Science of Computer Programming 147 (2017) 109–134

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Concolic testing for functional languages ☆

Aggelos Giantsios [a,*], Nikolaos Papaspyrou [a], Konstantinos Sagonas [a,b]

[a] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[b] Department of Information Technology, Uppsala University, Sweden

ARTICLE INFO

Article history:
Received 11 January 2016
Received in revised form 24 April 2017
Accepted 26 April 2017
Available online 12 May 2017

Keywords:
Concolic testing
Symbolic execution
Pattern matching
Erlang

ABSTRACT

Concolic testing is a software testing technique that simultaneously combines concrete execution of a program (given specific input, along specific paths) with symbolic execution (generating new test inputs that explore other paths, which gives better path coverage than random test case generation). So far, concolic testing has been applied, mainly at the level of bytecode or assembly code, to programs written in imperative languages that manipulate primitive data types such as integers and arrays. In this article, we demonstrate its application to a functional programming language core, the functional subset of Core Erlang, that supports pattern matching, structured recursive data types such as lists, recursion and higher-order functions. We present CutEr, a tool implementing this testing technique, and describe its architecture, the challenges that it needs to address, its current limitations, and report some experiences from its use.

1. Introduction

Testing is, and quite likely will continue to be, the most commonly used method to ensure the correctness and reliability of software. Compared to manually written tests, automated testing techniques have the potential to improve software reliability by discovering more situations (often "corner cases") that result in software errors, achieve better coverage, and reduce the costs of testing.

The declarative programming language community in general, and the community of functional languages in particular, has long ago realized the benefits of automated testing. However, research in this area has so far focused primarily on developing techniques for the random testing of program properties, also known as *property-based testing*. This form of testing is available e.g., in Haskell in the form of the QuickCheck [1] and SmallCheck [2] libraries, and in Erlang in the form of the QuviQ QuickCheck [3] and PropEr [4] tools. Despite its effectiveness, property-based testing is not effortless as it is only semi-automatic: it requires programmers to write and maintain properties as well as specify generators for checking these properties. The effort required to do so is often significant.

In imperative programming languages, such as C/C++ and Java, a fully automatic testing approach, called *concolic testing*, has gained popularity during the recent years. Starting with an arbitrary well-formed input, concolic testing consists of concretely executing the program unit under test, gathering symbolic constraints on inputs from conditional branches encountered along the concrete execution. The collected constraints are then systematically negated and solved with a con-

☆ This work has been partially supported by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software".

* Corresponding author.
E-mail addresses: aggelgian@softlab.ntua.gr (A. Giantsios), nickie@softlab.ntua.gr (N. Papaspyrou), kostis@cs.ntua.gr (K. Sagonas).

http://dx.doi.org/10.1016/j.scico.2017.04.008
0167-6423/

straint solver, whose solutions are mapped to new inputs that exercise different program execution paths. This process is repeated, using some appropriate search strategy, in an attempt to sweep through most (if not all) feasible execution paths of the program. Any assertion violations or crashes that occur during this process of concolic execution are reported as test failures and the corresponding inputs can also be collected in a set of (automatically generated) tests for the program unit. Using a lot of engineering, this approach has been fine-tuned and has resulted in very powerful and scalable testing tools: DART [5] and CUTE [6] for C, KLEE [7] which works at the level of LLVM code, Symbolic Java PathFinder [8] and jCUTE for Java, and the SAGE tool [9] which works at the level of x86 binaries, to name a few. Overall, it is fair to say that so far concolic testing has been applied, with reasonable success, mostly in the context of imperative or low-level languages.

In this article, we propose and demonstrate the use of concolic execution for testing functional programs at the level of their core language. At this level, concolic execution needs to address several challenges. For starters, in most functional languages, it needs to handle arbitrary precision integer arithmetic. It also needs to take into account pattern matching against structured data types such as lists and tuples and user-defined recursive data types. In addition, it needs to deal effectively with higher-order functions, recursion, and functions whose code is not available because they are implemented as built-ins. Finally, in our setting, that of Erlang which is a dynamically typed functional language, the test generation component of a concolic testing tool needs to be faithful to the operational semantics of the language and able to generate new inputs of arbitrary terms, not necessarily terms of some particular type. Naturally, an effective tool has to be able to take type information into account when such information is available.

This article is a revised and extended version of a paper with the same title that appeared in the proceedings of PPDP 2015 [10]. Compared with that paper, this article (i) handles a bigger part of the functional subset of Erlang; (ii) extends and improves the formalization; (iii) describes how concolic execution of functional programs can and needs to take advantage of pattern matching compilation (Section 6.1); (iv) discusses issues related to the encoding of type definitions given to the SMT solver (Section 6.2); and (v) presents more experiences and measurements (Section 7), especially regarding the coverage the current version of CutEr, the concolic testing tool we present in this article, achieves on some modules from the Erlang standard library (Section 2).

Our work is the first significant effort to apply concolic execution in the context of a functional language and create a testing tool aiming to maximize path coverage (up to a depth limit), in program units of considerable size. Recently, Palacios and Vidal presented an approach for the concolic execution of a significantly smaller subset of Core Erlang based on a program instrumentation [11]. Although they argue that their approach is “easier to maintain and may scale up better” they do not present any tool that substantiates this claim or allows comparison with the approach we have chosen. Other related work in the context of declarative languages includes an approach to defining a framework for concolic execution of pure Prolog programs aiming only at statement coverage [12], and a recent work that aims to provide choice coverage of logic programs [13].

The rest of the article describes how concolic execution occurs in the context of a functional language core (Sections 3 and 4), and presents CutEr, a concolic unit testing tool for Erlang that we have developed (Section 5) and issues that are crucial to pay attention to when implementing an efficient concolic testing tool for a functional language (Section 6). Some experiences from using CutEr are presented in Section 7. To make the article self-contained, we begin with some background information.

2. Background

This section briefly reviews concolic testing and Erlang. It also presents a program that we will use as a running example in the rest of the article.

2.1. Concolic testing

Concolic testing [14,5,6] (i.e., testing based on a combination of concrete and symbolic execution, which is also known as *dynamic symbolic execution*) is a method for test input generation where a given program is simultaneously executed both concretely and symbolically in order to achieve high path coverage; if possible, explore all paths of a program (more on that in Section 3.1). In concolic testing, test inputs are generated from the execution of the actual program instead of its model. The main idea behind this testing approach is to collect, during runtime, symbolic constraints on program inputs that specify the possible input values that force the program to follow a specific execution path. Symbolic execution is made possible by instrumenting the program with additional code that collects the constraints without disrupting its concrete execution.

In concolic testing, each variable that has a value depending on inputs to the program has also a symbolic value associated to it. When a (sequential) program is executed, the same execution path is followed regardless of the input values until a branching statement is encountered that selects one of its branches based on some variable that has a symbolic value. Given this symbolic value, it is possible to reason about the outcome of the statement symbolically by constructing a symbolic constraint. This constraint describes the possible input values that cause the program to take one of the branches at the branching statement in question. A *path constraint* is a conjunction of symbolic constraints that describe the input values causing the concrete execution to follow a specific feasible execution path.

In a concolic testing tool, the program under test is first executed with arbitrary concrete input values. During this initial test run, symbolic execution is used to collect the path constraints expressed in an appropriate logic, for each of the

branching statements along the execution. These collected constraints are used to compute new test inputs to the program by using off-the-shelf constraint solvers. Typical solvers used are SMT (Satisfiability Modulo Theories) solvers, and typical theories include linear integer arithmetic, arrays, and bit-vectors. The new test inputs will steer the future test runs to explore previously untested execution paths. This means that concolic testing can be seen as a method that systematically explores all the distinct execution paths of a program. These execution paths can be expressed as a *symbolic execution tree*, which is a structure where each path from root to a leaf node represents an execution path and each leaf node has a path constraint describing the input values that force the program to follow that specific path.

The concrete execution in concolic testing brings the benefit of making available accurate information about the program state, which might not be easily accessible when, e.g., using random testing or static analysis techniques. It is possible to under-approximate the set of possible execution paths by using concrete values instead of symbolic values in cases where symbolic execution is not possible (e.g., when there are calls to libraries of which neither source code nor other information about them is available). Furthermore, as each test is run concretely, concolic testing does not report spurious defects. As a result of all these, various researchers have argued that for sequential programs concolic testing is more effective than random testing techniques [5,6,8,9].

2.2. Erlang

Erlang [15] is a strict, dynamically typed functional programming language that comes with built-in support for actor-based message-passing concurrency, interprocess communication, distribution, and fault-tolerance. Although the syntax of Erlang is heavily influenced by logic programming languages such as Prolog (e.g., all variables in Erlang start with a capital letter or an underscore, the same list notation is used, function definitions end with a dot, etc.), its core is similar to those of modern functional programming languages such as ML or Haskell. In particular, Erlang variables are single-assignment, clause selection happens using pattern matching extended with guards, the language is higher-order and features function closures, list comprehensions, etc. On the other hand, Erlang does not support currying. All functions, besides a module and a function symbol, also have an arity (the number of arguments) as part of their name. As an example of a higher-order Erlang function, we show the definition of function `lists:foreach/2`, i.e., a `foreach` function defined in the `lists` module of the standard library, taking two arguments: a function of arity one in the first argument and a list in the second.

```
foreach(F, [H|T]) -> F(H), foreach(F, T);
foreach(F, []) when is_function(F, 1) -> ok.
```

This function definition has two clauses. Clause selection happens using pattern matching, examining the clauses from top to bottom. Actually, in this particular case, the patterns in the second argument of the function make the two clauses mutually exclusive and their order does not matter. Note however that there is no requirement that pattern matching is exhaustive, and the Erlang compiler does not warn for it. Moreover, as mentioned, the language is dynamically typed and there is no guarantee that function calls will always have the right argument types. Instead, the compiler inserts an implicit catch-all clause as a last clause of all function definitions, which throws a so called `badmatch` exception. Thus, the definition above is implicitly the same as:

```
foreach(F, [H|T]) -> F(H), foreach(F, T);
foreach(F, []) when is_function(F, 1) -> ok;
foreach(_, _) -> erlang:error(badmatch).
```

In this definition, the third clause will match if the function is not called with a proper (i.e., nil-terminated) list in its second argument, or if the function is called with the empty list in its second argument but its first argument is not a function of arity one. (If the second argument is a non-empty list but the first argument is not a function or does not have the right arity, a runtime error will happen at the `F(H)` call.)

In Erlang, pattern matching invokes the built-in `=/2`, which does not require that any variables in its left argument are unbound or occur only once. This provides a very powerful mechanism for specifying program *assertions*. For example, the pattern matching expression `[42,X,X|_] = f()` asserts that the function call will return a list of length at least three, whose first element is the integer 42 and its second and third elements are the same term, perhaps some specific one if `X` was previously bound. A `badmatch` exception, if raised, signifies that this assertion is violated and an error is found. This mechanism forms the basis of EUnit [16], Erlang's *unit testing* framework. The same mechanism can be used for *concolic unit testing*.

2.3. Running example

Rather than relying on manually written EUnit tests, our aim is to discover assertion violations and pattern matching exceptions fully automatically using concolic execution. We will explain how this is done with the program shown in Fig. 1. Erlang code resides in modules containing function definitions. Some of these functions, such as `foo/1` here, are exported and can be called from other modules using module-qualified, so called *remote*, calls. The remaining functions are *local* to the module. A function name can be used as a higher-order argument using the `fun` keyword.

The example program is small, but contains most of the elements of the language that need to be handled by a concolic testing tool. From data types, it involves simple types such as numbers and atoms (`gt`, `eq`, ...), and recursive structured

```
-module(example).
-export([foo/1]).

foo(L) ->
  lists:foreach(fun fcmp/1, L).

fcmp(X) ->
  case cmp(X) of
    gt -> ok;
    lt -> ok
  end.

cmp(X) when X > 42 -> gt;
cmp(42) -> eq;
cmp(X) when X < 42 -> lt.
```

Fig. 1. The running example.

types such as lists. Pattern matching is used with patterns only (in function `fcmp/1`) but also in conjunction with guards that call built-ins of the language (in function `cmp/1`) that do not have any definition in Erlang itself. Finally, there is a call to a higher-order function defined in another module, namely a call to the `lists:foreach/2` function whose code we presented in the previous section. The concolic execution of the program unit rooted by the entry point of function `foo/1` needs to include that code.

As a final note we mention that Erlang comes with a defined *total* ordering of all terms. In particular, the `</2` and `>/2` operators perform *term comparison*, not just comparison between numbers. That is, besides arithmetic inequalities such as `3 < 5`, `4.2 < 5.1` and `7.1 < 8`, which evaluate to `true` (notice how the latter correctly compares different types of numbers), all pairs of terms are comparable and the following (term) inequalities are also true in Erlang: `[1, 2, 3] < [1, 4]` (two lists are compared lexicographically), `17 < {ok, 42}` (an integer is smaller than a tuple), and `{ok, 42} < [17]` (a tuple is smaller than a list).

3. Concolic execution for Erlang

This section outlines, in an informal fashion and using our running example, the way in which concolic testing can be applied to Erlang programs. Our notion of bug finding is to find inputs that, if given to a program, will force execution to follow paths that terminate with a runtime error. In Erlang, a runtime error is the occurrence of an assertion violation or an unhandled exception, abiding to the philosophy of dynamically typed functional languages.

