Concolic Testing for Functional Languages

Aggelos Giantsios[1] Nikolaos Papaspyrou[1] Konstantinos Sagonas[1,2]

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

Abstract

Concolic testing is a software testing technique combining concrete execution of a program (given specific input, along specific paths) with symbolic execution (generating new test inputs that give better path coverage than random test case generation). Concolic testing has so far 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 paper, we demonstrate its application to a functional programming language core, a subset of the core language of Erlang, that supports pattern matching, structured recursive data types such as lists, recursion and higher-order functions. Moreover, we present CutEr, a tool implementing this testing technique. We describe CutEr's architecture, the challenges that need to be addressed by such a tool, 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. In particular, 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 compared to manually written tests.

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 random testing of properties of programs, also known as *property-based testing*. This form of testing is available e.g., in Haskell in the form of the QuickCheck [6] and SmallCheck [17] libraries, and in Erlang by the QuviQ QuickCheck [16] and PropEr [13] 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 (often non-trivial) generators for checking these properties.

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 a well-formed random 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 constraint 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 all/most 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 [8] and CUTE [18] for C, and Symbolic Java PathFinder [15] and jCUTE for Java, to name a few. Concolic testing tools for Java typically work at the level of JVM bytecode, KLEE [3] uses LLVM code, while many tools for C work at the level of assembly. In short, concolic testing has thus far been explored in a language and implementation level which is quite low.

This paper proposes and demonstrates 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, it needs to take pattern matching and the presence of structured data types such as lists and tuples into account, not deal only with integers and bit-vectors. In addition, it needs to deal effectively with higher-order functions, recursion, and 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 be able to generate new inputs of arbitrary terms, not necessarily terms of some particular type. (Of course, the tool has to be able to take type information into account when such information is available.) To the best of our knowledge, this is the first significant effort to apply concolic execution in the context of a functional language. (But we note that an approach to define a framework for concolic execution of Prolog programs [19] also exists.)

The rest of the paper describes how all these are done in the context of a mini functional language core (Sections 3 and 4), and presents CutEr, a concolic unit testing tool for Erlang that we have developed (Section 5). Some preliminary experiences from using CutEr are briefly presented in Section 6. To make the paper 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.

2.1 Concolic Testing

Concolic testing [8, 10, 18] (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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPDP'15, July 14–16, 2015, Siena, Italy.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3516-4/15/07. . . $15.00.
http://dx.doi.org/10.1145/2790449.2790519

a given program is executed both concretely and symbolically in order to achieve high path coverage. In concolic testing, test inputs are generated from the execution of the actual program instead of its model. The main idea behind this 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 the first or one of the remaining 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 the first or one of the remaining branches at the branching statement in question. A *path constraint* is a conjunction of symbolic constraints that describes 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 concrete random 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 that it makes 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 [8, 9, 15, 18].

2.2 Erlang

Erlang [1] 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 the module and the function symbol, also have their 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 and a list.

```
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 [4], Erlang's *unit testing* framework.

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 below. 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.

```
-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.
```

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 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.

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 between 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 using our running example, the way in which concolic testing can be applied to Erlang programs. Our notion of bug finding is to locate inputs that, if given to a program, will force execution to 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.

We assume that each test unit is defined by a function that acts as the entry point for its execution. The parameters of this function are considered the input of the test unit. We also assume that some starting values of these parameters are provided by the user; 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]`.

3.1 Concolic Execution of the Running Example

As mentioned, most concolic tools employ tracing and emulation on a low level representation of the program's code. 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 and a suitable representation to be used by a concolic tool would have to retain type information of (more or less) the same high level.

However, Erlang source code is not suitable as such a representation either, as it is more expressive than necessary. Fortunately, there exists a suitable intermediate representation, Core Erlang [5], that is used internally by the Erlang compiler as a middleware 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.

The running example translates to the Core Erlang code shown in Figure 1. (We took the liberty of slightly simplifying this code, omitting details that were irrelevant to the purpose of this paper 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

