allows its user to find discrepancies in ERLANG applications, based on information both from single modules and from an application-global level. It has so far been applied to programs consisting of several thousand lines of code from real-world telecom applications, and has been surprisingly effective in locating discrepancies in heavily used, well-tested code.

After briefly introducing the context of our work in the next section, the main part of the paper consists of a section which explains the rationale and main methods employed in the analysis (Sect. 3), followed by Sect. 4 which describes the architecture, effectiveness, and current and future status of DIALYZER. Section 5 reviews related work and finally this paper finishes in Sect. 6 with some concluding remarks.

2 The Context of our Work

The Erlang language and Erlang/OTP. ERLANG [1] is a strict, dynamically typed functional programming language with support for concurrency, communication, dis-

¹ DIALYZER: Discrepancy ANALYZer of Erlang programs. (From the Greek *διαλύω*: to dissolve, to break up something into its component parts.) System is freely available from www.it.uu.se/research/group/hipe/dialyzer/.

tribution and fault-tolerance. The language relies on automatic memory management. ERLANG's primary design goal was to ease the programming of soft real-time control systems commonly developed by the telecommunications (telecom) industry.

ERLANG's basic data types are atoms, numbers (floats and arbitrary precision integers), and process identifiers; compound data types are lists and tuples. A notation for objects (*records* in the ERLANG lingo) is supported, but the underlying implementation of records is the same as tuples. To allow efficient implementation of telecommunication protocols, ERLANG nowadays also includes a *binary* data type (a vector of byte-sized data) and a notation to perform pattern matching on binaries. There are no destructive assignments of variables or mutable data structures. Functions are defined as ordered sets of guarded clauses, and clause selection is done by pattern matching. In ERLANG, clause guards either succeed or silently fail, even if these guards are calls to builtins which would otherwise raise an exception if used in a non-guard context. Although there is a good reason for this behavior, this is a language "feature" which often makes clauses unreachable in a way that goes unnoticed by the programmer. ERLANG also provides a *catch/throw*-style exception mechanism, which is often used to protect applications from possible runtime exceptions. Alternatively, concurrent programs can employ so called *supervisors* which are processes that monitor other processes and are responsible for taking some appropriate clean-up action after a software failure.

Erlang/OTP is the standard implementation of the language. It combines ERLANG with the Open Telecom Platform (OTP) middleware. The resulting product, Erlang/OTP, is a library with standard components for telecommunications applications (an ASN.1 compiler, the Mnesia distributed database, servers, state machines, process monitors, tools for load balancing, etc.), standard interfaces such as CORBA and XML, and a variety of communication protocols (e.g., HTTP, FTP, SMTP, etc.).²

Erlang applications and real-world uses. The number of areas where ERLANG is actively used is increasing. However, its primary application area is still in large-scale embedded control systems developed by the telecom industry. The Erlang/OTP system has so far been used quite successfully both by Ericsson and by other companies around the world (e.g., T-Mobile, Nortel Networks, etc.) to develop software for large (several hundred thousand lines of code) commercial applications. These telecom products range from high-availability ATM servers, ADSL delivery systems, next-generation call centers, Internet servers, and other such networking equipment. Their software has often been developed by large programming teams and is nowadays deployed in systems which are currently in operation. Since these systems are expected to be robust and of high availability, a significant part of the development effort has been spent in their (automated) testing. On the other hand, more often than not, teams which are currently responsible for a particular product do not consist of the original program developers. This and the fact that the code size is large often make bug-hunting and software maintenance quite costly endeavors. Tools that aid this process are of course welcome.

Our involvement in Erlang and history of this work. We are members of the HiPE (High Performance Erlang) group and over the last years have been developing the

² Additional information about ERLANG and Erlang/OTP can be found at www.erlang.org.

HiPE native code compiler [10, 16]. The compiler is fully integrated in the open source Erlang/OTP system, and translates, in either a just-in-time (JIT) or ahead-of-time fashion, BEAM virtual machine bytecode to native machine code (currently UltraSPARC, x86, and AMD64). The system also extends the Erlang/OTP runtime system to support mixing interpreted and native code execution, at the granularity of individual functions.

