A Methodology for Applying Concolic Testing

Manuel Cherep
Abstract

A Methodology for Applying Concolic Testing

Manuel Cherep

Concolic testing is a technique that combines concrete and symbolic execution in order to generate inputs that explore different execution paths leading to better testing coverage. Concolic testing tools can find runtime errors fully automatically using available type specifications. The type specifications in a function define the type of each input. However, most specification languages are never expressive enough, which can lead to runtime errors caused by malformed inputs (i.e. irrelevant errors). Moreover, logic errors causing a program to operate incorrectly without crashing cannot be reported automatically. A universal methodology for any programming language is proposed. Preconditions force the concolic execution to generate well formed inputs before testing a function. On the other hand, postconditions lead to a runtime error when a program operates incorrectly, helping to find logic errors. The results obtained using the concolic testing tool CutEr, in the functional programming language Erlang, show how a program is only tested using well formed inputs specially generated to try to violate the defined postconditions.
Acknowledgements

I am extremely grateful to my supervisor Konstantinos Sagonas at Uppsala University for his valuable feedback sharing his expertise.

I would also like to thank my reviewer Justin Pearson at Uppsala University for his great feedback and dedication.

Last, thank you to Aggelos Giantsios at National Technical University of Athens for always answering my questions regarding CutEr.
# Contents

1 Introduction 4  
  1.1 Motivation and Goals .......................... 5  
  1.2 Contributions .................................. 5  

2 Background 6  
  2.1 Unit Testing .................................... 6  
  2.2 Concolic Testing .................................. 6  
  2.3 Erlang ......................................... 8  

3 Methodology 10  
  3.1 Problem ......................................... 10  
  3.2 Preconditions .................................... 11  
  3.3 Postconditions ................................... 14  
  3.4 Unit Testing ..................................... 16  

4 Concolic Testing in Erlang 20  
  4.1 CutEr ............................................. 20  
  4.2 Limited Type Language ......................... 20  
  4.3 A More Complex Example ....................... 22  

5 Results 26  
  5.1 Average Function ............................... 26  
  5.2 Rally Program ................................... 28  

6 Discussion 31  
  6.1 CutEr ............................................. 31  
  6.2 Multiple Postconditions ....................... 31  
  6.3 Integration ...................................... 32  
  6.4 Related Work ................................... 32  

7 Conclusions and Future Work 33  

8 Bibliography 34  

A Source Code 36
Chapter 1

Introduction

Testing [1] is the predominant method in industry to ensure software correctness and reliability. More sophisticated software techniques are necessary as a result of the growing complexity of software that make errors difficult to find.

Software systems nowadays have thousands of lines of code with multiple different execution paths. Therefore, it is infeasible for a test engineer to manually write tests covering all the possible execution paths, which results in poor software reliability.

In order to have better testing coverage and improve software reliability, automated software testing tools and techniques have been created. Some of these techniques use random testing where inputs are generated randomly, such as property-based testing [2, 3, 4, 5]. The problem with such techniques is that many different inputs may be repeatedly testing the same behaviors, and random inputs do not guarantee covering different execution paths [6]. Moreover, it requires manually writing properties for the program to be tested.

Concolic testing is a technique that combines concrete and symbolic execution of a program [7]. The goal is to generate inputs that exercise all the different execution paths. The program is executed concretely and at the same time symbolic constraints are collected during the symbolic execution generating a path constraint. The path constraint is then negated and solved using a constraint solver generating a new input that will exercise a different execution path.

Concolic testing has gained popularity in imperative programming languages such as C and Java [8, 9, 10], but it has recently been applied to functional languages [11]. CutEr [11], a concolic testing tool for Erlang, is the first tool applying concolic execution to a functional language.

Concolic testing tools are fully automatic, inputs are generated utilizing data type specification available. However, most type specification languages are not expressive enough, which can lead a concolic testing tool to generate malformed inputs (i.e. inputs that did not respect completely the expected type), later reported as inputs leading to runtime errors. Moreover, concolic testing tools require more information in order to find logic errors that do not necessarily
lead to runtime errors.