```
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 (_cor0, 1) -> ok
      <_cor3,_cor2> when true                          -> FAIL
    end
  ...
```

Figure 1. Simplified Core Erlang code for the running example.

exception will occur, which is shown here with the shorthand `FAIL` instead of an `erlang:error(badmatch)` call. As we have already explained, this unhandled exception corresponds to 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`.

Figure 2 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. 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). The usefulness of the labels will become apparent shortly.

In Figure 3 we can see a trace of the initial execution `example:foo/1([17])`, serializing a path following the control-flow graphs of Figure 2. 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. The left side of Figure 3 shows both concrete and symbolic bindings. 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.)

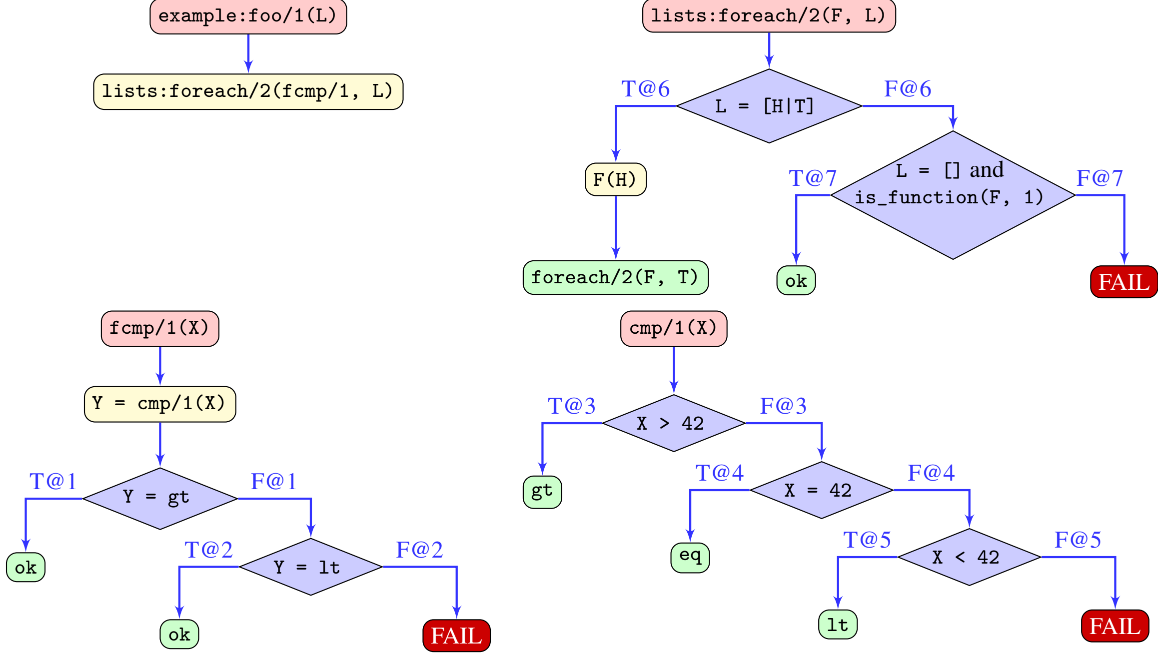

Figure 2. Control flow graphs for all the functions of the example.

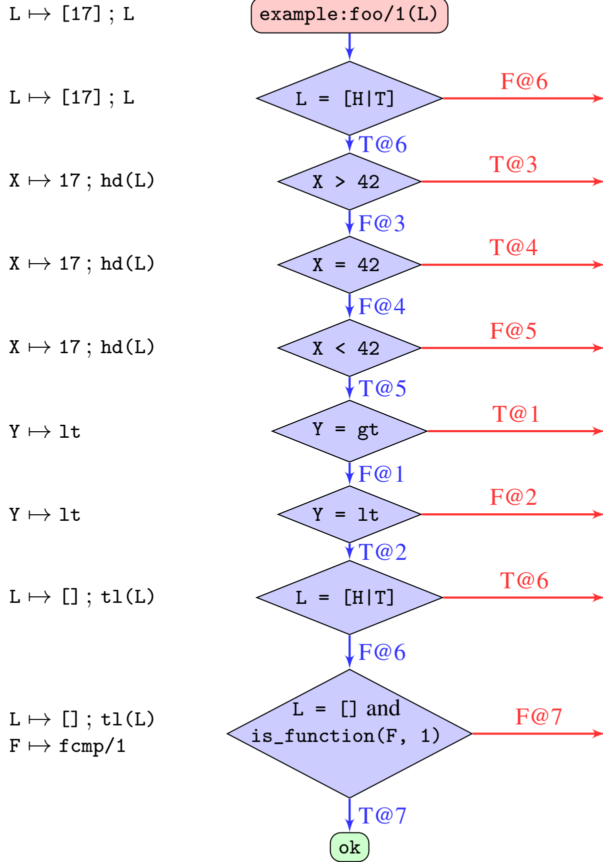

Figure 3. Initial execution of `example:foo([17])`.

The initial execution trace results in the value `ok` being returned and no bug has been found. Concolic execution proceeds by considering each decision node in this initial trace and by trying to *reverse* it, thus exploring more execution paths.

The *order* in which alternatives are attempted is crucial for the success of the concolic testing approach. Several heuristics have been proposed in literature. The heuristic that we use here is based on *path coverage*. To explain it simply:

- 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).
- If no decision node exists satisfying criterion (a), we take into account only criterion (b).
- We choose to stop either when all possible execution paths have been covered (i.e., all labels in the control-flow graphs have been visited), or when a certain depth in the search tree of possible alternatives has been reached.

In fact, in our current implementation we use a slightly different notion of *depth*, which only counts `case` constructs existing at the Core-Erlang source. In this way, all constraints related to the patterns and guards of a specific `case` construct are considered to be at the same level.

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:

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

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

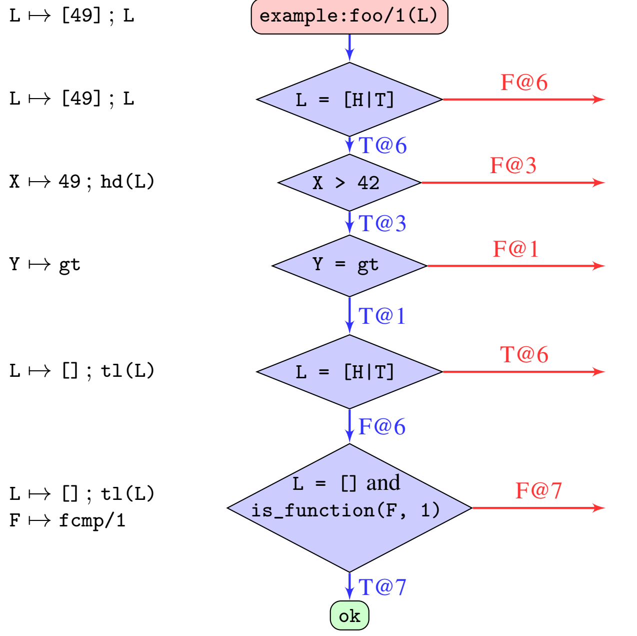

Figure 4. Second execution of `example:foo([49])`.

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 (shown in Figure 4) is generated for this value. The second execution trace results again in the value `ok` being returned and again no bug has been found.

Now there are two paths in the priority queue, those of Figures 3 and 4. In the former, the next alternative to be explored is T@4, whereas in the latter, 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:

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

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 not all possible execution paths have been explored. Sooner or later, alternative F@7 will be considered and we will try to find a term `L` that satisfies:

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

(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:

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

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]`.

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

Figure 5. The syntax of Mini Core Erlang.

To sum up, concolic execution for our running example starts by the user simply specifying an arbitrary call to the function constituting the entry point of the unit being tested (in this case `example:foo([17])`) as a seed, and 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 Figure 1 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 Figure 5.