One of the means for generating fast native code for a dynamically typed language is to statically eliminate as much as possible the (often unnecessary) overhead that type tests impose on runtime execution. During the last year or so, we have been experimenting with type inference and an aggressive type propagator, mainly for compiler optimization purposes. In our engagement on this task, we noticed that every now and then the compiler choked on pieces of ERLANG code that were obviously bogus (but for which the rather naïve bytecode compiler happily generated code). Since in the context of a JIT it does not really make much sense to stop compilation and complain to the user, and since it is a requirement of HiPE to preserve the observable behavior of the bytecode compiler, we decided to create a separate tool, the DIALYZER, that would statically analyze ERLANG (byte)code and report defects to its users. We report on the methods we use and the implementation of the tool below. However, we stress that the DIALYZER is not just a type checker or an aggressive type propagator.

3 Detecting Discrepancies through Lightweight Static Analysis

3.1 Desiderata

Before we describe the techniques used in DIALYZER, we enumerate the goals and requirements we set for its implementation before we embarked on it:

1. The methods used in DIALYZER should be *sound*: they should aim to maximize the number of reported discrepancies, but should not generate any false positives.
2. The tool should request minimal, preferably no, effort or guidance from its user. In particular, the user should not be *required* to do changes to existing code like providing type information, specifying pre- or post-conditions in functions, or having to write other such annotations. Instead the tool should be completely automated and able to analyze legacy ERLANG code that (quite often) no current developer is familiar with or willing to become so. On the other hand, if the user *chooses* to provide more information, the tool should be able to take it into consideration and improve the precision of the results of its analysis.
3. The tool should be able to do something reasonable even in cases where source code is not available, as e.g., could be the case in telecom switches under operation.
4. The analysis should be *fast* so that DIALYZER has a chance to become an integrated component of ERLANG development.

All these requirements were pragmatically motivated. The applications we had in mind as possible initial users of our tool are large-scale software systems which typically have been developed over a long period and have been tested extensively. This often creates the illusion that they are (almost) bug-free. If the tool reported to their maintainers 1,000 possible discrepancies the first time they use it, of which most are false alarms, quite

possibly it would not be taken seriously and its use would be considered a waste of time and effort.³ In short, what we were after for DIALYZER version 1.0 was to create a lightweight static analysis tool capable of locating discrepancies that are errors: i.e., software defects that are easy to inspect and are easily fixed by an appropriate correcting action.⁴ We could relax these requirements only once the tool gained the developers' approval; more on this in Sect. 4.4.

Note that the 2nd requirement is quite strong. It should really be obvious, but it also implies that there are no changes to the underlying philosophy of the language: ERLANG is dynamically typed and there is nothing in our method that changes that.⁵

3.2 Local Analysis

To satisfy the requirement that the analysis is fast, the core of the method is an *intra-procedural, forward* dataflow analysis to determine the set of possible values of live variables at each program point using a *disjoint union of prime types*. The underlying type system itself is based on an extension of the Hindley-Milner static type discipline that incorporates recursive types and accommodates a limited form of union types without compromising its practical efficiency. In this respect, our type system is similar to that proposed by Wright and Cartwright for Soft Scheme [18].

The internal language of the analysis to which bytecode is translated, called Icode, is an idealized ERLANG assembly language with unlimited number of temporaries and an implicit stack. To allow for efficient dataflow analyses and to speed up the fixpoint computation which is required when loops are present, Icode is represented as a control-flow graph (CFG) which has been converted into static single assignment (SSA) form [3]. In Icode, most computations are expressed as function calls and all temporaries survive these. The function calls are divided into calls to primitive operations (primops), built-in functions (bifs), and user-defined functions. Furthermore, there are assignments and control flow operations, including switches, type tests, and comparisons. The remainder of this section describes the local analysis; in Sect. 3.3 we extend it by describing the handling of user-defined functions and by making it inter-modular.

Although ERLANG is a dynamically typed language, type information is present both explicitly and implicitly at the level of Icode. The explicit such information is in the form of type tests which can be translations of explicit type guards in the ERLANG source code, or tests which have been introduced by the compiler to guard unsafe primitive operations. The implicit type information is hidden in calls to primops such as in e.g. addition, which demands that both its operands are numbers. Note that non-trivial

³ This was not just a hunch; we had observed this attitude in the past. Apparently, we are not the only ones with such experiences and this attitude is not ERLANG-specific; see e.g. [5, Sect. 6].

⁴ Despite the conservatism of the approach, we occasionally had hard time convincing developers that some of the discrepancies identified by the tool were indeed code that needed some correcting action. One reaction we got was essentially of the form: "*My program cannot have bugs. It has been used like that for years!*". Fortunately, the vast majority of our users were more open-minded.