Testing is performed on a *program unit*, which is a self-contained piece of code. For Erlang, it is reasonable to assume that a unit is always a function exported from a module, which acts as the entry point for execution. The parameters of this function are considered the input of the unit test. We also assume that some starting values of these parameters are either provided by the user or are generated randomly; these values will act as the seed in concolic testing. In the case of our running example, the entry point is function `example:foo/1` and the initial input could be `L = [17]`.

In practice, unit testing often requires certain setup and teardown operations, executed independently of the actual unit under test. For such cases, we choose not to provide any scaffolding fixtures (for now, at least) but we ask the programmers to define such unit tests by explicitly wrapping the units whose code they want to test. The wrapper, which must also be an exported function, should first build an appropriate environment and/or state, and subsequently call the unit to be tested with the proper parameters. Once the execution ends, it should also perform any cleanup needed.

3.1. Concolic execution of the running example

As mentioned, many concolic tools employ tracing and emulation on a low-level representation of the program's code and data. This approach has the advantage of emulating optimized code. It works well for languages like C, Java and the languages of the .NET family, especially when the type system does not change dramatically from source code to bytecode. Erlang, on the other hand, does not have a static type system; programs heavily use list and tuple values. If a concolic tool used a low-level representation for the program's code and data, it would have to retain type information of (more or less) the same high level as in the original source code. In our approach, we choose to directly emulate execution of high-level source code.

However, the original Erlang source code is not suitable for use in concolic testing, as it is more expressive than necessary. Fortunately, there exists a suitable intermediate representation, Core Erlang [17], that is used internally by the Erlang compiler as a mid-level tier between the high-level textual representation of the source code and the low-level bytecode. It has simple semantics that allow for a straightforward translation from Erlang and its main goal is to facilitate the development of tools that operate on the Erlang source code. Furthermore, the Erlang compiler provides a module that translates Erlang source to Core Erlang in *Abstract Syntax Tree* (AST) form.

```
module example [foo/1] =
  foo/1 = fun (_cor0) ->
    call lists:foreach (fcmp/1, _cor0)

  fcmp/1 = fun (_cor0) ->
    case <apply cmp/1 (_cor0)> of
      <gt>    when true -> ok
      <lt>    when true -> ok
      <_cor1> when true -> FAIL
    end

  cmp/1 = fun (_cor0) ->
    case <_cor0> of
      <X>     when call erlang:'>' (_cor0, 42) -> gt
      <42>    when true                        -> eq
      <X>     when call erlang:'<' (_cor0, 42) -> lt
      <_cor1> when true                        -> FAIL
    end
```

```
module lists [..., foreach/2, ...] =
  ...
  foreach/2 = fun (_cor1, _cor0) ->
    case <_cor1,_cor0> of
      <F,[H|T]> when true ->
        do apply F (H)
           apply foreach/2 (F, T)
      <F,[]> when call erlang:is_function (_cor1, 1) -> ok
      <_cor3,_cor2> when true                        -> FAIL
    end
  ...
```

Fig. 2. Simplified Core Erlang code for the running example.

The running example translates to the Core Erlang code shown in Fig. 2. (We took the liberty of slightly simplifying this code, omitting details that were irrelevant to the purpose of this article and would probably confuse readers unfamiliar with Erlang.) As part of the translation, fresh variables need to be introduced: they all start with the prefix `_cor`.

The most important implication of this transformation is that different function clauses have been merged into one, whose body contains an outer `case` expression. Each branch of a `case` expression consists of a *pattern* and a *guard*. Notice that the translation introduces branches corresponding to pattern matching failure; e.g., the last branch in the definition of `fcmp/1` will be used if the function is called with a value of `X` that is not equal to `gt` or `lt`. In this case, an exception will occur, which is shown here with the shorthand `FAIL` instead of an `erlang:error(badmatch)` call. As we have already explained, these unhandled exceptions are the type of runtime errors that we want to detect. Notice also the use of Erlang built-ins, defined in the `erlang` module, such as `erlang:'>'/2` and `erlang:is_function/2`. Their code is not shown; they are implemented in C anyway.

Fig. 3 depicts the control-flow graphs of the four functions involved in the running example, including `lists:foreach/2`. Pink nodes represent the entry points; yellow nodes represent function calls and intermediate actions, such as assignments; blue diamond-shaped nodes represent decision points; green nodes correspond to returned results; finally, the red "FAIL" nodes correspond to unhandled exceptions. Decision nodes correspond to pattern matching conditions. Each such node has two outgoing edges, which carry labels of the form "T@i" (*true*, for a successful pattern matching) and "F@i" (*false*, for an unsuccessful one), where "i" is a unique integer identifying the decision node. The usefulness of the labels will become apparent shortly.

In Fig. 4 (left) we see a trace of the initial execution `example:foo/1([17])`, this execution follows a path spanning the control-flow graphs of Fig. 3. Along the blue execution path, the labels show the outcome of each decision node. Also, at each such node, the red edge extending to the right shows the path that would have been taken, had the outcome at the said decision node been the opposite one. As explained in Section 2.1, in concolic testing, execution proceeds on two parallel fronts, keeping track simultaneously of concrete and of symbolic values for all program variables. In Fig. 4, both concrete and symbolic bindings are shown (unless they are equal, in which case the notation is simplified). For example, `X` $\mapsto$ `17; hd(L)` means that parameter `X` of function `fcmp/1` has the concrete value `17` and, at the same time, the symbolic value `hd(L)`, that is, it is the head of the list `L` that was passed as the initial input to the entry point. (For simplicity, we have kept the names of function arguments from the original Erlang source code, instead of using the automatically generated ones introduced by the translation to Core Erlang.)

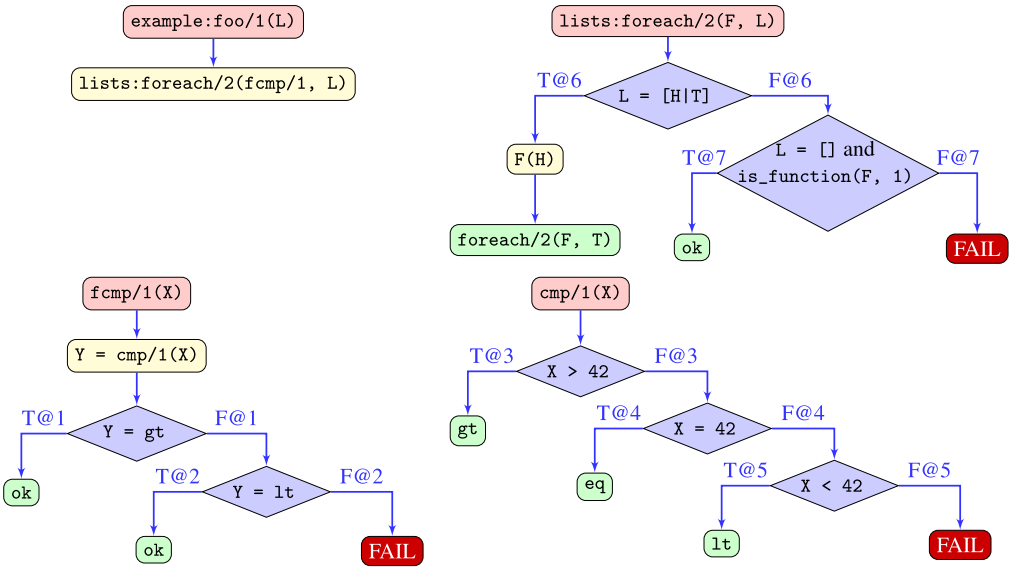

Fig. 3. Control flow graphs for all the functions of the example. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

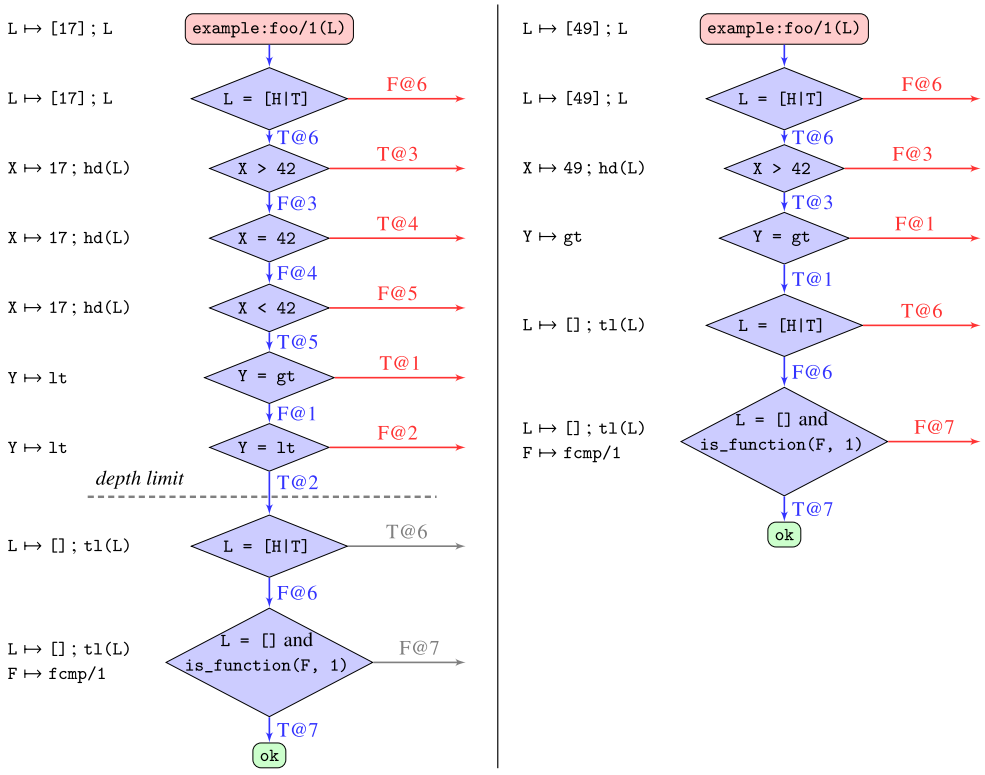

Fig. 4. Execution of `example:foo([17])` (left side) and execution of `example:foo([49])` (right side), in the running example. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

The initial execution trace, shown on the left half of Fig. 4, results in the value `ok` being returned. Concolic execution proceeds by considering each decision node in this initial trace and by trying to *reverse* the decision, thus exploring additional execution paths.

As mentioned, concolic execution aims to explore all execution paths of a program unit, i.e., maximize *path coverage* for this unit. Doing so is not always possible. For starters, some paths may be infeasible, because one of their edges is an

outgoing edge out of a decision node whose test always follows the other edge. Obviously, these paths cannot be explored, but this is not a problem as far as exhaustive testing of the program unit is concerned. However, even if one restricts attention to just feasible paths, in programs containing recursion, the set of feasible paths may be infinite or at least extremely large and one may want to employ some mechanism that bounds the concolic execution process and thus the number of paths that will be explored. Such a mechanism can be based on a timeout ("test for t time units"), on a limit of how many paths to explore ("explore p different paths"), or on a *depth limit* ("explore paths that consist of at most d decision nodes; consider all decision nodes after this limit is exhausted as straight-line code, i.e., do not try to reverse them and explore their other alternative(s)"). The last option is the mechanism that CutEr currently uses. In fact, the notion of *depth* that is currently used by the tool, only counts `case` expressions existing at the Core Erlang source. In this way, all constraints related to the patterns and guards of a specific `case` expression are considered to be at the same level.

Assuming that a depth limit of three is employed, i.e., three `case` expressions at the level of Core Erlang source (cf. Fig. 2), the left part of Fig. 4 contains a dotted line that shows the point where this limit is exhausted. What this limit will do is to instruct CutEr to only reverse decision nodes above this line (their alternatives are colored red) and *not* reverse decision nodes below this line (their alternatives are shown in grey).

Still, even when employing a limit based on depth and not on e.g., time, the *order* in which alternatives are attempted influences how fast the first error can be discovered, and thus is crucial for the success of the concolic testing approach in practice. Several heuristics have been proposed [18,19]. They all share a common approach. Certain criteria are used to set a priority to each condition or branch and then try to explore them in descending order of priority. The criteria vary from CFG analysis [20] to complex metrics that involve covered branches and statements and incrementally calculating the priority of each branch [21,9]. The heuristic that we use is the following:

- For each execution path that has already been tried, we choose to reverse the outcome of the decision node: (a) whose reversed (red) label has not yet been visited during concolic execution; and (b) which is closer to the root (i.e., at the smallest depth).
- We place all execution paths that have already been tried in a priority queue, ordered by criteria (a) and (b). Criterion (a) takes priority over criterion (b), i.e., we consider unvisited labels first. This corresponds to faster maximizing *clause coverage* in an attempt to examine all clauses of `case` expressions as soon as possible.
- We stop either when all possible execution paths have been covered, either fully (i.e., their choices have been exhausted) or only up to the default or user-specified depth in the search tree.

Following this heuristic, the decision node that we will first attempt to reverse is the one leading to T@3, as F@6 has already been visited. In order to follow this path, the following must be true[1]:

$$(\texttt{L = [H|T]}) \wedge (\texttt{hd(L)> 42}) \tag{1}$$

In other words, we want the condition of the first decision node to be true, thus taking edge T@6, but we want to reverse the outcome of the second decision node, thus taking edge T@3. We are therefore looking for a list that is not empty, whose head contains a term larger than 42. One possible value of `L` satisfying the above is `L = [49]`, and a new execution trace is generated for this value, shown in Fig. 4 (right). The second execution trace yields again the value `ok` and, again, no bug is found.