In Mini Core Erlang, terms can be atoms, integers, floats, lists, tuples or functions. A module is a list of function definitions where some functions are exported. A function definition consists of the function's name, its parameters and the expression that represents its body. Expressions also include `call`, `apply`, `case`, `let` and `do`.

The difference between the first two is that `apply` expressions are used for module-local function applications, while `call` expressions are module-qualified function applications. First the expressions that denote the module and the function are evaluated. These must evaluate to atoms. Then, the actual parameters are evaluated and subsequently the function application is performed.

Case expressions are the only control statements in this grammar. The list of supplied expressions is matched against a sequence of guarded patterns. The first pattern that matches will be selected. The patterns are tried in the order they appear. Evaluation continues from the body of the selected pattern. As mentioned, the Core Erlang compiler ensures that all `case` expressions become exhaustive by adding catch-all clauses if it cannot verify that the programmer provided an exhaustive match.

Built-in functions, such as comparison operators, `is_function/2`, etc., can be included as predefined functions in Mini Core Erlang. Most Erlang built-ins are part of the special module `erlang`, e.g., `erlang:'<'/2`.

We should note that Core Erlang is more expressive than the language defined in Figure 5. For example, it supports `letrec` definitions, unnamed functions, `try-catch` blocks for handling exceptions, and `receive` expressions for message passing between processes. Here we have restricted the language only for presentation purposes. Our tool handles the complete set of Erlang expressions.

3.3 Constraint Generation for Patterns and Guards

Concolic execution involves running an instrumented program both concretely and symbolically. The 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; environments are functions mapping variables to such values:

$$Env = Var \rightarrow Val$$

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

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.

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

Paths are lists of nodes, such as the ones shown in Figures 3 and 4. We are primarily interested in decision nodes. For each such 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.

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. 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.

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 which actually generates constraints, adding nodes to execution paths. When a `case` is evaluated, its 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 expression in a `case`, pattern matched against `true` and `false`.

Let's 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:

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

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/1`, the two environments are:

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

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:

$$\Gamma_c = \{\texttt{F} \mapsto \texttt{fcmp/1}, \texttt{L} \mapsto \texttt{[17]}, \texttt{H} \mapsto \texttt{17}, \texttt{T} \mapsto \texttt{[]}\}$$
$$\Gamma_\sigma = \{\texttt{F} \mapsto \texttt{fcmp/1}, \texttt{L} \mapsto \texttt{L}, \texttt{H} \mapsto \texttt{hd(L)}, \texttt{T} \mapsto \texttt{tl(L)}\}$$

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.

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/1`, 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:

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

the symbolic value of the expression `H < 42 or H > 42` will be `hd(L) < 42 or hd(L)> 42`. For efficiency reasons, a better option is to introduce a fresh symbolic variable for the result of each built-in function, to avoid possible exponential growth when generating symbolic expressions. (Such exponential growth is not only witnessed when serializing terms with shared subterms; in Erlang, it can also be problematic even if such terms are only kept in memory, as term sharing is not always preserved by the Erlang VM [14].)