⁵ This sentence should *not* be interpreted as a religious statement showing our conviction on issues of programming language design; instead it simply re-enforces that we chose to follow a very pragmatic, down-to-earth approach.

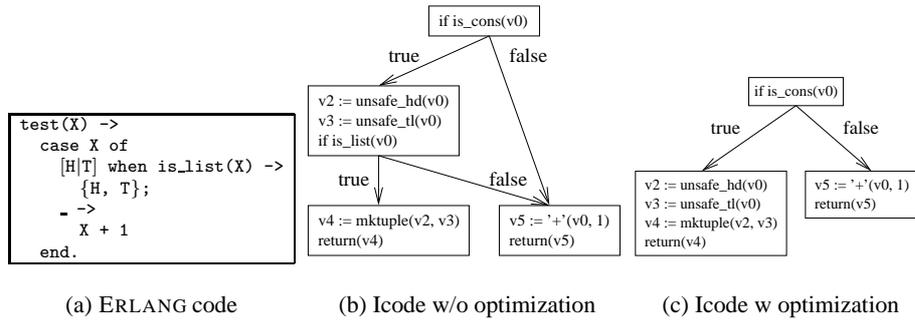


Fig. 1. ERLANG code with a redundant type guard.

types for arguments and return values for all primops and bifs can be known *a priori* by the analyzer. These types can be propagated forward in the CFG to jump-start the discrepancy analysis. For example, if a call to addition succeeds, we know for sure that the return value must be a number. We also know that, from that point forward in the CFG the arguments must be numbers as well, or else the operation would have failed. Similarly, if an addition is reached and one of its arguments has a type which the analysis has already determined is not a number, then this is a program point where a discrepancy occurs.

More specifically, the places where the analysis changes its knowledge about the types of variables are:

1. At the *definition point* of each variable.⁶ At such a point, the assigned type depends on the operation on the right-hand side. If the return type of the operation is unknown, or if the operation statically can be determined to fail, the variable gets assigned the type *any* (the lattice's top) or *undefined* (its bottom), respectively.
2. At *splits* in the CFG, such as in nodes containing type tests and comparisons. The type propagated in the success branch is the *infimum* (the greatest lower bound in the lattice) of the incoming type and the type tested for. In the fail branch, the success type is subtracted from the incoming set of types.
3. At a point where a variable is used as an *argument* in a call to a primop or a bif with a known signature. The propagated type is the *infimum* of the incoming type and the demanded argument type for the call. If the call is used in a guard context, then this is a split in the CFG and the handling will be as in case 2 above.

When paths join in the CFG, the type information from all incoming edges is unioned, making the analysis *path-insensitive*. Moreover, when a path out of a basic block cannot be taken, the dead path is removed to simplify the control flow. In Fig. 1 the `is_list/1` guard in the first clause of the `case` statement can be removed since the pattern matching compiler has already determined that `X` is bound to a (possibly non-proper) list. This removal identifies a possible discrepancy in the code.

⁶ Note that since Icode is on SSA form there can be only one definition point for each variable.

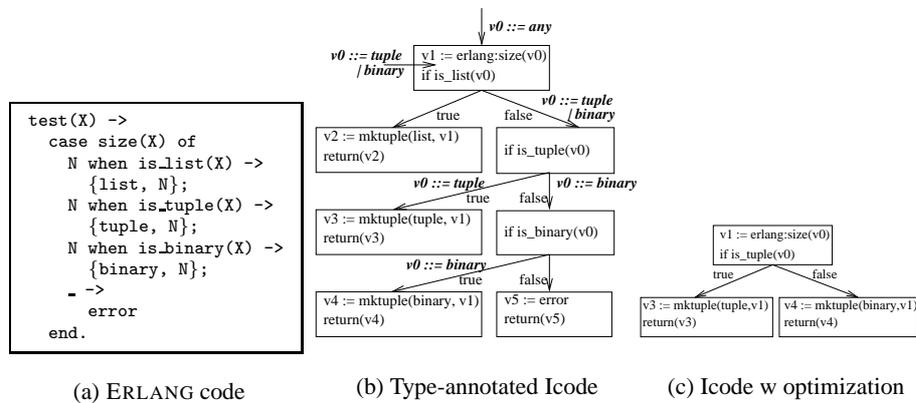


Fig. 2. An ERLANG program with two discrepancies due to a misuse of the bif `size/1`.