Now there are two paths in the priority queue, those of Fig. 4. In the initial path (left), the next alternative to be explored is T@4, whereas in the second path (right), the next alternative is F@7; both have not yet been visited. Our heuristic chooses the former, as it has smaller depth. Therefore, we are trying to satisfy:

$$(\texttt{L = [H|T]}) \wedge \neg(\texttt{hd(L)> 42}) \wedge (\texttt{hd(L)= 42}) \tag{2}$$

or, in other words, we are looking for a list that is not empty and whose head contains a term equal to 42. One possible solution is `L = [42]`, which leads us to the third execution trace. This time, this produces an unhandled exception, as `cmp/1` returns `eq` and this value is not handled by `fcmp/1`. We have found a bug!

Concolic testing of our running example does not stop here, as all possible execution paths up to the depth limit have not been explored yet. Sooner or later, alternative F@7 on the right path will be considered and we will try to find a term `L` that satisfies:

$$\neg(\texttt{L = [H|T]}) \wedge \neg(\texttt{L = []}) \tag{3}$$

(Notice that Erlang is a dynamically typed language and, in the code of the running example, there is nothing restraining `L` to be a list.) This is possible if, e.g., `L = 0` and this will be the next reported bug.

Also, notice that the "FAIL" node in `cmp/1`, is indeed reachable by edge F@5 (the only remaining unvisited edge). This may seem strange, at first, as it implies finding a term `X` satisfying:

$$\neg(\texttt{X > 42}) \wedge \neg(\texttt{X = 42}) \wedge \neg(\texttt{X < 42}) \tag{4}$$

[1] Note that, when used as a constraint in the rest of the article, `L = [H|T]` usually means that the term `L` is a cons cell; the bindings of its head and its tail to `H` and `T` respectively are not relevant. However, to make this consistent with the control-flow graphs of Fig. 3, we keep this notation in constraint paths.

$$
\begin{array}{lll}
\mathit{module} & ::= & \texttt{module}\ \mathit{Atom}\ \texttt{[}\ \mathit{fname}_1, \ldots \mathit{fname}_m\ \texttt{]} = \mathit{def}_1 \ldots \mathit{def}_n \\
\mathit{def} & ::= & \mathit{fname} = \mathit{fun} \\
\mathit{fun} & ::= & \texttt{fun (}\ \mathit{Var}_1, \ldots \mathit{Var}_n\ \texttt{) ->}\ \mathit{expr} \\
\mathit{fname} & ::= & \mathit{Atom} \mid \mathit{Integer} \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid \texttt{[]} \\
\mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid \mathit{fun} \mid \texttt{[}\ \mathit{expr}_1\ \texttt{|}\ \mathit{expr}_2\ \texttt{]} \mid \texttt{\{}\ \mathit{expr}_1, \ldots \mathit{expr}_n\ \texttt{\}} \\
 & \mid & \ll \mathit{bitexpr}_1, \ldots \mathit{bitexpr}_n \gg \mid \texttt{let}\ \mathit{Var} = \mathit{expr}_1\ \texttt{in}\ \mathit{expr}_2 \mid \texttt{do}\ \mathit{expr}_1\ \mathit{expr}_2 \\
 & \mid & \texttt{apply}\ \mathit{expr}\ \texttt{(}\ \mathit{expr}_1, \ldots \mathit{expr}_n\ \texttt{)} \mid \texttt{call}\ \mathit{expr}_m : \mathit{expr}_f\ \texttt{(}\ \mathit{expr}_1, \ldots \mathit{expr}_n\ \texttt{)} \\
 & \mid & \texttt{case <}\ \mathit{expr}_1, \ldots \mathit{expr}_n\ \texttt{>}\ \texttt{of}\ \mathit{clause}_1; \ldots \mathit{clause}_m\ \texttt{end} \\
\mathit{bitexpr} & ::= & \mathit{expr} \mid \mathit{expr} : \mathit{expr} \mid \mathit{expr} / \mathit{type} \mid \mathit{expr} : \mathit{expr} / \mathit{type} \\
\mathit{clause} & ::= & \texttt{<}\ \mathit{pat}_1, \ldots \mathit{pat}_n\ \texttt{>}\ \texttt{when}\ \mathit{expr}_1\ \texttt{->}\ \mathit{expr}_2 \\
\mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid \texttt{[}\ \mathit{pat}_1\ \texttt{|}\ \mathit{pat}_2\ \texttt{]} \mid \texttt{\{}\ \mathit{pat}_1, \ldots \mathit{pat}_n\ \texttt{\}} \mid \ll \mathit{bitpat}_1, \ldots \mathit{bitpat}_n \gg \\
\mathit{bitpat} & ::= & \mathit{pat} \mid \mathit{pat} : \mathit{expr} \mid \mathit{pat} / \mathit{type} \mid \mathit{pat} : \mathit{expr} / \mathit{type} \\
\mathit{type} & ::= & \texttt{integer} \mid \texttt{float} \mid \texttt{binary} \mid \texttt{bitstring}
\end{array}
$$

Fig. 5. The syntax of Mini Core Erlang.

However, according to the semantics of Erlang, `=` denotes pattern matching, i.e., exact term equality (for numbers, arithmetic equality that coerces integers to floats is denoted by `==`). The term `42.0` is neither smaller nor larger than `42`, nor does it match with `42`. (In fact, `42.0` is the only Erlang term for which function `cmp/1` throws an exception.) This results in one more bug found, manifested by `L = [42.0]`.

To sum up, concolic execution for our running example starts by the user simply specifying an optional depth limit and an arbitrary call to the function constituting the entry point of the unit being tested (in this case `example:foo([17])`) as a seed. The process automatically discovers three other calls (`example:foo([42])`, `example:foo(0)`, and `example:foo([42.0])`) that result in three different runtime exceptions for this program unit. They correspond to the three places in Fig. 2 where failures can occur.

3.2. Mini Core Erlang

For presentation purposes, we define here a subset of Core Erlang that will be used in the following sections. The syntax of Mini Core Erlang is defined in Fig. 5.

In Mini Core Erlang, terms can be atoms, integers, floats, lists, tuples, bitstrings or functions. A module is a list of function definitions (n in total) where some ($m \leq n$) of these functions are exported. A function definition consists of the function's name and a function literal, which contains the function's parameters and body. Expressions also include `let`, `do`, `apply`, `call` and `case`. The semantics of `let` is straightforward, `do` represents sequential execution, whereas `call` and `apply` represent function application. Notice that `apply` is used for module-local function applications, whereas `call` is used for module-qualified ones. Built-in functions, such as comparison operators, `is_function/2`, etc., can be seen as predefined functions in Core Erlang. Most Erlang built-ins are part of the special module `erlang`, e.g., `erlang:'<'/2`. Bitstring expressions consist of sequences of *bitstring segments*, each of which is a sequence of bits. In each segment, the number of bits and the intended representation type can be optionally specified.

Case expressions are the most interesting control statements in this grammar. The sequence of supplied expressions is matched against a list of guarded patterns (clauses), tried in the order in which they appear. Evaluation continues from the body of the first clause that matches. As mentioned, the Erlang compiler ensures that all `case` expressions become exhaustive by adding catch-all clauses if it cannot verify that the programmer has provided an exhaustive set of clauses.

We should note that Core Erlang is more expressive than the language defined in Fig. 5, e.g., it supports local `letrec` definitions, `try-catch` blocks for exception handling, and `receive` expressions for message passing between processes. Here we have restricted the language only for presentation purposes. The tool that we have developed (described in Section 5) handles the complete set of Core Erlang expressions.

3.3. Constraint generation for patterns and guards

Concolic execution involves running the program both concretely and symbolically. Symbolic execution follows the execution path dictated by the concrete execution. We perform these two tasks simultaneously. A program is essentially interpreted by evaluating its Core Erlang representation, where every node of the AST is evaluated both concretely and symbolically.

During evaluation, we keep two separate environments: one mapping variables to *concrete values*, and one mapping variables to *symbolic values*. Both types of values are subsets of terms ($Val \subset expr$); environments map variables to such values:

$$Env = Var \rightarrow Val$$

We will denote the concrete environment by Γ_c and the symbolic environment by Γ_s.

The evaluation function eval takes an expression and the two environments. It returns two values, one concrete and one symbolic, representing the result of the evaluation. It also returns the execution path that led to the returned result.

$$\text{eval} : expr \times Env \times Env \rightarrow Val \times Val \times Path$$

Paths are lists of nodes, such as the ones shown in Fig. 4. For each node, we keep the *logical proposition* that corresponds to a test performed during evaluation (e.g., a pattern matching) and the two *labels*: first the one that was followed during evaluation, and then the one that possibly remains to be followed by a subsequent evaluation.

$$Path = \{\bullet\} \cup (Prop \times Label \times Label \times Path)$$

For evaluating a variable, we simply look up its value in the concrete and the symbolic environment. Evaluating literals is even simpler; both values coincide with the literal itself and, in both cases, the execution path is empty. Evaluating lists and tuples is a bit more involved. Their subexpressions must first be evaluated; their concrete and their symbolic values form, respectively, the concrete and the symbolic value of the result. Furthermore, the execution path is the concatenation of the execution paths of the subexpressions. The same approach applies to bitstrings. Each bitstring consists of a list of subexpressions that form the individual bitstring segments. Once all segments are evaluated, the resulting bitstring is their concatenation.

The evaluation of `let` and `do` expressions is also straightforward. The two subexpressions must be evaluated and their execution paths are again concatenated. In the case of `let`, the second subexpression is evaluated in a concrete and a symbolic environment that have been extended with the bound variable, mapped to the concrete and the symbolic result of the first subexpression, respectively.

Evaluating `call` and `apply` expressions again requires that subexpressions be evaluated. The concrete value of the function to be applied is then used to determine the function's body, which starts being evaluated with the updated concrete and symbolic environments (mapping the function's formal parameters to the concrete and symbolic values of the actual parameters, respectively). Again, the execution path is the concatenation of the execution paths of the subexpressions and the function's body.

It is the evaluation of `case` expressions that actually generates constraints, adding nodes to execution paths. When a `case` is evaluated, its sequence of subexpressions must first be evaluated. Subsequently, pattern matching is driven by the concrete values. Whenever a pattern match is attempted, a decision node is generated and added to the execution path. The order in which the labels appear depends on whether the match was successful (in which case the bottom label is T@i and the side label is F@i) or unsuccessful (in which case, the two labels are reversed). The evaluation of guards also generates decision nodes; a guard is roughly equivalent to one more subexpression in the sequence of subexpressions that a `case` contains, pattern matched against `true` and `false`.

Let us see, for example, what happens during the evaluation of `example:foo([17])` of the running example. Evaluation starts in the body of `example:foo/1` with:

$$\Gamma_c = \{\texttt{L} \mapsto \texttt{[17]}\}$$

$$\Gamma_s = \{\texttt{L} \mapsto \texttt{L}\}$$

where the last `L` is a symbolic variable corresponding to the initial input. When evaluation reaches the `case` expression in the body of `lists:foreach/2`, the two environments are:

$$\Gamma_c = \{\texttt{F} \mapsto \texttt{fcmp/1}, \texttt{L} \mapsto \texttt{[17]}\}$$

$$\Gamma_s = \{\texttt{F} \mapsto \texttt{fcmp/1}, \texttt{L} \mapsto \texttt{L}\}$$

The first pattern matches with the concrete value of `L`, which is indeed a non-empty list, binding `H` and `T` to the head and tail of this list, respectively. Just before `F(H)` is evaluated, the two environments are:

$$\Gamma_c = \{\texttt{F} \mapsto \texttt{fcmp/1}, \texttt{L} \mapsto \texttt{[17]}, \texttt{H} \mapsto \texttt{17}, \texttt{T} \mapsto \texttt{[]}\}$$

$$\Gamma_s = \{\texttt{F} \mapsto \texttt{fcmp/1}, \texttt{L} \mapsto \texttt{L}, \texttt{H} \mapsto \texttt{hd(L)}, \texttt{T} \mapsto \texttt{tl(L)}\}$$

Furthermore, the decision node with the constraint "`L = [H|T]`" is added to the execution path, following label T@6 and leaving label F@6 for further exploration. Notice that the guard expression always evaluates to `true` and therefore no decision node needs to be generated for it, in this case.

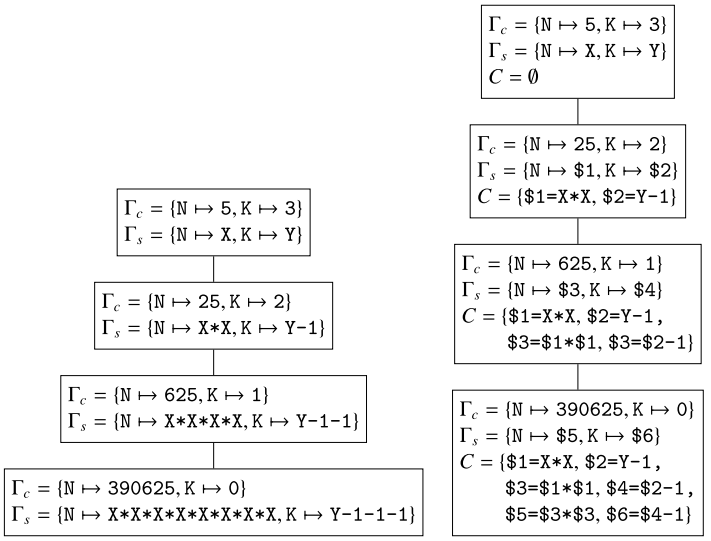

Fig. 6. The concrete and symbolic environments of all the recursive calls of `sq(5,3)` without introducing new symbolic variables (on the left) and when using a set C of new symbolic variables (on the right). The size of the symbolic environments on the left grows exponentially, while those on the right only linearly.