Using this approach, in the example above where the BIFs `erlang:'<'/2`, `erlang:'>'/2`, and `erlang:or/2` are used, the symbolic result could be `T3` and the symbolic environment would be extended to contain three more fresh variables: `T1`, `T2`, and `T3`.

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

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 should 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 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 give "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 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 [12], 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 [11], the following specification:

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

This expresses the programmer's intention 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` be used in Section 3.1, the constraint paths (1), (2), (3), and (4) would be, 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. On the other hand, the third one 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 given 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.

The interesting part, however, is that this additional constraint rules out the fourth constraint path as unsatisfiable. This is because of the following axiom, which should be adopted by the solver:

$$\texttt{is_integer_list(L)} \implies \texttt{L = []} \vee (\texttt{is_integer(hd(L))} \wedge \texttt{is_integer_list(tl(L))})$$

With this axiom, the fourth constraint path would expand to:

4. $\texttt{X = hd(L)} \wedge \texttt{is_integer(X)} \wedge$
   $\neg(\texttt{X > 42}) \wedge \neg(\texttt{X = 42}) \wedge \neg(\texttt{X < 42})$

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]` as a test case that crashes this unit would be to declare the following type specification for `cmp/1`:

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

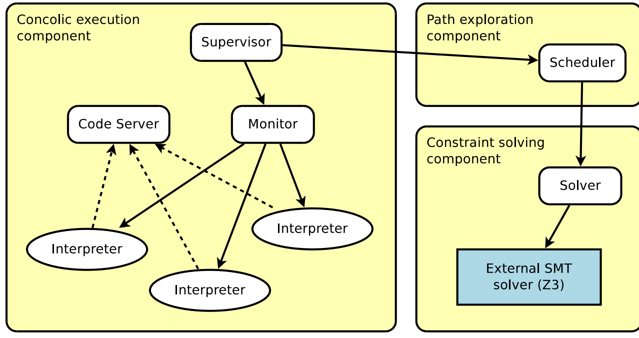

Figure 6. High-level architecture of CutEr.

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 loose ones. 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 or unwanted errors. Note however, that this process depends heavily on the expressiveness of the specification language and also whether the additional constraints can be processed by the underlying constraint solver.

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

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

although 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`), does not express the fact that its first argument 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, [a,b])` will report that the call `lists:nth(39, [0])` leads to a runtime error.

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. A small part is 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 nodes, possibly running on different machines over a network. Although in the rest of this section we describe how the architecture of CutEr supports concurrency and distribution, in this paper we focus on applying concolic testing to the functional subset of Erlang. 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 paper.

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 Figure 6, and outline how these components interact with each other.

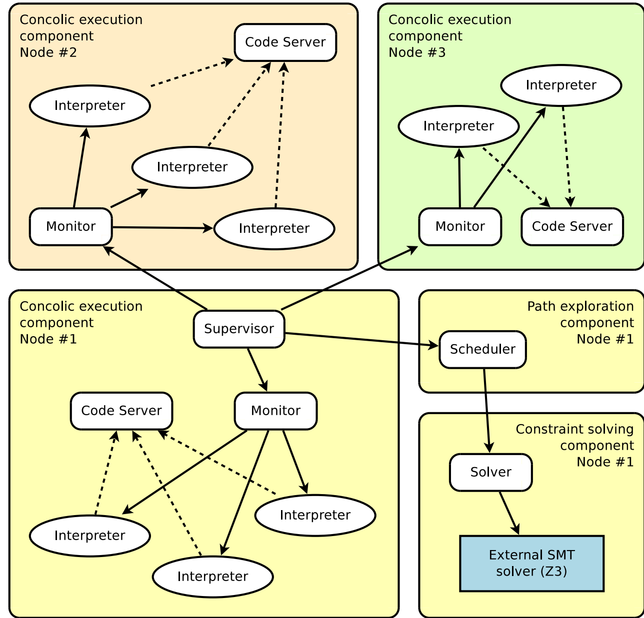

Figure 7. Concolic execution of distributed Erlang applications.

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.

Figure 7 shows the architecture supporting the concolic execution of distributed Erlang applications. The figure shows three Erlang nodes. Notice that only the concolic execution component spans on all three nodes. Furthermore, there is just one supervisor process, running on the first node, where also the path exploration and constraint solving takes place. For scalability reasons, as constraint solving is typically expensive in terms of time and memory, it may

Algorithm 1. Exploration of execution paths.