The analysis, through such local type propagation aided by liveness analysis and by applying aggressive global sparse conditional constant propagation and dead code elimination [13], tries to reason about the intent of the programmer. If the most likely path out of a node with a type test is removed, or if a guard always fails, this is reported to the user as a discrepancy. Other discrepancies that are identified by local static analysis include function calls that always fail, pattern matching operations that will raise a runtime exception, and dead clauses in switches (perhaps due to an earlier more general clause). For example, on the program of Fig. 2(a), given that the signature of the `erlang:size/1` built-in function is

```
size(tuple | binary) -> integer
```

the analysis first annotates the Icode control-flow graph with type information. This can be seen in Fig. 2(b) which shows the result of propagating types for variable `v0` only. Given such a type-annotated CFG, it is quite easy to discover and report to the user that both the first case clause and the catch-all clause are dead, thereby removing these clauses; see Fig. 2(c). In our example, finding code which is redundant, especially the first clause, reveals a subtle programming error as the corresponding ‘measuring’ function for lists in ERLANG is `length/1`, not `size/1`.

3.3 Making the Analysis Intra- and Inter-Modular

Currently, the only way to provide the compiler with information about the arguments to a function is by using non-variable terms and guards in clause heads. (This information is primarily used by pattern matching to choose between the function clauses.) Since DIALYZER employs a forward analysis, when analyzing only one function, there can be no information at the function’s entry point, but at the end of the analysis there is information about the type of the function’s return value. By unioning all proper (i.e., non-exception) type values at the exit points of a function, we get additional type

information that can then be used at the function's call sites. The information is often non-trivial since most functions are designed to return values of a certain type and do not explicitly fail (i.e., raise an exception). To take advantage of this, the local analysis is extended with a *persistent lookup table*, mapping function names to information about their return values. The table is used both for intra-modular calls and for calls across module boundaries, but since the table only contains information about functions which have already been analyzed, some kind of iterative analysis is needed.

First consider intra-modular calls. One approach is to iteratively analyze all functions in a module until the information about their return values remains unchanged (i.e., until a fixpoint is reached). The possible problem with this approach is that the fixpoint computation can be quite expensive. Another approach, which is currently the default, is to construct a static call graph of the functions in a module, and then perform one iteration of the analysis by considering the strongly connected components of this graph in a bottom-up fashion (i.e., based on a reversed topological sort). If all components consist of only one function, this will find the same information as an iterative analysis. If there are components which consist of mutually recursive functions, we can either employ fixpoint computation or heuristically compute a safe approximation of the return value types in one pass (for example, the type *any*). Note that this heuristic is acceptable in our context; the discrepancy analysis remains sound but is not complete (i.e., it is not guaranteed to find all discrepancies).

Now consider function calls across module boundaries. In principle, the call graph describing the dependencies between modules can be constructed *a priori*, but this imposes an I/O-bound start-up overhead which we would rather avoid. Instead, we construct this graph as the modules are analyzed for the first time, and use this information only if the user requests a complete analysis which requires a fixpoint computation.

A final note: So far, the analysis has been applied to code of projects which are quite mature. However, as mentioned, our intention is that the tool becomes an integrated component of the program development cycle. In such situations, the code of a module changes often, so the information in the lookup table may become obsolete. When a project is in a phase of rapid prototyping, it might be convenient to get reports of discrepancies discovered based on the code of a single module. The solution to this is to analyze one module till fixpoint, using a lookup table that contains function information from only the start of the analysis of that module. The tool supports such a mode.

4 DIALYZER

4.1 Architecture and Implementation

Figure 3 shows the DIALYZER in action, analyzing the application **inets** from the standard library of the Erlang/OTP R9C-0 distribution.

In DIALYZER v 1.0, the user can choose between different modes of operation. The *granularity* option controls whether the analysis is performed on a single module or on all modules of an application. The *iteration* option selects between performing the analysis till fixpoint or doing a quick-and-dirty, one-pass analysis. The meaning of this