A universal methodology is proposed which guarantees testing programs with well formed inputs, finding logic errors in the results.

1.1 Motivation and Goals

The aim of this project is to define a methodology for applying concolic testing independently from the programming language and tool used. The goal of this thesis is to design a methodology which guarantees testing functions with well formed inputs and helps to find logic errors hidden in the program. The function under test can only be executed with inputs that satisfy all the preconditions. Furthermore, postconditions are provided to find logic errors when one of them is violated.

There are different tools for concolic testing implemented in different programming languages with different characteristics. However, common patterns can be extrapolated in order to find a universal methodology that can be applied to all of them.

For the sake of illustration the functional programming language Erlang and the tool CutEr [11] are being used.

1.2 Contributions

The main contribution of this thesis is providing guidelines for testing a program using concolic testing. The designed methodology is an important contribution, because it is the first one for applying concolic testing. Although it has been inspired by property-based testing [2, 3, 4, 5] and CutEr [11]. This methodology solves the two recurrent problems when using concolic testing tools. It shows real examples where the program is only tested with inputs that satisfy all the constraints (i.e. well formed inputs). Furthermore, it shows how to introduce assertions that can lead to finding errors in the logic of a program.
Chapter 2

Background

Testing coverage measures the range of different behaviors tested (i.e. the number of execution paths exercised). A higher testing coverage means that a program has been more thoroughly tested.

2.1 Unit Testing

Unit testing is a method by which different individual components or units of a program, which are collections of functions or procedures, are tested independently. A program is divided into different units that are then tested independently.

In this method it is necessary to specify input values for every unit that is going to be tested. The specification of such inputs can be done manually but this does not ensure that all possible execution paths would be exercised during the testing. It is usually an arduous work and the testing coverage is low.

Automatically generating values for the inputs would reduce the effort of writing these values manually and it might increase the testing coverage.

2.2 Concolic Testing

Concolic testing [7] is a technique that combines concrete and symbolic execution of a program to generate inputs that exercise all the different execution paths. The goal is to achieve high testing coverage. The combination of concrete and symbolic execution running simultaneously is called concolic execution.

In concolic execution, the concrete execution is the normal execution of the program. Symbolic execution [12, 13, 14] collects symbolic constraints over the symbolic inputs of the program at each conditional branch. The symbolic execution has to be done without altering the concrete execution of the program which is accomplished by adding code that collects the constraints, also known as instrumentation. The program is executed with a symbolic value (i.e. a variable has a symbol associated with it instead of a value) for each variable
that depends on inputs to the program. During the execution the same path is followed until there is a conditional expression where the execution follows one of the possible branches based on variables that have symbolic values. At this conditional expression, with the given symbolic values it is possible to create a symbolic constraint that describes the possible input values that lead the execution of the program to follow one branch or another (i.e. determines an execution path). At the end of the concolic execution, the conjunction of all the symbolic constraints collected at each branch point is called path constraint. A path constraint describes the input values that lead the concrete execution to follow an execution path.

In a concolic testing tool, concrete random input values are generated to execute the program that is going to be tested. During this first concrete execution the symbolic constraints are collected creating a path constraint. Each constraint in the path is then negated and solved systematically using a constraint solver, if it is feasible, generating new test inputs that will exercise an unexplored execution path in the next iteration. This process is repeated until all feasible execution paths in the program have been exercised. In order to understand this better a concolic execution will be explained using the example shown in Figure 2.1.

```erlang
-spec foo(number(), number()) -> ok.
foo(X, Y) ->
  case bar(X) =:= Y of
    true ->
      erlang:error("RUNTIME ERROR");
    false ->
      ok
  end.

-spec bar(number()) -> number().
bar(Z) -> Z + 1.
```

Figure 2.1: A running example

Figure 2.2 shows the control-flow graph of the function `foo` in the previous example. The pink node corresponds to the entry point of the function; the blue node represents the condition expression; the green node represents a result point (i.e. the end of an error-free path); the red node corresponds to an error in the execution.
Figure 2.2: Control flow graph for the function foo in the example