```
1:  R := ∅                 — errors found
2:  Q := ∅                 — priority queue
3:  visited := ∅           — the set of visited labels
4:  D := ∅                 — mapping of inputs to depths
5:  D_max := 20            — maximum depth of the search tree

6:  function STOREEXECUTION(input, result, ℓ)
7:      if result is not normal termination then
8:          R := R ∪ {⟨input, result⟩}
9:      CS := ∅
10:     i := 0
11:     for all ⟨C, L_T, L_F⟩ ∈ ℓ do
12:         visited := visited ∪ {L_T}
13:         if D[input] ≤ i ≤ D_max then
14:             ENQUEUE(Q, ⟨CS ∪ {N(C)}, i, L_F⟩)
15:         CS := CS ∪ {P(C)}
16:         i := i + 1

17: function REQUESTINPUT()
18:     while Q ≠ ∅ do
19:         ⟨CS, i, L_F⟩ := DEQUEUE(Q)
20:         answer := SMTSOLVE(CS)
21:         if answer = ⟨sat, input⟩ then
22:             D[input] := i + 1
23:             return input
```

be a reasonable choice to use more than one solvers, distributed in more than one nodes.

5.2 Heuristics and Coverage

The component for execution path exploration performs a search on the space of all possible execution paths. As explained in Section 3.1, the order in which execution paths are considered is crucial for the effectiveness of concolic testing; the search strategy that we use has been informally described in the same section. 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$(input, result, \ell)$ and REQUESTINPUT$()$, shown as parts of Algorithm 1. Both functions are re-entrant, so they can be used in a setting where multiple concolic executions and solvers are executed concurrently.

Function STOREEXECUTION$(input, result, \ell)$ 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 Figures 3 and 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$()$ in order to ask for a new input is used, which will be used to start a new concolic execution.

The state of the server consists of the following elements:

- 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 corresponding to 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. Once an execution with some input yielding some concrete result and a constraint path ℓ has been completed, function 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 Z3 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 (in that case, 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 input that will lead 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 it is not satisfiable or if Z3 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 a very 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, which will be given to Z3 in order to find a possible model, i is the order of the constraint that 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 favour 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, and this can be problematic, at least from the point of view of the queue's implementation. 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 will happen 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 that separates 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