3.4. Constraint generation for built-in functions

Whenever concolic execution reaches a function call, we try to access the source code of the function in order to interpret it. We can also do the same in the case of library functions, such as `lists:foreach/2`, as long as we have access to their source code. However, some functions are preloaded to the Erlang runtime. Such functions are called *built-in functions* (BIFs) and are typically written in C. Most of them belong to the `erlang` module, but there are more in other commonly used modules. For such functions, we do not have access to their bodies in Core Erlang form, and therefore we cannot evaluate them. Examples of BIFs that we already saw in our example are `erlang:'>'/2` and `erlang:is_function/2`.

For built-in functions, we can easily evaluate the concrete value of the returned result, by calling them directly with the concrete arguments. It is not clear, however, what the symbolic value of the result should be.

The obvious approach is to introduce them directly as uninterpreted functions to our language of symbolic expressions. For example, in the environment:

$$\Gamma_s = \{\texttt{H} \mapsto \texttt{hd(L)}\}$$

the symbolic value of expression `H < 42 or H > 42` will be `hd(L)< 42 or hd(L)> 42`. However, this approach may lead to memory and performance issues. Consider the following function that uses the arithmetic built-ins for multiplication and subtraction:

```
sq(N, 0) -> N;
sq(N, K) -> sq(N * N, K - 1).
```

On the left part of Fig. 6, we show the concrete and symbolic environments for each of the recursive calls in the execution of `sq(5, 3)`. For simplicity, we assign `X` and `Y` as the initial symbolic values of the parameters.

With this representation the concolic execution of the program requires exponentially more memory than that required for its concrete execution. The symbolic representations of `N` and `K` grow exponentially and linearly respectively, whereas their concrete representations remain constant.

This is not only a memory consumption problem, but it also affects the time performance of the tool. Symbolic values that participate in decision nodes are serialized and recorded in logs which are later passed to the constraint solver. Their large size increases the memory usage and slows down both of these components. An additional Erlang-specific issue, which is magnified by this behavior, is that term sharing is not always preserved by the Erlang VM [22].

A better representation is to introduce a fresh symbolic variable for the result of each operation, and use these variables to avoid the possible exponential growth of symbolic expressions. As we can see on the right part of Fig. 6, the size of the symbolic environment now remains constant and only depends on the number of variables in it. The set of constraints C for the intermediate variables increases linearly as we add more variables, but there is no need to keep it in memory and we can store it in a file. Using this approach, in our running example with the comparisons, in the places where the comparison BIFs `erlang:'<'/2`, `erlang:'>'/2` and `erlang:or/2` are used, three fresh variables `T1`, `T2` and `T3` are introduced to the symbolic environment; the symbolic result is `T3`.

$$\Gamma_s = \left\{ \begin{array}{ll} \texttt{H} \mapsto \texttt{hd(L)}, & \texttt{T1} \mapsto \texttt{H < 42}, \\ \texttt{T2} \mapsto \texttt{H > 42}, & \texttt{T3} \mapsto \texttt{T1 or T2} \end{array} \right\}$$

Of course, having BIFs like `erlang:'<'/2`, `erlang:'>'/2`, and `erlang:or/2` as uninterpreted functions in our symbolic expressions is not sufficient for our purpose. If we are to use such expressions in the generated constraints, the solver that we will use needs to interpret them, specifically to treat them as term comparison operators and logical disjunction, faithfully to the semantics of Erlang. For this to happen, equivalent operators must exist or be representable in the logic of the constraint solver. This is true in the case of arithmetic, comparison and logical operators, or BIFs such as `erlang:hd/1` and `erlang:tl/1`.

For other BIFs, we can follow a different approach: we can replace them with equivalent, ordinary functions, written in Erlang, and evaluate those symbolically. As an example, the BIF `erlang:length/1` returns the length of a list and cannot be represented efficiently in the logic of most constraint solvers, because of its recursive nature. It is, however, equivalent to the following, which can be transformed to Core Erlang and executed symbolically:

```
length([]) -> 0;
length([H|T]) -> 1 + length(T).
```

As a last resort, in the case of BIFs which are too complicated or impossible to express in the constraint solver's logic, nor to emulate using Erlang code, we can ignore symbolic evaluation completely and use the concrete result also in the place of the symbolic value. We try to avoid this option, if possible, as we completely lose track of the data flow of symbolic values. However, this is the only viable option for BIFs such as `os:timestamp/0`. Furthermore, disabling symbolic evaluation allows us to incrementally support progressively larger subsets of Erlang BIFs.

Some BIFs impose additional constraints on the values of their arguments. For example, roughly speaking, `erlang:'+'/2` requires that both its arguments be numeric values. When such BIFs are evaluated, additional decision nodes must be generated to reflect these constraints. This is automatically achieved if these BIFs are emulated, as suggested above, using Erlang code.

3.5. Constraint solving

As explained in Section 3.1, concolic testing is driven by constraint solving. After each execution, a new input must be found that will drive execution to yet unexplored paths. Constraint solving is the process of creating a properly encoded path predicate, based on the symbolic constraints that were recorded along an execution path, negating the selected constraint and then invoking a suitable solver to find a solution (i.e., a mapping of the input's symbolic variables to Erlang terms) that satisfies the constraints. This solution will be used as the input for the next execution.

There are several types of constraints generated by the process described in Sections 3.3 and 3.4. Some of them involve determining the type of an Erlang term, e.g., whether `X` is an atom, an integer, a non-empty list, a tuple of five elements, etc. Others involve equality with specific term literals, e.g., whether `X` is identical to the term `42`. Finally, some constraints are specific to BIFs that implement arithmetic, comparison, and logical operators, e.g., whether `X` equals the sum of `Y` and `Z`, whether it is greater than the term `W`, etc. A set of constraints to be solved is a set of such simple constraints or their negations, interpreted as a logical conjunction.

As the solution to such constraints is a set of Erlang terms that will be used as the next program input, the constraint solver needs to be aware of the structure of Erlang terms. If the solver's logic is not expressive enough to represent terms as elements of a "data type" or equivalent definition, then they will have to be appropriately encoded using values supported by the solver, such as integer numbers.

We must always keep in mind that most constraint solvers are incomplete. A solution may exist for a set of constraints, yet a solver may be unable to find it. Typically, in this case, a solver will either return "unknown" as an answer, or will take a long time trying to find the solution and, eventually, time out. In both cases, there is not much we can do: we have to proceed to the next execution path and try to negate the next possible constraint. Similarly, this is also what we do if a solver answers that a set of constraints is unsatisfiable.

4. Support for type specifications

Let us come back to our running example. Recall that, starting from the seed `example:foo([17])`, concolic testing revealed three input terms that crashed the test unit, namely `[42]`, `0`, and `[42.0]`. The last two may come as a surprise to readers used to statically typed programming languages. In fact, because of the presence of the call `lists:foreach(fun fcmp/1, L)` in `example:foo/1`, most Erlang programmers would not consider the case `L=0` as an actionable error, on the grounds that `lists:foreach/2` should take a list as its second argument, and `0` is not a list.

Although Erlang is a dynamically typed language, it supports a notation for declaring sets of Erlang terms that belong to specific *types*. These types can then be used to provide function *specifications*, in other words, to specify the subset of terms that form a function's intended arguments and the subset of terms that may be returned as the function's result. Besides documentation, such type information can be used by tools, such as Dialyzer [23], performing static analyses to detect definite type errors.

A simplified specification of `lists:foreach/2` reads:

```
-spec foreach(fun((T) -> term()), [T]) -> ok.
```

According to this, the function expects two arguments: (1) a function expecting an argument of type `T` and returning an Erlang term (of some unspecified type), and (2) a proper list of elements of type `T`. It can only return the atom `ok`. Type `T` can be any subtype of `term()`.

If this specification is taken into account, then `example:foo/1` can be given, by the programmer or automatically by a tool such as TypEr [24], the following specification:

```
-spec foo([term()]) -> ok.
```

This expresses the constraint that this function is to be called with proper lists as arguments, not any Erlang term.

Type specifications can be used during concolic testing to impose additional constraints on program inputs. Such constraints act as *preconditions*: they can be thought of as special nodes in the beginning of execution paths, which need to be satisfied upon program entry and are not to be negated. Were the above type specification for `example:foo/1` used in the program of Fig. 1, constrains (1) to (4) would be extended to, respectively:

1. $\texttt{is\_list(L)} \wedge (\texttt{L = [H|T]}) \wedge (\texttt{hd(L)} > 42)$
2. $\texttt{is\_list(L)} \wedge (\texttt{L = [H|T]}) \wedge \neg(\texttt{hd(L)} > 42) \wedge (\texttt{hd(L)} = 42)$
3. $\texttt{is\_list(L)} \wedge \neg(\texttt{L = [H|T]}) \wedge \neg(\texttt{L = []})$
4. $\texttt{is\_list(L)} \wedge \texttt{X = hd(L)} \wedge \neg(\texttt{X} > 42) \wedge \neg(\texttt{X} = 42) \wedge \neg(\texttt{X} < 42)$

where `is_list(L)` is a predicate that should guide the solver to only generate solutions for `L` that are proper lists of terms.

Now, the first, second and fourth are still satisfiable, producing the same solutions as in Section 3.1, namely `[49]`, `[42]` and `[42.0]`. On the other hand, the third constraint is not satisfiable anymore, as it requires a list to be neither empty, nor non-empty. Thus, the reported program input `L = 0` which leads to an unhandled exception, is now eliminated.

Going one step further, let us now suppose that the programmer has provided a stricter specification for `example:foo/1`:

```
-spec foo([integer()]) -> ok.
```

Now, the function should only be called with arguments that are lists of integer numbers (not any other type of terms). This would result in `is_integer_list(L)` being used, instead of `is_list(L)` in all four constraint paths. For the first two, the solutions would be the same as in Section 3.1, as they involve lists of integer numbers. The third one would again be ruled out as unsatisfiable, because a list of integer numbers cannot be neither empty, nor non-empty.

However, the interesting part is that now also the fourth constraint path would be ruled out as unsatisfiable, because of the following axiom, defining the stricter predicate `is_integer_list`, which would be automatically provided to the solver:

$$\texttt{is\_integer\_list(L)} \implies \texttt{L = []} \vee (\texttt{is\_cons(L)} \wedge \texttt{is\_integer(hd(L))} \wedge \texttt{is\_integer\_list(tl(L))})$$

Using this, the fourth constraint path would expand to a conjunction containing:

$$\texttt{is\_integer(X)} \wedge \neg(\texttt{X} > 42) \wedge \neg(\texttt{X} = 42) \wedge \neg(\texttt{X} < 42)$$

which would now be unsatisfiable. Therefore, with this specification, the only bug found would be `L = [42]`.

We note in passing that another way of avoiding the generation of `L = [42.0]` would be to introduce the following type specification for `cmp/1`:

```
-spec cmp(integer()) -> gt | eq | lt.
```

More generally, concolic testing of programs written in dynamically typed languages such as Erlang can start even from a program containing no specifications, or only relatively loose ones which perhaps have been automatically inserted into the code by a type inference tool such as TypEr. Then, a fully automatic process of concolic execution can generate test cases that violate assertions or result in errors. Subsequently, the user can gradually add or refine some type specifications to impose extra constraints that filter out unintended uses of functions or unwanted errors. Note however, that this process depends heavily on the expressiveness of the specification language and also on whether the additional constraints can be processed by the underlying constraint solver.

An example of a function for which Erlang's current type language is not expressive enough to fully describe its intended uses is `lists:nth/2`. Its type specification:

```
-spec nth(pos_integer(), [T,...]) -> T.
```

Although it is already more expressive than simply specifying that its first argument is an integer and its second argument is a list (in the type language of Erlang, the notation `[T,...]` specifies a non-empty list of type `T`), this type specification does not express the fact that the first argument of this function is expected to be an integer between `1` and `N`, where `N` is the length of the list in its second argument. As a result, the concolic execution of this function started with seed `lists:nth(1, [1,2])` will report that e.g., the call `lists:nth(39, [0])` leads to a runtime error.

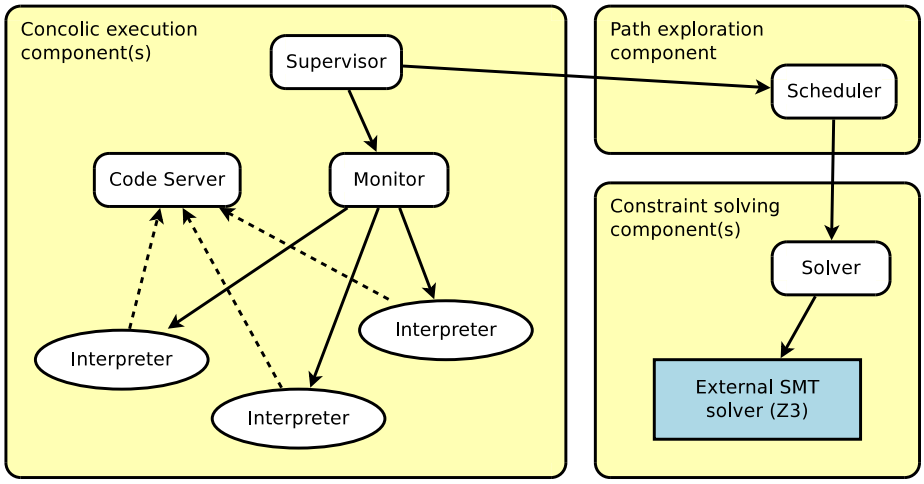

Fig. 7. High-level architecture of CutEr.