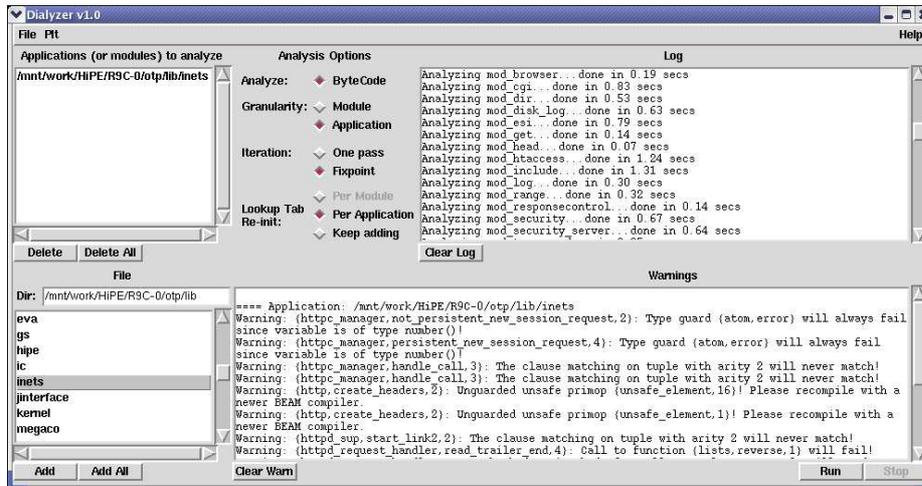


Fig. 3. The DIALYZER in action.

option partly depends on the selected granularity. For example, if the granularity is per application and the one-pass analysis is selected, each module is only analyzed once, but fixpoint iteration is still applied inside the module. Finally, the *lookup table re-init* option specifies when the persistent lookup table is to be re-initialized, i.e., if the information is allowed to leak between the analysis elements specified by the granularity. Combinations of options whose semantics is unclear are automatically disabled.

While the analysis is running, a log displays its progress, and the discrepancies which are found are reported by descriptive warnings in a separate window area; see Fig. 3. When the analysis is finished, the log and the warnings can be saved to files. As described in Sect. 3.3, the module-dependency graph is calculated during the first iteration of the analysis of an entire application. If a fixpoint analysis on the application level is requested, DIALYZER uses this information to determine the order in which the modules are analyzed in the next iteration to reduce the number of iterations needed to reach a fixpoint. In fact, even in the one-pass mode the module-dependency graph is constructed, just in case the user decides to request a fixpoint analysis on completion. Requesting this is typically not burdensome as the analysis is quite fast; this can also be seen in the figure. On a 2GHz laptop running Linux, the DIALYZER analyzes roughly 800 lines of ERLANG code per second, including I/O. (For example, the sizes of **mod_cgi**, **mod_disk_log**, and **mod_htaccess** modules are 792, 405, and 1137 lines, respectively. As another example, one run of the analysis for the complete Erlang/OTP standard library, comprising of about 600,000 lines of code, takes around 13 minutes.)

The DIALYZER distribution includes the code for the graphical user interface and the analyzer, both written in ERLANG. As its analyzer depends on having access to a specific version of the HiPE native code compiler, on whose infrastructure (the BEAM bytecode disassembler, the translator from BEAM to Icode, and the Icode supporting code such as SSA conversion and liveness analysis) it relies, the presence of a recent

Erlang/OTP release is also required. Upon first start-up, DIALYZER will automatically trigger the fixpoint-based analysis of the Erlang/OTP standard library, `stdlib`, to construct a persistent lookup table which can be used as a basis for all subsequent analyses.

4.2 The DIALYZER in Anger

In order to show that our analysis is indeed effective in identifying software defects, we present some results obtained from using the DIALYZER to analyze code from large-scale telecom applications written in ERLANG. These applications all have in common that they are heavily used and well-tested commercial products, but as we will see, DIALYZER still exposed problems that had gone unnoticed by testing. Some brief additional information about these applications appears below:

- AXD301 is an asynchronous transfer mode (ATM) switching system from Ericsson [2]. The project has been running for more than eight years now and its team currently involves 200 people (but this number also includes some support staff; not only developers or testers). The ATM switch is designed for non-stop operation, so robustness and high availability are very important and taken seriously during development. As a consequence, a significant effort (and part of the project’s budget) has been spent on testing its safety-critical components; see also [17].
- GPRS (General Packet Radio Service) is a telecom system from Ericsson. A large percentage of its code base is written in ERLANG. The project has been running for more than seven years now and its testing includes extensive test suites, automated daily builds, and code coverage analysis. Since this was a pilot-study for the applicability and effectiveness of DIALYZER in identifying discrepancies, only part of GPRS’s ERLANG code has so far been analyzed. Although only part of the total code base, the analyzed code is rather big: it consists of 580,000 lines of ERLANG code, excluding comments.
- Melody is a control system for a “Caller Tunes” ringbacktone service developed by T-Mobile. It is an implementation of a customer database with interfaces to media players, short message service centers, payment platforms, and provisioning systems. The core of Melody is significantly smaller than the other telecom products which were analyzed; however, it includes parts of T-Mobile’s extensively used and well-tested standard telecom library.