```
Term, TList, IList = Datatypes('Term, TList, IList')

Term.declare('int', ('ival', IntSort()))
Term.declare('real', ('rval', RealSort()))
Term.declare('atm', ('aval', IList))
Term.declare('lst', ('lval', TList))
Term.declare('tpl', ('tval', TList))

TList.declare('nil')
TList.declare('cons', ('hd', Term), ('tl', TList))

IList.declare('anil')
IList.declare('acons', ('ahd', IntSort()), ('atl', IList))
```

Figure 8. The representation of Erlang terms in Z3.

shown in Algorithm 1, where the priority queue is assumed to be ordered dynamically.

5.3 Solving Constraints with Z3

The SMT solver that CutEr currently uses is Z3 [7], a solver developed at Microsoft Research and recently gone open-source. It is an efficient SMT solver that is targeted at solving problems in software analysis and software verification. It supports the SMT-LIBv2 standard [2] and provides APIs for C/C++, .NET, OCaml and Python. CutEr uses the Python API, namely Z3Py.

The choice for Z3 was made primarily because Z3 also supports algebraic data types, in addition to basic types like booleans, integers, floats, arrays and bit-vectors. Algebraic data types are an essential feature in our approach, as it allows us to easily represent Erlang terms at a higher level.

Representing Erlang terms in Z3. The most general type in Erlang is `term()`, representing all valid terms. For Mini Core Erlang, as defined in Figure 5, we consider `term()` to consist of integers, floating-point numbers, atoms, lists, and tuples. To simplify presentation, we will consider only *proper* lists, i.e., lists terminating with `[]`. Notice that Erlang (and, in fact, also the syntax in Figure 5) allows improper lists, such as `[1|2]`.

In the notation of Z3, we declare an algebraic type `Term` for this purpose, as shown in Figure 8. We also declare two auxiliary types `TList` and `IList`, representing lists of terms and lists of integers, respectively. The latter is used in the internal representation of atoms, which are modeled as lists containing the ASCII codes of the characters that form them.

Integers and floating-point numbers are built-in types in Z3. In the notation above, the algebraic data type `Term` is defined, having five constructors: `int`, `real`, `atm`, `lst`, and `tpl`. 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 Z3 as follows:

```
t1 = Term.int(42)
t2 = Term.lst(TList.cons(Term.int(17),
       TList.cons(Term.int(42), TList.nil)))
t3 = Term.tpl(TList.cons(Term.int(42),
       TList.cons(Term.atm(
         IList.acons(111, IList.acons(107, IList.anil))
       ), TList.nil)))
```

It is relatively straightforward to define a function $M(t)$, mapping an Erlang term t to its encoding in Z3. This function can be extended to also support variables; we assume that each Erlang variable corresponds to a Z3 variable and, for simplicity, that they have the same name.

Table 1. Encoding of constraints in Z3.

Constraint C	Positive axiom $P(C)$	Negative axiom $N(C)$
$t = t'$	$M(t) = M(t')$	$M(t) \neq M(t')$
$t \neq t'$	$M(t) \neq M(t')$	$M(t) = M(t')$
$t =$ `[]`	`Term.is_lst(`$M(t)$`)` `TList.is_nil(Term.lval(`$M(t)$`))`	
$t =$ `[_|_]`	`Term.is_lst(`$M(t)$`)` `TList.is_cons(Term.lval(`$M(t)$`))`	

t is tuple of size n	`# calculate axioms in As`
	`As = [Term.is_tpl(`$M(t)$`)]`
	`tx = Term.tval(`$M(t)$`)`
	`for i in range(n):`
	`  As.append(TList.is_cons(tx))`
	`  tx = TList.tl(t)`
	`As.append(TList.is_nil(tx))`	

t is integer	`Term.is_int(`$M(t)$`)`	
t is real	`Term.is_real(`$M(t)$`)`	
t is number	`Or(Term.is_int(`$M(t)$`), Term.is_real(`$M(t)$`))`	
t is atom	`Term.is_atm(`$M(t)$`)`	
t is list	`Term.is_list(`$M(t)$`)`	
t is tuple	`Term.is_tpl(`$M(t)$`)`	

Encoding constraints in Z3. 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., appropriate values for unbound variables such that every one of these constraints is true.

We therefore need a way to encode constraints in Z3; 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 Z3 axioms.) Although Z3 obviously supports negation, instead of taking $N(C) =$ `Not`$(P(C))$ as the encoding of the negation of C, we choose to define a separate function $N(C)$ for this purpose, the reason being that, often, we can generate simpler Z3 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 Z3 for a consistent model.

The definition of functions $P(C)$ and $M(C)$ is pretty straightforward. They are given in parallel in Table 1, following the syntactic structure of the constraints. In this table, wherever we do not specify a value for $N(C)$, we take $N(C) =$ `Not`$(P(C))$.

For example, the constraint path (3) on page 5 will be translated as the following set of axioms:

```
Not(And(Term.is_lst(L), TList.is_cons(Term.lval(L))))
Not(And(Term.is_lst(L), TList.is_nil(Term.lval(L)))))
```

Unfortunately, none of the other constraint paths shown as examples in Section 3.1 have simple translations, so we do not show them here. 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 Figure 9. As a result, the constraints that must be encoded in Z3 are simpler comparison constraints between terms of the same simple type, e.g., `lt_int` and `lt_float`. Even `lt_list`, which compares two