5. CutEr: concolic unit testing tool for Erlang

To demonstrate the feasibility and evaluate the proposed approach, we have developed CutEr, a concolic unit testing tool for Erlang. For the most part, CutEr itself is implemented in Erlang, with a small part implemented in Python.

CutEr aims to apply the idea of concolic testing to detect bugs in Erlang applications, especially bugs that are very difficult to find using other methods. The applications under test need not be simple sequential programs, like the ones shown in the previous sections. They can also be concurrent programs, spawning multiple processes, or they can be distributed over more than one Erlang node, possibly running on different machines over a network. CutEr currently handles concurrent and distributed applications, with limitations; a thorough discussion of how concolic testing can be applied in the presence of concurrency is beyond the scope of this article.

5.1. Architecture

CutEr comprises three main components, each responsible for a different task: one for *concolic execution*, one for *constraint solving*, and one for *execution path exploration*. In the rest of this section, we briefly present the architecture of the tool, shown in Fig. 7, and outline how these components interact with each other. The SMT solver that CutEr currently uses is Z3 [25], a solver developed at Microsoft Research. We discuss this choice further in Section 5.3. For performance reasons, CutEr's implementation may use more than one concolic execution components working in parallel to produce different execution paths. Similarly, it may use more than one constraint solving components working in parallel to solve different constraints.

During execution, a number of processes are spawned within the concolic execution component. They belong to four categories:

Interpreter processes. These are the worker processes where execution actually takes place. Each interpreter process emulates the execution of a single program process and records the symbolic constraints. If the program would spawn multiple processes, when executed by the Erlang VM, then multiple interpreter processes will be required, in a one-to-one relationship.

Code server processes. Interpreter processes need to access the Core Erlang AST of the (fragment of the) program that they execute. A dedicated type of process, the code server process, performs the task of compiling source code and providing the Core Erlang AST to the interpreter processes that ask for it. We normally spawn one code server process per Erlang node.

Monitor processes. The basic goal of CutEr is to look for exceptions in the execution of a program. To achieve this, we use a dedicated type of process, which acts as a monitor for interpreter processes and is notified by the Erlang VM, whenever an unhandled exception occurs in any of them. Again, we normally spawn one monitor process per Erlang node.

Supervisor processes. These processes are responsible for supervising concolic execution and there is one for each such execution, i.e., for each program input that is passed to the entry point. The supervisor process intercepts notifications from monitor processes and performs regulatory actions as needed. During execution, the supervisor waits to be notified for two kinds of events: either normal program termination, or the occurrence of an unhandled exception. In both cases, it notifies the path exploration component, to schedule the next concolic execution.

5.2. Heuristics and coverage

The component for execution path exploration performs a search in the space of all possible execution paths up to a depth limit. As explained in Section 3.1, the order in which execution paths are considered is crucial for the effectiveness of concolic testing when detecting errors; the search strategy that we use has already been informally described. In the rest of this section, we formalize the search strategy and outline its implementation in CutEr.

This component is implemented as a stateful server which provides two functions, StoreExecution($input$, $result$, ℓ) and RequestInput(), shown as parts of Algorithm 1.

Algorithm 1. Exploration of execution paths.

```
 1: function ConcolicTesting(unit, seed)
 2:     R := ∅                                          — errors found
 3:     Q := ∅                                          — priority queue of paths for exploration
 4:     visited := ∅                                    — the set of visited labels
 5:     D := ∅                                          — mapping of inputs to depths
 6:     D_max := 25                                     — maximum depth of the search tree
 7:     input := seed
 8:     repeat
 9:         ⟨result, _, ℓ⟩ := eval(unit(input), ∅, ∅)
10:         StoreExecution(input, result, ℓ)
11:         input := RequestInput()
12:     until input = nothing
13:     return R

14: function StoreExecution(input, result, ℓ)
15:     if result is not normal termination then
16:         R := R ∪ {⟨input, result⟩}
17:     CS := ∅
18:     i := 0
19:     for all ⟨C, L_T, L_F⟩ ∈ ℓ do
20:         visited := visited ∪ {L_T}
21:         if D[input] ≤ i ≤ D_max then
22:             Enqueue(Q, ⟨CS ∪ {¬C}, i, L_F⟩)
23:         CS := CS ∪ {C}
24:         i := i + 1

25: function RequestInput()
26:     while Q ≠ ∅ do
27:         ⟨CS, i, L_F⟩ := Dequeue(Q)
28:         answer := SmtSolve(CS)
29:         if answer = ⟨satisfiable, input⟩ then
30:             D[input] := i + 1
31:             return input
32:     return nothing
```

Both are re-entrant, so they can be used in a setting where multiple concolic executions and solvers are executed concurrently. Function ConcolicTesting($unit$, $seed$) is the entry point, performing concolic testing for a given unit and an arbitrary seed input. The **repeat**-**until** loop inside ConcolicTesting can be implemented concurrently.

Function StoreExecution($input$, $result$, ℓ) is used whenever a concolic execution is finished, to store the input and the result of execution. The complete execution path ℓ is stored together with the input and result. Let us recall that ℓ is a sequence of nodes and, in particular, decision nodes such as the ones shown in Fig. 4. Each such node is represented as a triple of the form $\langle C, L_T, L_F\rangle$, where C is a recorded constraint, L_T is the (blue) label that was followed and L_F is the (red) label that was not followed. On the other hand, function RequestInput() is used whenever a new input is required, to start a new concolic execution.

The state of the server consists of the following (initialized in ConcolicTesting):

- The set R of inputs that lead to runtime errors.
- The priority queue Q of collected path predicates that await to be supplied to the solver, in order to generate new inputs.
- The set *visited* which keeps track of the labels in the control flow graph that have been "visited", i.e., those occurring in paths that have been explored.
- The mapping D from generated inputs to integer numbers. If a program input is generated by negating the constraint of the i-th node of some execution path, then this input will be mapped to i while its concolic execution is performed.
- The maximum depth $D_{\max}$ of nodes in the search tree that we consider during exploration. Once execution goes deeper than this number, we stop recording constraints and we continue the execution only for its concrete result.

Storing execution information. As soon as execution with some input is complete, yielding some concrete result and a constraint path ℓ, StoreExecution($input, result, \ell$) is called. If *result* is an error, this instance is properly archived. Then we traverse the constraint path ℓ and generate sets of SMT constraints, at the same time updating the *visited* set with the labels that were followed. The constraints along this path that we will attempt to negate are those of order i, where i is not smaller than $D[input]$ (i.e., the point where this concolic execution started exploring possibly new paths) and not larger than $D_{\max}$. We place each such negated path in the priority queue, to be explored later in the process.

Requesting a new input. Whenever a new input is required, to start a new concolic execution, function RequestInput() is called. Assuming that Q is not empty (if it is, all paths have been explored and we can stop), an element $\langle CS, i, L_F \rangle$ is removed from the head of Q and its set of constraints is supplied to the solver. This means that the solver will try to find an input that will lead the execution along a path that coincides with a previously executed path in the first $i-1$ nodes and follows a different label at the i-th node. If the set of constraints is satisfiable, the solver will calculate a model which we interpret as a program input. We add this input to mapping D, thus stating that we are only interested in what happens after the i-th node of constraint paths, generated by this input. On the other hand, if the set of constraints is not satisfiable or if the solver is unable to solve it, we proceed with the next element in the queue.

Ordering of paths and implementation of the priority queue. The priority queue Q plays an important role in our search strategy. Elements of the form $\langle CS, i, L_F \rangle$ are placed in this queue, where CS is a set of axioms corresponding to a desired constraint path that will be given to the solver in order to find a possible model, i is the index of the node whose constraint was negated to obtain this path, and L_F is the label in the control-flow graph that will be followed, if execution follows this path. As explained in Section 3.1, the ordering of elements in the priority queue follows two rules:

1. Elements whose labels are not in the *visited* set come before elements whose labels are in the *visited* set. Therefore, we favor unvisited labels as a first priority.
2. If two elements have labels that are either both visited or both unvisited, the one with the smallest i comes first. Therefore, we perform a breadth-first search as a second priority.

As function StoreExecution updates the *visited* set, the ordering of elements in the priority queue is dynamic. From the point of view of the queue's implementation, this can be problematic. However, notice that labels are only added to the *visited* set, never removed. This means that elements already placed in the queue may have to be moved "downward", i.e., scheduled at a lower priority, never "upward"; this happens if their label becomes visited as the result of executing a different path that was higher in the queue. This remark allows us to ignore the dynamic reordering of the queue. Instead, we keep a marker in the queue separating the elements whose labels were unvisited when they were placed in the queue from those whose labels were visited. (In practice, we could equivalently keep two separate queues.) Whenever we remove an element from a position before the marker, we check its label and, if it is now visited, we immediately put it back in the queue. To simplify presentation, this optimization is not shown in Algorithm 1, where the priority queue is assumed to be ordered dynamically.

Measuring code coverage. A common metric used for assessing the effectiveness of a testing tool is code coverage. CutEr aims at comprehensive program testing by maximizing path coverage, up to a depth limit. However, accurately measuring the achieved path coverage, even when the number of paths is finite, can be quite challenging. Some of the paths may actually be infeasible, but there is no practical way of identifying them. In addition, Erlang has higher-order functions so the call graph cannot in general be completely calculated at compile time; dynamic analysis is required. In order to avoid providing misleading measurements for such a complex metric, CutEr currently reports only the achieved clause coverage as a *rough measurement* of its effectiveness.

As we have already mentioned, each decision node in the control flow graph is annotated with two labels, representing the two possible outgoing edges corresponding to the control expression being `true` or `false`. CutEr's scheduler keeps track of the set of visited tags throughout all the executions. Using this set, it reports (a) which clauses have matched, and (b) which conditions, that is patterns and corresponding guard expressions, have been evaluated. This, combined with a rough estimate of all reachable code that is obtained by folding over the Core Erlang AST and following the function calls, provides the user with the measurement of the achieved clause coverage.

5.3. Solving constraints with Z3

Z3 [25] is an efficient SMT solver, mostly targeted at solving problems in software analysis and verification. The main reason we chose Z3 was because, in addition to basic types (booleans, integers, floats, arrays and bit-vectors), it supports algebraic data types, which allow us to easily represent Erlang terms at a higher level. Z3 is a relatively low level tool, primarily intended as a resource for other tools that require SMT solving [26]. Its input format is an extension of the SMT-LIBv2.5 standard [27] but APIs are provided for C/C++, .NET, OCaml and Python. CutEr uses the Python Z3Py API; establishing a bidirectional communication between Erlang and Python was not only simpler than developing an Erlang API for Z3, but also more portable for easily supporting more SMT solvers in the future.

```
(declare-datatypes () (
    (Term       (int   (ival Int))
                (real  (rval Real))
                (atm   (aval IList))
                (lst   (lval TList))
                (tpl   (tval TList))
                (bin   (bsz Int) (bval BitVecList)))
    (TList      (nil)
                (cons  (hd Term) (tl TList)))
    (IList      (anil)
                (acons (ahd Int) (atl IList)))
    (BitVecList (bnil)
                (bcons (bhd (_ BitVec 1)) (btl BitVecList)))))
```

Fig. 8. The representation of Erlang terms in SMT-LIB notation.

$M(Var)$ = Var
$M(Integer)$ = (int $Integer$)
$M(Float)$ = (real $Float$)
$M([])$ = (lst nil)
$M(Atom)$ = (atm (acons a_1 ... (acons a_n anil)...)) where $Atom$ = '$a_1 \ldots a_n$'
$M(\{t_1, \ldots t_n\})$ = (tpl (cons $M(t_1)$ ... (cons $M(t_n)$ nil)...))
$M([t_1 | t_2])$ = (lst (cons $M(t_1)$ (lval $M(t_2)$)))
$M(\ll b_1{:}1, \ldots b_n{:}1 \gg)$ = (bin (bcons #bb_1 ... (bcons #bb_n bnil)...))

Fig. 9. The $M(t)$ function mapping an Erlang term t to its encoding in SMT-LIB.

Representing Erlang terms in SMT-LIB. The most general type in Erlang is `term()`, representing all valid terms. For Mini Core Erlang, as defined in Fig. 5, we consider `term()` to consist of integers, floating-point numbers, atoms, lists, tuples and bitstrings. To simplify presentation, we will consider only *proper* lists, i.e., lists terminating with `[]`. Notice that Erlang (and, in fact, also the syntax in Fig. 5) allows improper lists, such as `[1|2]`. We will also ignore function literals (they are not very practical with a first-order SMT solver, such as Z3) and maps.

In the SMT-LIB notation, we declare an algebraic type `Term` for representing Erlang terms, as shown in Fig. 8. We also declare three auxiliary types `TList`, `IList` and `BitVecList`, representing lists of terms, lists of integers and lists of bits, respectively. `IList` is used in the internal representation of atoms, which are modeled as lists containing the ASCII codes of the characters that form them. `BitVecList` is used in the representation of bitstrings, which goes nicely with the left-to-right approach that Erlang employs when pattern matching or concatenating bitstrings and binaries. Although SMT-LIB natively supports bit-vectors of arbitrary size, they are of fixed size and incompatible with Erlang's bitstring pattern matching.

Integers and floating-point numbers are built-in types in SMT-LIB, and in fact they are similar to numbers in Erlang. However, some of operations on them, like for example floor and ceiling, have different semantics so we had to modify the axioms they produce in order to be sound for Erlang. In the notation above, the algebraic data type `Term` is defined having six constructors: `int`, `real`, `atm`, `lst`, `tpl` and `bin`. Similarly, `TList` is defined with two constructors: `nil` and `cons`. The constructor `cons` takes two parameters, the first being a `Term` and the second a `TList`. The names `hd` and `tl` are used to extract these two parameters from a `cons` element.