In addition to these commercial applications of ERLANG, we also analyzed the complete set of standard libraries from Erlang/OTP release R9C-0 from Ericsson and code from Jungerl,⁷ which is an open-source code repository for ERLANG developers.

In order to have a more refined view of the kinds of discrepancies DIALYZER found, we can manually divide them into the following categories:

Explosives These are places in the code that would raise a run-time exception. Examples of this are calls to ERLANG built-in functions with the wrong type of arguments, operators not defined on certain operands, faulty (byte) code, etc. An explosive can of course be conditional (e.g., firing on some execution paths only, rather than in all paths).

⁷ A Jungle of ERLANG code; see sourceforge.net/projects/jungerl/.

Camouflages These are programming errors that for example make clauses or branches in the control flow graph unreachable — although the programmer did not intend them as such — without causing the program to stop or a supervisor process being notified that something is wrong. The most common error of this kind is a guard that will always silently fail.

Cemeteries These are places of dead code. Such code is of course harmless, but code that can never be executed often reveals subtle programming errors. A common kind of cemeteries are clauses in case statements which can never match (because of previous code) and are thus redundant.

For example, in the code of Fig. 2(a) if the analysis encounters, in this or in some other module, a call of the form `test([_|_])`, this is classified as an explosive since it will generate a runtime exception. In the same figure, both the first and the last clause of the case statement are cemeteries, as they contain dead code. On the other hand, the code fragment below shows an example of a camouflage: the silent failure of the `size(X)` call in a guard context will prevent this clause from ever returning, although arguably the programmer’s intention was to handle big lists.

```
test(X) when is_list(X), size(X) > 10 ->
    {list, big_size};
... %% Other clauses
```

Table 1 shows the number of discrepancies found in the different projects.⁸ The numbers in the column titled “lines of code” show an indication of the size of each project (comments and blank lines have been excluded) and justify our reasoning why requiring type information or any other user annotations *a posteriori* in the development cycle is not an option in our context. Although we would actually strongly prefer to have any sort of information that would make the analysis more effective, we are fully convinced that it would be an enormous task for developers to go through all this code and provide type information — especially since this would entail intimate knowledge about code that might have been written by someone else years ago. Realistically, the probability of this happening simply in order to start using DIALYZER in some commercial project, is most certainly zero.

Despite these constraints, DIALYZER is quite effective in identifying software defects in the analyzed projects; see Table 1. Indeed, we were positively surprised by the amount of discrepancies DIALYZER managed to identify, given the amount of testing effort already spent on the safety-critical components of these projects and the conservatism of the methods which DIALYZER version 1.0 currently employs.

In addition to finding programming errors in ERLANG code, DIALYZER can also expose software errors which were caused by a rather flawed translation of record expressions by the BEAM bytecode compiler. In Table 1, 31 of the reported explosives for Erlang/OTP R9C-0 and 7 for Melody (indicated in parentheses) are caused by the BEAM compiler generating unsafe instructions that fail to be guarded by an appropriate type test. This in turn could result in buffer overruns or segmentation faults if the

⁸ Actually, DIALYZER also warns its user about the use of some archaic ERLANG idioms and code relics; these warnings are not considered discrepancies and are not reported in Table 1.

Table 1. Number of discrepancies of different kinds found in the analyzed projects.

Project	Lines of code (total)	Discrepancies (total)	Classification		
			Explosives	Camouflages	Cemeteries
OTP R9C-0	600,000	57	38 (31)	5	14
AXD301	1,100,000	132	26	2	104
GPRS	580,000	44	10	2	32
Jungerl	80,000	12	5	2	5
Melody	25,000	9	8 (7)	1	0

instructions' arguments were not of the (implicitly) expected type. This compiler bug has been corrected in release R9C-1 of Erlang/OTP.

4.3 Current Features and Limitations

The tool confuses programming errors with errors in the BEAM bytecode. Typically this is not a problem as DIALYZER has built-in knowledge about common discrepancies caused by flawed BEAM code. When such a discrepancy is encountered, DIALYZER recommends its user to re-generate the bytecode file using a newer BEAM compiler and re-run the analysis. As a matter of fact, we see this ability to identify faulty BEAM code as an advantage rather than as a limitation.