```
-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(X), atom_to_list(Y));
%% atoms are smaller than other terms except numbers
'>'(X, _Y) when is_atom(X) -> false;
%% ...
```

Figure 9. Fragment of CutEr's code for emulating `erlang:'>'/2`.

list terms lexicographically, is emulated as a recursive function in Erlang, eventually comparing values of simple types.

Simplifying complex sets of axioms. If the set of axioms given to Z3 is too complex, the solver may not be able to find a model and resort to replying `unknown`. In these cases, we try to simplify the set of axioms, aiming to obtain one whose satisfiability can be verified. For example, operations that the solver cannot handle, such as some non-linear operations, are not recorded at all as symbolic constraints during execution.

A useful kind of simplification is obtained by limiting the number of variables in the universe. Let S be the set of the variables in our model. 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 determine that our set of constraints is satisfiable, it does not help us if no solution is found after trying all possible subsets $F \subset S$.

5.4 Current Limitations

The current version of CutEr can only handle the types of Mini Core Erlang; in particular, it does not support Erlang binaries and maps. Also, in its current version, CutEr does not emulate all of Erlang's BIFs in such a way so as to be able to reason about their results in symbolic form. We do not expect any serious difficulties in supporting binaries, or in emulating more BIFs in a future version.

Also, even though CutEr currently supports higher-order functions, it cannot 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. We believe that some limited support for such higher-order functions can be implemented in a future version of CutEr, using uninterpreted functions in Z3. However, full support of higher-order functions as first-class citizens in concolic testing is bound to hit a wall, not only with Z3 but with most off-the-shelf solvers.

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

Another current limitation is that CutEr does not support many search strategies. Also, from a user's perspective point-of-view, it would be nice for the tool to report some indication of the path coverage that it managed to achieve in case of incomplete searches.

6. Some Experiences

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 all/most 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 on purpose, 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 semantically equivalent variant of the `fcmp/1` function where its `case` statement has been turned into an assertion:

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

No doubt this version may initially manage to confuse some readers, but it does not succeed in deceiving CutEr! The tool still 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]`. This shows both the power of the tool, but also its dependence on its SMT solver component, which works not only as a black box but as black magic in this case.

As mentioned in Section 4, an issue is that the language of type specifications 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:

```
-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. Its 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, which has not RTFM, 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}]).
```

7. Concluding Remarks and Future Work

We have presented an approach to apply concolic testing to functional programs at the level of their core language, rather than at the level of their low-level implementation. 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 high-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 search strategies but also significant engineering effort in order to become scalable and effective. 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.

Acknowledgments

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".

References

[1] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010. DOI: 10.1145/1810891.1810910.

[2] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.

[3] C. Cadar, D. Dunbar, and 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*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[4] R. Carlsson and M. Rémond. EUnit: A lightweight unit testing framework for Erlang. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 1–1, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1. DOI: 10.1145/1159789.1159791.

[5] R. Carlsson, T. Lindgren, B. Gustavsson, S.-O. Nyström, R. Virding, E. Johansson, and M. Pettersson. Core Erlang 1.0.3 language specification. Technical report, Uppsala University, Nov. 2004. URL: https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.

[6] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, New York, NY, USA, 2000. ACM. DOI: 10.1145/351240.351266.

[7] L. De Moura and 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*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. DOI: 10.1007/978-3-540-78800-3_24.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM Press, June 2005. DOI: 10.1145/1065010.1065036.

[9] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, Mar. 2012. DOI: 10.1145/2093548.2093564.

[10] E. Larson and T. Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, Aug. 2003. URL: http://dl.acm.org/citation.cfm?id=1251353.1251362.

[11] T. Lindahl and K. Sagonas. TypEr: A type annotator of Erlang code. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, pages 17–25, New York, NY, USA, 2005. ACM. DOI: 10.1145/1088361.1088366.

[12] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 167–178, New York, NY, USA, 2006. ACM Press. DOI: 10.1145/1140335.1140356.

[13] M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, pages 39–50, New York, NY, USA, 2011. ACM. DOI: 10.1145/2034654.2034663.

[14] N. Papaspyrou and K. Sagonas. On preserving term sharing in the Erlang virtual machine. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, pages 11–20, New York, NY, USA, 2012. ACM. DOI: 10.1145/2364489.2364493.

[15] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 34–44, New York, NY, USA, 2011. ACM. DOI: 10.1145/2001420.2001425.

[16] QuviQ. Erlang QuickCheck. URL: http://www.quviq.com/products/erlang-quickcheck/.

[17] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy Smallcheck: Automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, pages 37–48, New York, NY, USA, 2008. ACM. DOI: 10.1145/1411286.1411292.

[18] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the Fifth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272. ACM Press, 2005. DOI: 10.1145/1081706.1081750.

[19] G. Vidal. Concolic execution and test case generation in Prolog. In *Proceedings of the 24th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 8981 of *LNCS*, pages 167–181. Springer, 2015. DOI: 10.1007/978-3-319-17822-6_10.