For example, the Erlang terms `42`, `[17,42]` and `{42,ok}` can be represented in SMT-LIB as follows:

```
(int 42)
(lst (cons (int 17) (cons (int 42) nil)))
(tpl (cons (int 17) (cons (atm (acons 111 (acons 107 anil))) nil)))
```

The definition of function $M(t)$, mapping an Erlang term t to its encoding in SMT-LIB, is shown in Fig. 9. Notice that only a subset of terms corresponding to first-order values need to be represented in SMT-LIB. This function can be extended with better support for variables; in the current definition, we assume that each Erlang variable corresponds to an SMT variable and that, for simplicity, they have the same name.

Encoding constraints in SMT-LIB. The path predicate of an execution is a conjunction of constraints $C_1, C_2, \ldots, C_n$, witnessed during the execution. The task of the solver is to take the first k of these constraints ($k < n$), add the negation of C_{k+1}, and find a *model* that makes all of them true, i.e., find appropriate values for unbound variables such that every one of these constraints is true.

We therefore need a way to encode constraints in SMT-LIB; for this purpose, we define a function $P(C)$, mapping a constraint C (such as the ones we generated in Sections 3.3 and 3.4) to a set of axioms that can directly be asserted in Z3. (Notice that we interpret sets of constraints as their conjunction; the same is true for SMT axioms.) Although SMT-LIB

Table 1
Encoding of constraints in SMT-LIB notation. The negative axiom $N(C)$ is equivalent to $\neg P(C)$ but may be simplified, e.g., using logic tautologies.

Constraint C	Positive axiom $P(C)$
Structural constraints	
t is a true guard	`(= M(t) M(true))`
$t = t'$	`(= M(t) M(t'))`
t is a non-empty list	`(and (is-lst M(t))` `  (is-cons (lval M(t))))`

t is a tuple of size n	`(and (is-tpl M(t))`
	`  (is-cons (tval M(t)))`
	`  (is-cons (tl (tval M(t))))`
	`  (is-cons (tl (tl (tval M(t)))))`
	`  ... )`

t is a bitstring starting with n bits given by t_1 and followed by t_2	`(and (is-bin M(t))`
	`  (is-int M(t1))`
	`  (is-bin M(t2))`
	`  (= M(t) (bin (bcons b1 ... (bcons bn (bval M(t2)))...)))`
	`  (= b (concat b1 b2 ... bn))`
	`  (= (ival M(t1)) (bv2int b)))`

BIF constraints	

$t = \texttt{hd}(t')$	`(and (is-lst M(t'))`
	`  (is-cons (lval M(t')))`
	`  (= M(t) (hd (lval M(t')))))`

$t = \texttt{tl}(t')$	`(and (is-lst M(t'))`
	`  (is-cons (lval M(t')))`
	`  (= M(t) (lst (tl (lval M(t'))))))`

$t = \texttt{is\_integer}(t')$	`(= M(t) (ite (is-int M(t')) M(true) M(false)))`

$t = t_1 + t_2$	`(and (or (is-int M(t1)) (is-real M(t1)))`
	`  (or (is-int M(t1)) (is-real M(t1)))`
	`  (= M(t) (ite (and (is-int M(t1)) (is-int M(t2)))`
	`  (int (+ (ival M(t1)) (ival M(t2))))`
	`  (ite (and (is-int M(t1)) (is-real M(t2)))`
	`  (real (+ (ival M(t1)) (rval M(t2))))`
	`  (ite (and (is-real M(t1)) (is-int M(t2)))`
	`  (real (+ (rval M(t1)) (ival M(t2))))`
	`  (real (+ (rval M(t1)) (rval M(t2)))))))))`

natively supports negation, instead of taking $N(C) = \texttt{(not } P(C)\texttt{)}$ as the encoding of the negation of C, we choose to define a separate function $N(C)$ for this purpose. The reason for this is that, often, we can generate simpler SMT constraints in this way, e.g., by avoiding double negation. Using these two functions, we simply assert axioms $P(C_1), \ldots, P(C_k), N(C_{k+1})$ and ask the solver for a consistent model.

The definition of functions $P(C)$ and $N(C)$ is pretty straightforward. The former is given in Table 1, following the syntactic structure of the constraints. Here we present only the primary structural constraints generated by pattern matching, and some constraints generated by the use of BIFs (specifically, `hd/1`, `tl/1`, `is_integer/1`, and `'+'/2`).

For example, Eq. (3) on page 115 will be translated as the following set of axioms:

```
(not (and (is-lst l)
          (is-cons (lval l)) ))
(not (and (is-lst l)
          (is-nil (lval l)) ))
```

Here we used the recognizer predicates that SMT-LIB provides once the data types are declared. They allow us to directly query the constructor of a data type. They are created from the name of the constructor prefixed with `"is-"`.

Most of the other constraint paths shown as examples in Section 3.1 do not have simple translations: they all contain occurrences of term comparisons, using Erlang BIFs such as `erlang:'>'/2`, which are emulated using Erlang code. In fact, a constraint coming from a term comparison, such as `X > 42`, would not occur in a constraint path, as the BIF `erlang:'>'/2` is emulated by code, a fragment of which is shown in Fig. 10. As a result, the constraints that must be encoded in SMT-LIB are simpler comparison constraints between terms of the same simple type, e.g., `lt_int` and `lt_float`. Even `lt_list`, which compares two list terms lexicographically, is emulated as a recursive function in Erlang, eventually comparing values of simple types.

Encoding type specifications. As mentioned in Section 4, type specifications impose additional constraints on the input. They basically enforce a structure on a parameter. Simple specifications are encoded using just the recognizer predicates, constructors and accessors. For example, the following specification:

```
-spec f(X :: integer(), Y :: {atom()}) -> ok.
```

```
-spec '>'(term(), term()) -> boolean().
%% arithmetic comparison, including coercions
'>'(X, Y) when is_integer(X), is_integer(Y) -> lt_int(Y, X);
'>'(X, Y) when is_float(X), is_float(Y) -> lt_float(Y, X);
'>'(X, Y) when is_integer(X), is_float(Y) -> lt_float(Y, float(X));
%% ...
%% numbers are smaller than other terms
'>'(X, _Y) when is_number(X) -> false;
'>'(X, Y) when is_atom(X), is_number(Y) -> true;
%% atoms are compared lexicographically
'>'(X, Y) when is_atom(X), is_atom(Y) ->
  lt_list(atom_to_list(Y), atom_to_list(X));
%% atoms are smaller than other terms except numbers
'>'(X, _Y) when is_atom(X) -> false;
%% ...
```

Fig. 10. Fragment of CutEr's code for emulating `erlang:'>'/2`.

is encoded as the following assertion:

```
(and (is-int X)
     (and (is-tpl Y)
          (is-cons (tval Y))
          (is-atm (hd (tval Y)))
          (is-nil (tl (tval Y))) ))
```

More complex specifications, e.g., involving lists, are encoded with the help of SMT recursive functions, introduced in SMT-LIBv2.5 [27]. For example, the predicate `is_integer_list(L)` in Section 4 can be encoded using a recursive function as:

```
(define-fun-rec
  is-integer-list ((t Term)) Bool
  (and (is-lst t)
       (or (is-nil (lval t))
           (and (is-cons (lval t))
                (is-int (hd (lval t)))
                (is-integer-list (lst (tl (lval t))))))))
```

This approach can be generalized to work for recursive user defined types. For each such type, a predicate on terms can be automatically defined. For example, consider the following type definition for binary trees containing integers:

```
-type tree() :: nil | {integer(), tree(), tree()}.
```

The predicate `is-tree` can be defined as:

```
(define-fun-rec
  is-tree ((t Term)) Bool
  (or (= t (atm (icons 110 (icons 105 (icons 108 inil)))))  ; t is nil
      (and (is-tpl t)                                       ; t is tuple
           (is-cons (tval t))
           (is-int (hd (tval t)))              ; 1st element is integer()
           (is-cons (tl (tval t)))
           (is-tree (hd (tl (tval t))))        ; 2nd element is tree()
           (is-cons (tl (tl (tval t))))
           (is-tree (hd (tl (tl (tval t)))))   ; 3rd element is tree()
           (is-nil (tl (tl (tl (tval t))))))))
```

Simplifying complex sets of axioms. If the set of axioms given to Z3 is too complex, the solver may be unable to find a model and reply with `unknown`. In these cases, we try to simplify the set of axioms, aiming to obtain one whose satisfiability can be verified. A useful kind of simplification is obtained by limiting the number of variables in the universe. In the initial query, we consider all variables as unknown, i.e., free. If this query returns `unknown`, we can try again, this time fixing the values of a subset $F \subset S$ of the variables to their respective concrete values. Notice that although this procedure may help us, in the case that our set of constraints turns out to be satisfiable, it does not help us if no solution is found even if we could try all possible subsets $F \subset S$.

5.4. Current limitations

The current version of CutEr does not support maps, a data type that was introduced only quite recently to Erlang. Also, in its current version, CutEr does not emulate all of Erlang's BIFs in such a way to be able to reason about their results in symbolic form. We do not expect any serious difficulties in supporting maps, or in emulating more BIFs in a future version.

Also, even though CutEr currently supports higher-order functions, it cannot fully reason about functional terms. For example, if the entry function takes a functional parameter and the initial execution instantiates this with function `f`, CutEr will treat this as a concrete value; it will not try to generate other functions that, when given as arguments in the place of `f`, will drive execution to an unhandled exception. Full support of higher-order functions is currently our primary ongoing work. We believe that reasoning about higher-order functions is possible in CutEr using uninterpreted functions in Z3 (or any other off-the-shelf first-order SMT solver), with some limitations.

On a related note, the effectiveness of concolic testing tools in general is sensitive to the capabilities of their constraint solver(s). In this respect, it would be nice to allow solvers other than Z3 to be used as plugins.

6. Implementation aspects

Let us now discuss two implementation issues that are important to pay attention to in functional programs for their efficient concolic execution, namely: (1) avoiding repetition in the generated constraints by performing pattern matching compilation (Section 6.1) and (2) transforming the definitions of recursive types into a form that is simpler for SMT solvers to handle (Section 6.2).

6.1. Taking advantage of pattern matching compilation

By using the Core Erlang AST for executing a program, CutEr does not profit from various optimizations typically applied at the intermediate code level. However, one particular optimization, namely *pattern matching compilation* was found to be particularly important and worth the effort of implementing at the level of Core Erlang code.[2] This optimization essentially compiles a series of pattern matching attempts to a decision tree, in order to find the matching clause with as few checks as possible and thus speed up program execution. The compilation of pattern matching to compact decision trees is very common in compilers of functional languages [28–31].

We did not consider this optimization only to improve the performance of concrete or symbolic evaluation. Instead, we observed that CutEr was spending a lot of time trying to solve constraints that were unsatisfiable. The problem, as we will show, was caused because the unoptimized control flow graph of functions contained duplicate conditions, which (all but the first occurrence) could not be reversed.

Most constraints are generated by `case` expressions, where a sequence of expressions is evaluated and then matched against a list of clauses. In each clause, the success of the match against the value of the sequence of expressions can be expanded into the conjunction of separate checks which involve the patterns and the guard expression of the clause. Each clause is attempted from left to right and against the root of the sequence's value. Let us see how this performs in the simple example of function `or/2`, implementing Boolean disjunction, and written here in Erlang (left) and transformed to Core Erlang (right) code:

```
or(false, true)  -> true;
or(true, true)   -> true;
or(false, false) -> false;
or(true, false)  -> true.
```

$\Longrightarrow$

```
or/2 = fun (_cor1, _cor0) ->
  case <_cor1,_cor0> of
    <false,true>  when true -> true
    <true,true>   when true -> true
    <false,false> when true -> false
    <true,false>  when true -> true
    <_cor3,_cor2> when true -> FAIL
  end
```

Assume there is a call to `or(true, false)` and let us call `X` and `Y` the two parameters `_cor1` and `_cor0` (which are here bound to `true` and `false`, respectively). The clause that will match this call will be the fourth one but let us see the order in which the checks are performed. The first clause will fail when `X` fails to match `false`. In the second clause, `X` will match `true` but subsequently `Y` will not match `true`, thus causing the clause to fail. The third clause will also fail immediately, when `X` fails to match `false`. We can see however that this check is redundant, as it has already been performed in the first clause. The first check in the fourth clause will be to match `X` to `true`. This will succeed but, again, this check is also redundant as it has already been performed in the second clause. Finally, `Y` will match to `false` and the clause will be selected.

Even in this simple example, six checks were required and two of them were unnecessarily repeated. In functions with more clauses, this repetition may have adverse effects on CutEr's performance; we will report performance results from such a case in Section 7.2. The path predicate that corresponds to the example above is:

$$\mathtt{X} \neq \mathtt{false} \wedge \mathtt{X} = \mathtt{true} \wedge \mathtt{Y} \neq \mathtt{true} \wedge \mathtt{X} \neq \mathtt{false} \wedge \mathtt{X} = \mathtt{true} \wedge \mathtt{Y} = \mathtt{false}$$

[2] The Erlang bytecode compiler does perform pattern matching compilation; however not at the Core Erlang representation, but at a later stage.