Starting from bytecode unfortunately means that warning messages cannot be descriptive enough: in particular they do not precisely identify the clause/line where the discrepancy occurs; see also Fig. 3. This can often be confusing. Also, since soundness currently is a major concern, the DIALYZER only reports warnings when it is clear that these are discrepancies. For example, if a switch contains a clause with a pattern that cannot possibly match then this is reported since it is a clear discrepancy. On the other hand, if the analysis finds that the patterns in the cases of the switch fail to cover all possible type values of the incoming term, this is not reported since it might be due to over-approximation caused by the path-insensitivity of the analysis. Of course, we could easily relax this and let the programmer decide, but as explained in Sect. 3.1 soundness is a requirement which DIALYZER religiously follows at this point.

4.4 Planned Future Extensions

One of the strengths of DIALYZER version 1.0 is that no alterations to the source code are needed. In fact, as we have pointed out, the tool does not even need access to it. However, if the source code is indeed available, it can provide the analysis with additional information. Work is in progress to generate Icode directly from CORE ERLANG, which is the official core language for ERLANG and the language used internally in the BEAM compiler. Since CORE ERLANG is on a level which is closer to the original source, where it is easier to reason about the programmer's intentions, it can provide DIALYZER with means to produce better warning messages; in particular line number information can be retained at this level. The structure of CORE ERLANG can also help in deriving, in a more precise way, information about the possible values used as arguments to functions that are local to a module.

We also plan to extend DIALYZER with the possibility that its user incrementally adds optional type annotations to the source code. The way to do this is not yet decided, but the primary goal of these annotations, besides adding valuable source code documentation, is to aid the analysis in its hunt for discrepancies, not to make ERLANG a statically typed language. If a type signature is provided for a function, and this signature can be verified by DIALYZER as described below, it can be used by the analysis in the same way as calls to bifs and primops are used in the current version. The way to verify a signature is as follows: instead of trying to infer the types at each call site (as would be the case in most type systems), the signature would be trusted until the function is analyzed. At this point the signature would be compared to the result of the analysis and checked for possible violations. Since DIALYZER is not a compiler, no programs would be rejected, but if violations of user-defined signatures are discovered, this would be reported to the user together with a message saying that the results of the discrepancy analysis could not be trusted.

Taking this idea further, we also plan to experiment with relaxing soundness by allowing the user to specify annotations that in general cannot be statically verified (for example, that a certain argument is a non-negative integer). This is similar to the direction that research for identifying defects such as buffer overruns and memory leaks in C (see e.g. [6, 4]) or for detecting violations of specifications in Java programs [8] has recently taken.

5 Related Work

Clearly, we are not the first to notice that compiler and static analysis technology can be employed for identifying defects in large software projects.⁹ Especially during the few last years, researchers in the programming language community have shown significant interest in this subject; see e.g. the work mentioned in the last paragraph of the previous section and the references therein. Most of that work has focused on detecting errors such as buffer overruns, memory access errors such as accessing memory which has already been freed or following invalid pointer references in C, race detection in multi-threaded Java programs, etc. These software defects are simply not present in our context, at least not directly so.¹⁰ Similarly to what we do, some of these analyses do not need source code to be present, since they start from the binary code of the executable. On the other hand, we are not aware of any work that tries to detect flaws at the level of virtual machine bytecode caused by its flawed generation.

During the late 80's and the beginning of the 90's, the subject of automatic type inference without type declarations received a lot of attention; see e.g. [12] for an early work on the subject. A number of soft type systems have been developed, most of them for the functional languages Lisp and Scheme, and some for Prolog. The one closest to our work is that of Soft Scheme [18]. Perhaps sadly, only a few of them made it into actual distributions of compilers or integrated development environments for these languages. Some notable exceptions are DrScheme [7], a programming environment for Scheme which uses a form of set-based analysis to perform type inference and to mark

⁹ We are also willing to bet our fortunes that we will not be the last ones to do so either!

¹⁰ They can only occur in the VM interpreter which is written in C, not in ERLANG code.

potential errors, and the NUDE (the NU-Prolog Debugging Environment [14]) and Ciao Prolog [9] systems which also incorporate type-annotation-guided static debuggers.