At some point, CutEr will try to reverse the fourth constraint, thus asking the solver to try to satisfy the formula:

$$\texttt{X} \neq \texttt{false} \wedge \texttt{X} = \texttt{true} \wedge \texttt{Y} \neq \texttt{true} \wedge \texttt{X} = \texttt{false}$$

This is inherently unsatisfiable due to the contradiction $\texttt{X} \neq \texttt{false} \wedge \texttt{X} = \texttt{false}$. The same will happen if it tries to reverse the fifth constraint. CutEr is therefore spending time to satisfy formulae that are bound to be unsatisfiable.

By applying pattern matching compilation to the example above, function `or/2` is essentially rewritten in the following forms, Erlang source (left) and Core Erlang (right), which implement a decision tree:

```
or(X, Y) ->
  case X of
    false ->
      case Y of
        true -> true;
        false -> false
      end;
    true ->
      case Y of
        true -> true;
        false -> true
      end
  end.
```

$\Longrightarrow$

```
or/2 = fun (_cor1, _cor0) ->
  case <_cor1> of
    <false> when true ->
      case <_cor0> of
        <true>  when true -> true
        <false> when true -> false
        <_cor2> when true -> FAIL
      end
    <true> when true ->
      case <_cor0> of
        <true>  when true -> true
        <false> when true -> true
        <_cor3> when true -> FAIL
      end
    <_cor4> when true -> FAIL
  end
```

Using this method, just four checks are necessary for our previous example. The resulting path constraint is the following, where each constraint can be reversed without contradicting any of the previous ones:

$$\texttt{X} \neq \texttt{false} \wedge \texttt{X} = \texttt{true} \wedge \texttt{Y} \neq \texttt{true} \wedge \texttt{Y} = \texttt{false}$$

To take advantage of pattern matching compilation in CutEr, we updated an outdated implementation of pattern matching compilation at the level of Core Erlang that existed already in Erlang/OTP, extended it to also handle binaries and bitstrings, and used it to transform the Core Erlang AST before concolic execution begins. The impact of this optimization on CutEr's performance is significant and the number of calls to the solver (especially the number of calls that involved unsatisfiable constraints) is considerably reduced, as we will see in Section 7.2 with a module of significant size.

Note however that pattern matching compilation also interacts with the depth limit that CutEr uses. For example, for the original `or/2` function, a depth limit value of one suffices to explore all its clauses. Instead, for the version of this function where pattern matching compilation has been applied, we need to use a depth limit of two to visit all its clauses. This is because the notion of depth is based on the number of Core Erlang `case` expressions that concolic execution encounters on each path. The example we present in Section 7.2 also illustrates this point.

6.2. Better encoding of type definitions

At the end of Section 5.3, we saw how type specifications are encoded in an SMT solver's logic. Recursive types, such as lists, are encoded as SMT recursive functions that essentially correspond to globally quantified formulas, as shown in the SMT-LIB v2.5 standard [27].

Modern SMT solvers use incomplete instantiation-based methods to reason about such formulas [32]. However, this approach does not work well in practice when the goal is to show satisfiability and to produce models, as opposed to proving unsatisfiability [33]. Given how often recursive formulas occur in type predicates, this proves to be a significant issue for CutEr. Although research in this area is very active and has displayed promising results [34,33], we have found it necessary to modify the definition of type predicates to facilitate the solver.

To give an example, consider the following type definitions:

```
-type l() :: [ { l(), t() } | integer() ].
-type t() :: { [ t() ], [ t() ] }.
```

These define two recursive predicates `is-l` and `is-t`. However, their definitions are relatively complicated, as they involve several levels of nesting of list and tuple types. When we query the solver to find an element of `t()` with the additional constraint that the tuple's elements must not be empty lists, the solver will fail to find a model and respond with `unknown`.

To facilitate the solver's job, the trick that CutEr uses is to keep the bodies of all recursive functions as simple as possible. This transformation allows Z3 `v4.5.0`, which is the SMT solver we are currently using, to find a satisfiable model if one exists. This is achieved by breaking up complex type definitions into smaller but simpler ones. In the above example, types `l()` and `t()` are transparently transformed to:

```
-type l()  :: [ l1() ].
-type l1() :: l2() | integer().
-type l2() :: { l(), t() }.
-type t()  :: { t1(), t1() }.
-type t1() :: [ t() ].
```

Notice that each type definition contains just one nesting level of list and tuple types; the generated recursive functions now have much simpler bodies. Notice also that type `t1()` is used in the definition of `t()` as a common type subexpression. In practice, this simplification allows the solver to prove satisfiability and produce models in all the cases of recursive types that we have encountered so far.

7. Some experiences

In this section we describe some of our current experiences with concolic testing of Erlang programs.

7.1. Variations of the running example

Let us, once more, re-examine our running example. Recall that once its code gets extended with a type specification that constraints the input to the `foo/1` function to be a list of integers, the only error remaining is with `L = [42]`. This error is due to the `case` statement of the `fcmp/1` function not handling the atom `eq`, which is one of the possible return values of `cmp/1`. One could argue, of course, that this error is something that even some more lightweight method, e.g., a *pattern matching non-exhaustiveness* analysis, would also be able to discover given sufficient type information. More generally, the reader may be wondering whether most (if not all) errors that CutEr discovers are pattern matching failures that are as simple as that.

Our experience is that this is not the case. We chose this rather simplistic example in order to ease the exposition of the constraints that are generated in its concolic execution and keep the presentation of the techniques that CutEr uses simple. In a pragmatic programming language such as Erlang, a similar error can easily remain hidden under an arbitrary number of function calls, perhaps type-converting ones, that can manage to confuse even the strongest of analyses. Below, we show a variant of the `fcmp/1` function where its `case` statement has been turned into an assertion using pattern matching. This version is based on the representation of atoms as strings, i.e., lists of characters, and the fact that both `lt` and `gt` end with a "t" character, whereas `eq` does not:

```
fcmp(X) ->     % 116 is ASCII for 't'
  116 = hd(tl(atom_to_list(cmp(X)))),
  ok.
```

This version may manage to confuse some readers, but it does not succeed in deceiving CutEr! The tool easily manages to report that the program crashes with `L = [42]`.

Another somewhat surprising experience is that sometimes the failing test cases that CutEr generates are not so "expected". For example, consider the following function, which calls the `fcmp/1` function, either the one in the running example or the one above:

```
-spec bar([integer()]) -> ok.
bar(L) when length(L) < 4 -> ok;
bar(L) -> fcmp(lists:sum(L)).
```

The only purpose of this made-up example is to test the capabilities of the tool by forcing the generation of a list with at least four elements. CutEr, starting from the call `bar([])`, quickly discovers that concolic execution of this function will fail for the input list `L = [1323,1888,-12894,9725]`, where the sum of the four integers is equal to 42. This shows both the power of the tool and its dependence on its SMT solver component, which works not only as a black box but as black magic in this case. This, however, is a known problem in the symbolic testing world and efforts are made to provide smaller/minimal counter-examples to the users.

7.2. Experiences from a bigger example

Prompted by a bug report on the `erlang-bugs` mailing list, which reported a crash in the Erlang compiler,[3] we have used CutEr both to automatically locate the input that caused that particular error and to verify that no other inputs leading to crashes exist for that compiler component. The culprit code is a showcase for the capabilities of a concolic testing tool. The bug involves only a single module of Erlang/OTP's standard library, called `otp_internal`, whose source in the pre-release of Erlang/OTP 18.0[4] consists of 647 lines of code and has only one exported function, called `obsolete/3`. The

[3] http://erlang.org/pipermail/erlang-bugs/2015-May/004944.html.

[4] The buggy version of this module in the pre-release of Erlang/OTP 18.0 is accessible at https://github.com/erlang/otp/blob/OTP-18.0-rc1/lib/stdlib/src/otp_internal.erl.

three arguments of this exported function are: a module name (atom), a function name (atom), and a byte-sized integer denoting the arity of the function. Running CutEr on this module is simple; all it takes is to issue the following command at the Unix prompt:

```
$ cuter otp_internal obsolete '[lists, foreach, 2]'
```

where `cuter` is the name of the CutEr script, and its arguments are the name of the module containing the program unit to test, the function which is the unit's entry point, and a list of (arbitrary in this case) arguments that this function takes. Running this command causes the following output:

```
Compiling otp_internal.erl ... OK
Testing otp_internal:obsolete/3 ...
.........................x....x......x........xx........xx.......
    <more dots and x's here deleted>
...........xxxxxxxxxxxxxxxxxxx.............xxxxxxxxxxxxxxxxxx.....
otp_internal:obsolete(ssl, negotiated_next_protocol, 1)
...xxxxxxxxxxxxxxxxxxx.......xxxxxxxxxxxxxxxxxx.......xxxxxxxxx
    <more dots and x's here deleted>
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
=== INPUTS THAT LEAD TO RUNTIME ERRORS ===
#1 otp_internal:obsolete(ssl, negotiated_next_protocol, 1)
```

where each dot denotes that the constraint solver component produced a test input that explores a new path, and each `x` denotes that the constraints supplied to the constraint solver were unsatisfiable. As we see, concolic execution discovers one set of inputs that cause the execution of the `otp_internal:obsolete/3` function to crash (this is the input that triggered the reported bug) and also finds that there are no other inputs that result in crashes (unrelated to the one found) for this program unit.

The above command finished in about eight minutes on a commodity PC with an Intel i7 processor. Since the processor has multiple (four physical, eight logical) cores, the user can specify that CutEr uses multiple pollers (`-p`) and multiple SMT solvers (`-s`). Indeed, one can run the command:

```
$ cuter otp_internal obsolete '[lists, foreach, 2]' -p 4 -s 4
```

which terminated with the same result in about two minutes.

Both of the above commands ran with the default depth limit (currently 25) of the tool. As it turns out, this particular bug is quite shallow and can be discovered even with a depth limit of one without pattern matching compilation and with a depth limit of three (for the three `case` statements required for the three arguments of the entry function) with the pattern matching compilation described in Section 6.1, which is enabled by default. Running the command:

```
$ cuter otp_internal obsolete '[lists, foreach, 2]' -p 4 -s 4 -d 3
```

still discovered the erroneous input and finished concolic execution in only 28 seconds.

Even though the required depth limit to discover the bug is even lower without pattern matching compilation, the effects of disabling this optimization are detrimental for the concolic execution of this program unit. With a smaller depth limit and pattern matching disabled (i.e., with `-d 1 -disable-pmatch` as the only options) concolic execution requires 218 minutes (i.e., more than three and a half hours) to terminate in single-threaded mode; with many pollers and solvers running in parallel (i.e., adding options `-p 4 -s 4`), this time is reduced to 55 minutes. Compared with 28 seconds, the time difference is huge. Pattern matching compilation is so effective that the option to disable it is nowadays available in CutEr only for debugging and profiling reasons.

The above crash in the pre-release of the 18.0 Erlang compiler was quickly fixed after it was reported. Slightly more than a year later, release 19 of Erlang/OTP came out and some more functions from the Erlang standard libraries had now been removed or superseded by other ones, which caused changes in the `otp_internal` module. We tested the 19.2 version of this module with CutEr in March 2017 and discovered two more calls to the `obsolete/3` function that would make the compiler crash:

```
otp_internal:obsolete(rpc, safe_multi_server_call, 2)
otp_internal:obsolete(rpc, safe_multi_server_call, 3)
```

Rather than reporting these as bugs, we decided to directly submit a patch that fixes them.[5] Soon afterwards, the pull request was merged into the main repository and the fix was included in the next minor Erlang/OTP release (19.3).

7.3. Limits of type specifications

As mentioned in Section 4, the language of type specifications in Erlang is not expressive enough to capture the actual constraints of all types. This is currently something to address for the fully automatic use of the tool in all situations. For example, the `calendar` module of the standard library defines essentially the following data type:

[5] The relevant pull request is at https://github.com/erlang/otp/pull/1371.

```
-type date() :: {Year::non_neg_integer(), 1..12, 1..31}.
```

which is then used in various functions of this module. CutEr reports that these functions will fail when supplied with `{42,4,31}`, i.e., 31st of April, as input.

Similar issues exist in some library functions that have hidden preconditions. For example, Erlang defines an `orddict` data type, which essentially is a key-value dictionary where a list of pairs is used to store the keys and values and the list is ordered after the keys. Its type declaration reads:

```
-type orddict(Key, Val) :: [{Key::term(), Val::term()}].
```

Most functions of this module work with dictionaries that contain arbitrary terms as keys and values. Some others like `orddict:append/3` have extra constraints. This function's type specification is:

```
-spec append(Key, Value, orddict(Key, Value)) -> orddict(Key, Value) when Key::term(), Value::term().
```

but its manual page reads:

> This function appends a new `Value` to the current list of values associated with `Key`. An exception is generated if the initial value associated with Key is not a list of values.

CutEr, ignoring the manual page, quickly finds that the function will fail for the call:

```
orddict:append(0, 1, [{0,17}, {3,[12]}, {7,29}]).
```

However, note that the function does *not* insist that all values are lists; for example, the following call returns successfully:

```
orddict:append(3, 1, [{0,17}, {3,[12]}, {7,29}]).
```

because the key (`3`) that the operation involves is associated with a list value (`[12]`) in this dictionary.

In cases such as these, where functions which are used as test units have preconditions that are not reflected in their declared types, CutEr may come up with a number of failing inputs that are uninteresting; there may be no easy way of distinguishing them from test unit failures that are essential to correct. Admittedly, if the test unit is a whole application instead of an individual function, the input of the entry point rarely is a complex data structure with invariants such as those of `orddict:append/3`. Still, it is common practice to test small and less complex parts of an application individually. Therefore, it is essential to have a mechanism for silencing any irrelevant reports for successfully testing those units. One way to approach this problem is to use a slightly different test unit that calls the problematic component only after validating its input and ensuring it complies to all of its preconditions.

As a first example, assume that we want to test the `orddict:append/3` function. We can create a new wrapper function, say `safe_append/3`, that first verifies that the arguments have the proper types and dependencies among them and then calls `orddict:append/3` with those arguments. Roughly:

```
safe_append(Key, Val, Orddict) ->
  case is_valid_call_for_append(Key, Val, Orddict) of
    true  -> ordict:append(Key, Val, Orddict);
    false -> skip
  end.
```

Here, `is_valid_call_for_append/3` is a function that performs all validations and returns `true` only if it is OK to proceed with the actual call. If not, then the execution simply ends at this point.

The advantage of writing such a wrapper function per test unit is that no irrelevant failures are reported. The downside is that more work is required by the programmer, because validating the input can occasionally be as complex as the actual program.

In many cases though, such Boolean functions that check dependencies between data types defined and used in a module are already provided by the module. As a concrete example, consider that we want to test the `calendar:day_of_the_week/3` function. We can use the `valid_date/3` function that the `calendar` module provides and consider as our test unit the following function instead:

```
-spec test_day_of_the_week(non_neg_integer(), 1..12, 1..31) -> 1..7.
test_day_of_the_week(Y, M, D) ->
  case calendar:valid_date(Y, M, D) of
    true  -> calendar:day_of_the_week(Y, M, D);
    false -> skip
   end.
```

On the other hand, one could also argue that programs need to self-protect against malformed input by catching any exceptions that may occur in these cases; i.e., that program units should be written more defensively. We do not disagree with this. But we want the use of concolic testing and of CutEr to be equally usable independently of whether the functions to be tested have been written defensively or not. In the cases that they have not been written defensively, unit wrappers like the above need to be written by the users.

Table 2
Clause coverage measurements of some modules from the Erlang/OTP 19.1 standard library.

Module name	LOC	Without pattern matching compilation	With pattern matching compilation	Excluding compiler generated clauses
`lists`	2 840	105 of 143 (73.43%)	155 of 203 (76.35%)	97 of 98 (98.98%)
`orddict`	234	38 of 47 (80.85%)	59 of 78 (75.64%)	36 of 36 (100.00%)
`ordsets`	262	51 of 65 (78.46%)	72 of 93 (77.42%)	51 of 52 (98.08%)
`calendar`	552	85 of 108 (78.70%)	92 of 121 (76.03%)	77 of 79 (97.47%)
`erl_internal`	586	286 of 296 (96.62%)	741 of 751 (98.67%)	283 of 283 (100.00%)
`otp_internal`	626	273 of 277 (98.56%)	718 of 730 (98.36%)	272 of 274 (99.27%)
`string`	542	135 of 191 (70.68%)	172 of 240 (71.67%)	131 of 135 (97.04%)

7.4. Isolating and fine-tuning the test unit

It is common practice in unit testing and in concolic testing to isolate the test unit from other components and test it independently, using function stubs, mock modules and other similar techniques. This also applies to functions that are called solely for their side effects and are irrelevant to the business logic of the actual program. For example, logging functions commonly fall into this category.

In such cases, it is useful to be able to limit concolic execution in such a way that irrelevant parts of the unit that is being tested are only concretely executed. This helps reduce the number of paths that need to be explored, allowing CutEr to explore the interesting ones faster. Using logging as an example, by disabling concolic execution for the logging function we can practically save CutEr from trying to find bugs in the logging library code every time we test any other functions that happen to use logging.

To this end, CutEr allows the user to define modules and functions that are meant to be executed only concretely. It is important, however, not to abuse this feature as it may end up hindering rather than improving the effectiveness of the tool.

7.5. Rough measurements of coverage

As we mentioned at the end of Section 5.2, we can use the labels that are already placed on the decision nodes of the AST to calculate the achieved clause coverage. We have run CutEr on a few selected modules of the Erlang standard library and measured the clause coverage which was achieved; i.e., the number of Core Erlang `case` clauses whose *body* (not just their pattern and guard expression) was visited. The results are shown in Table 2.

Four of these modules (`lists`, `orddict`, `ordset`, and `string`) were selected because they are frequently used by applications. From the remaining three, two of them (`calendar` and `otp_internal`) were selected because we have also used them before in this section, and the last one (`erl_internal`) due to being important for the Erlang/OTP system and relatively self-contained.

For each module, we tested all its exported functions and took into consideration their type specifications. The number of lines of code of these modules is shown in the second column of the table. However, note that four of these modules (`orddict`, `ordset`, `string`, and `erl_internal`) call functions of the `lists` module and all of them also call built-in functions defined in the `erlang` module (which are transformed by CutEr to calls to functions that emulate these built-ins), so the size of the code that this experiment involves is considerably larger. The tests ran on a machine with four AMD Opteron(TM) Processor 6276 CPUs (2.30 GHz) which gives a total of 64 cores, and 128 GB of RAM running Debian Linux 4.8.0-1-amd64 SMP x86_64. The installed Erlang version was Erlang/OTP 19.1. Since the search depth largely affects the results, each test was executed with the maximum possible depth value that allowed CutEr to finish within two minutes with 64 SMT solver (`-s 64`) and 32 poller (`-p 32`) processes.

Each function was tested under two configurations. The first one was without pattern matching compilation; in the second one, pattern matching compilation was enabled. Note that this increases the number of `case` clauses; often more than double.

Recall that the Erlang compiler introduces catch-all clauses to all `case` expressions that do not already have one; cf. Sections 2.2 and 3. In general, the introduction of these catch-all clauses skews clause coverage measurements: most of their bodies are not expected to be examined unless of course there is some input that is allowed by the function's specification but not handled by its regular clauses. However, since these clauses are annotated as compiler generated, it is relatively easy to exclude them from the measurements and calculate the clause coverage without them.

In Table 2, we show the coverage without and with pattern matching compilation in its third and fourth columns respectively. The last column shows the clause coverage of the AST without pattern matching compilation when the compiler generated clauses are not taken into account. Basically, it represents the coverage of the actual code that the source code of these modules contains.

CutEr achieves similar coverage both with and without pattern matching compilation in all modules. For some modules, we notice that the tool fails to cover a significant amount of the compiler generated clauses. If we exclude these, then CutEr covers almost 99% of the case clauses in six out of seven modules. This is an indication that those clauses are essentially unreachable.

8. Concluding remarks and future work

We have presented an approach to apply concolic testing to functional programs at the level of their core language. Moreover, we have presented the architecture and implementation technology of CutEr, a concolic testing tool for the functional subset of Erlang that implements this approach. We are not aware of any other attempt to apply concolic testing at this language level or any such similar tool, not only for Erlang but for functional programming languages in general.

Still, our work is only a (very important) first step in this effort. Besides working on lifting the limitations described in Section 5.4, a concolic testing tool also requires a variety of strategies to intelligently explore and prune the search space and significant engineering effort to become scalable and effective. There is still quite a lot that can be done in this front, especially regarding more focused search and experimentation with a variety of SMT solvers.

Furthermore, we would like to allow the programmer to define the invariants of the data structures they are using which cannot be expressed with the current type annotations. This addition will enhance CutEr's effectiveness by addressing the issue mentioned in Section 7.3. Last but not least, for a language like Erlang, an effective concolic testing tool also needs to handle the concurrency part of the language, not only its functional part. CutEr actually already handles some of Erlang's concurrency constructs, such as `receive`, but the description of the techniques it employs and its implementation is left for another paper.

Acknowledgement

The authors would like to thank the anonymous reviewers for comments and suggestions that have improved the presentation of our work.

References

[1] K. Claessen, J. Hughes, QuickCheck: a lightweight tool for random testing of Haskell programs, in: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ACM, New York, NY, USA, 2000, pp. 268–279.
[2] C. Runciman, M. Naylor, F. Lindblad, SmallCheck and Lazy Smallcheck: automatic exhaustive testing for small values, in: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, ACM, New York, NY, USA, 2008, pp. 37–48.
[3] Erlang QuickCheck, http://www.quviq.com/products/erlang-quickcheck/.
[4] M. Papadakis, K. Sagonas, A PropEr integration of types and function specifications with property-based testing, in: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, ACM, New York, NY, USA, 2011, pp. 39–50.
[5] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2005, pp. 213–223.
[6] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, New York, NY, USA, 2005, pp. 263–272.
[7] C. Cadar, D. Dunbar, D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, 2008, pp. 209–224, http://dl.acm.org/citation.cfm?id=1855741.1855756.
[8] C.S. Păsăreanu, N. Rungta, W. Visser, Symbolic execution with mixed concrete-symbolic solving, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, 2011, pp. 34–44.
[9] P. Godefroid, M.Y. Levin, D. Molnar, SAGE: whitebox fuzzing for security testing, Commun. ACM 55 (3) (2012) 40–44, http://dx.doi.org/10.1145/2093548.2093564.
[10] A. Giantsios, N. Papaspyrou, K. Sagonas, Concolic testing of functional languages, in: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, ACM, New York, NY, USA, 2015, pp. 137–148.
[11] A. Palacios, G. Vidal, Concolic execution in functional programming by program instrumentation, in: Proceedings of the 25th International Symposium on Logic-Based Program Synthesis and Transformation, in: LNCS, vol. 9527, Springer, 2015, pp. 277–292.
[12] G. Vidal, Concolic execution and test case generation in Prolog, in: Proceedings of the 24th International Symposium on Logic-Based Program Synthesis and Transformation, in: LNCS, vol. 8981, Springer, 2015, pp. 167–181.
[13] F. Mesnard, É. Payet, G. Vidal, Concolic testing in logic programming, Theory Pract. Log. Program. 15 (4–5) (2015) 711–725, http://dx.doi.org/10.1017/S1471068415000332.
[14] E. Larson, T. Austin, High coverage detection of input-related security faults, in: Proceedings of the 12th USENIX Security Symposium, USENIX Association, 2003, http://dl.acm.org/citation.cfm?id=1251353.1251362.
[15] J. Armstrong, Erlang, Commun. ACM 53 (9) (2010) 68–75, http://dx.doi.org/10.1145/1810891.1810910.
[16] R. Carlsson, M. Rémond, EUnit: a lightweight unit testing framework for Erlang, in: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, ACM, New York, NY, USA, 2006, p. 1.
[17] R. Carlsson, T. Lindgren, B. Gustavsson, S.-O. Nyström, R. Virding, E. Johansson, M. Pettersson, Core Erlang 1.0.3, language specification, Tech. rep., Uppsala University, Nov. 2004, https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.
[18] S. Anand, E.K. Burke, T.Y. Chen, J. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, P. Mcminn, An orchestrated survey of methodologies for automated software test case generation, J. Syst. Softw. 86 (8) (2013) 1978–2001, http://dx.doi.org/10.1016/j.jss.2013.02.061.
[19] T. Chen, X.-S. Zhang, S.-Z. Guo, H.-Y. Li, Y. Wu, State of the art: dynamic symbolic execution for automated test generation, Future Gener. Comput. Syst. 29 (7) (2013) 1758–1773, http://dx.doi.org/10.1016/j.future.2012.02.006.
[20] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA, 2008, pp. 443–446.
[21] S. Park, B.M.M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, Q. Xie, CarFast: achieving higher statement coverage faster, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, 2012, 35, pp. 1–11.
[22] N. Papaspyrou, K. Sagonas, On preserving term sharing in the Erlang virtual machine, in: Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop, ACM, New York, NY, USA, 2012, pp. 11–20.
[23] T. Lindahl, K. Sagonas, Practical type inference based on success typings, in: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, ACM Press, New York, NY, USA, 2006, pp. 167–178.

[24] T. Lindahl, K. Sagonas, TypEr: a type annotator of Erlang code, in: Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, ACM, New York, NY, USA, 2005, pp. 17–25.
[25] L. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.
[26] Z3 SMTv2 Guide, http://rise4fun.com/z3/tutorial/guide.
[27] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.5, Tech. rep. Department of Computer Science, The University of Iowa, 2015, http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf.
[28] A. Laville, Comparison of priority rules in pattern matching and term rewriting, J. Symb. Comput. 11 (4) (1991) 321–347, http://dx.doi.org/10.1016/S0747-7171(08)80109-5.
[29] F. Le Fessant, L. Maranget, Optimizing pattern matching, in: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming, ACM, New York, NY, USA, 2001, pp. 26–37.
[30] L. Maranget, Compiling pattern matching to good decision trees, in: Proceedings of the ACM SIGPLAN Workshop on ML, ACM, New York, NY, USA, 2008, pp. 35–46.
[31] P. Gustafsson, K. Sagonas, Efficient manipulation of binary data using pattern matching, J. Funct. Program. 16 (1) (2006) 35–74, http://dx.doi.org/10.1017/S0956796805005745.
[32] L. de Moura, N. Bjørner, Efficient e-matching for SMT solvers, in: Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction, in: LNCS, vol. 4603, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 183–198.
[33] A. Reynolds, J.C. Blanchette, S. Cruanes, C. Tinelli, Model finding for recursive functions in SMT, in: Proceedings of the 8th International Joint Conference on Automated Reasoning, in: LNCS, vol. 9706, Springer-Verlag, New York, NY, USA, 2016, pp. 133–151.
[34] Y. Ge, L. de Moura, Complete instantiation for quantified formulas in satisfiabiliby modulo theories, in: Proceedings of the 21st International Conference on Computer Aided Verification, in: LNCS, vol. 5643, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 306–320.