In the context of ERLANG, two type systems have been developed before: one based on subtyping [11] and a recent one based on soft types [15]. To the best of our knowledge, the latter has not yet been used by anybody other than its author, although time might of course change this. The former ([11]) allows for declaration-free recursive types using subtyping constraints, and algorithms for type inference and checking are also given in the same paper. It is fair to say that the approach has thus far not been very successful in the ERLANG community. Reasons for this include the fact that the type system constrains the language by rejecting code that does not explicitly handle cases for failures, that its inference algorithm fails to infer types of functions depending on certain pattern matching constructs, and that it demands a non-trivial amount of user intervention (in the form of type annotations in the source code). Stated differently, what [11] tries to do is to impose a style of programming in ERLANG which is closer to that followed in statically typed languages, in order to get the benefits of static type-error detection. Clearly this goal is ambitious and perhaps worthwhile to pursue, but then again its impact on projects which already consist of over a million lines of code is uncertain. Our work on the other hand is less ambitious and more pragmatically oriented. We simply aim to locate (some of the) software defects in already developed ERLANG code, *without imposing a new method for writing programs, but by trying to encourage an implicit philosophy for software development* (namely, the frequent use of a static checker tool rather than just relying on testing) which arguably is better than the practice the (vast majority of the) ERLANG community currently follows.

6 Concluding Remarks

DIALYZER version 1.0 represents a first attempt to create a tool that uses lightweight static analysis to detect software defects in large telecom applications and other programs developed using ERLANG. While we believe that our experiment has been largely successful, there are several aspects of the tool that could be improved through either better technology or by relaxing its requirements (e.g., no false warnings), which are currently quite stringent. Given support, we intend to work in these directions.

On a more philosophical level, it is admittedly the case that most of the software defects identified by DIALYZER are not very deep. Moreover, this seems to be an inherent limitation of the method. For example, problems such as deadlock freedom of ERLANG programs cannot be checked by DIALYZER. One cannot help being a bit skeptical about the *real* power of static analysis or type systems in general, and wonder whether a tool that used techniques from software model checking would, at least in principle, be able to check for a richer set of properties and give stronger correctness guarantees. On the other hand, there is enough evidence that neither static analysis nor software model checking are currently at the stage where one dominates the other; see also [5].

More importantly, one should *not* underestimate the power of simplicity and ease of use of a (software) tool. In a relatively short time and with very little effort, DIALYZER managed to identify a large number of software defects that had gone unnoticed after years of testing. Moreover, it managed to identify bugs that are relatively easy to correct

— in fact some of them have been already — which brings software in a state closer to the desired goal of total correctness. One fine day, some projects might actually win their war!

Acknowledgments

This research has been supported in part by VINNOVA through the ASTEC (Advanced Software Technology) competence center as part of a project in cooperation with Ericsson and T-Mobile. The research of the second author was partly supported by a grant by Vetenskapsrådet (the Swedish Research Council). We thank Ulf Wiger and Hans Nilsson from the AXD301 team at Ericsson, Kenneth Lundin from the Erlang/OTP team, and Sean Hinde from T-Mobile for their help in analyzing the code from commercial applications and for their kind permission to report those results in this paper.

References

1. J. Armstrong, R. Viriding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.
2. S. Blau and J. Rooth. AXD 301—A new generation ATM switching system. *Ericsson Review*, 75(1):10–17, 1998.
3. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.
4. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167. ACM Press, June 2003.
5. 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*, number 2937 in LNCS, pages 191–210. Springer, Jan. 2004.
6. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan./Feb. 2002.
7. R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, Mar. 2002.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, June 2002.
9. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program development using abstract interpretation (and the Ciao system preprocessor). In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, number 2694 in LNCS, pages 127–152, Berlin, Germany, June 2003. Springer.
10. E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43, New York, NY, Sept. 2000. ACM Press.
11. S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 136–149. ACM Press, June 1997.

12. P. Mishra and U. S. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 7–21. ACM Press, 1984.
13. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Francisco, CA, 1997.
14. L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog debugging environment. In A. Porto, editor, *Proceedings of the Sixth International Conference on Logic Programming*, pages 521–536. The MIT Press, June 1989.
15. S.-O. Nyström. A soft-typing system for Erlang. In *Proceedings of ACM SIGPLAN Erlang Workshop*, pages 56–71. ACM Press, Aug. 2003.
16. M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In Z. Hu and M. Rodríguez-Artalejo, editors, *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 228–244, Berlin, Germany, Sept. 2002. Springer.
17. U. Wiger, G. Ask, and K. Boortz. World-class product certification using Erlang. *SIGPLAN Notices*, 37(12):25–34, Dec. 2002.
18. A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst.*, 19(1):87–152, Jan. 1997.