An example with inputs and its corresponding path constraints for a concolic execution of the function foo in Figure 2.1 is shown in Table 2.1. It is assumed that the first input values are \{X \mapsto 2, Y \mapsto 7\}. The constraints collected during the concrete execution of the function with the given inputs result in the path constraint \(<X_0 + 1 \neq Y_0>\), where \(X_0\) and \(Y_0\) are the symbolic values of \(X\) and \(Y\) respectively. This concrete execution does not fail. The last constraint in the path constraint is negated (i.e. the only constraint in this example) and the resulting path constraint is \(<X_0 + 1 = Y_0>\). This path constraint is solved generating values for the next concrete execution. Assuming the values generated are \{X \mapsto 1, Y \mapsto 2\}, the concrete execution reveals the error in the code. If the path constraint had more than one constraint then it would be necessary to iterate again, negating the next constraint and solving it.

<table>
<thead>
<tr>
<th>Input</th>
<th>Path Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>{X \mapsto 2, Y \mapsto 7}</td>
<td>(&lt;X_0 + 1 \neq Y_0&gt;)</td>
</tr>
<tr>
<td>{X \mapsto 1, Y \mapsto 2}</td>
<td>(&lt;X_0 + 1 = Y_0&gt;)</td>
</tr>
</tbody>
</table>

Table 2.1: Inputs and path constraints of a concolic execution of function foo

### 2.3 Erlang

Erlang [15] is a concurrent functional programming language designed for programming fault-tolerant distributed systems.

Erlang is dynamically typed which means that the type checking is done at runtime. Type checking is the verification of the constraints of types in a program. Figure 2.3 shows an example of a function that fails because types do not match. This function is not rejected by the compiler but fails at runtime.
foo() ->
  %% Bar is a string
  Bar = "Hello World!",
  %% This will fail because Bar is not a integer
  integer_to_list(Bar).

Figure 2.3: A function containing a runtime error

Erlang achieves fault-tolerance having a process supervising the behavior of another process. If the supervised process fails, the supervising process must be able to detect it and take over to handle the error. While other languages try to make error-free programs, Erlang assumes that errors will happen.

The recommended way of programming in Erlang is non-defensive programming. A function should crash if there is an unexpected behavior instead of handling every possible scenario. This programming style leads to more clean and compact code. Figure 2.4 shows examples of defensive and non-defensive programming, where the specification of the function represents the expected type.

```
-spec non_defensive_add1(number()) ->
    number().
non_defensive_add1(Value) ->
    Value+1.

-spec defensive_add1(any()) ->
    number() | value_is_not_number.
defensive_add1(Value) when is_number(Value) ->
    Value+1;
defensive_add1(Value) ->
    value_is_not_number.
```

Figure 2.4: Non-defensive and Defensive functions

The Erlang philosophy is “let it crash”. Failing processes should crash immediately and another supervising process will detect the crash and correct the error.
Chapter 3

Methodology

There are many different programming paradigms with different characteristics. However, it is possible to extrapolate common problems mentioned in the following chapter to define a methodology that can be applied to all of them.

3.1 Problem

Concolic testing is a powerful testing technique to find runtime errors. There are errors in a program, also known as logic errors, that cause the program to operate incorrectly but do not lead the program to crash. Since these kinds of errors do not lead the program to crash during runtime, a concolic testing tool will not find them automatically unless more information is provided. On the other hand, some functions require the inputs to always be well formed or otherwise these functions would crash. It is more convenient a concolic testing tool that reports runtime errors caused only by well formed inputs.

Figure 3.1 shows a function that aims to calculate the average of two integer numbers. The two inputs are given as strings, but it is implicitly assumed that these strings can be converted to integers and the function will crash if this condition is not satisfied. Moreover, the function contains a logic error due to operator precedence. This example is a simplified version of real examples where strings are used to represent inputs that are telephone numbers, e-mail addresses, etc.
average(A, B) ->
A_Int = list_to_integer(A),
B_Int = list_to_integer(B),
%% \((\text{A}_{\text{Int}}+\text{B}_{\text{Int}})/2\) is the correct code
A_{\text{Int}}+B_{\text{Int}}/2.

Figure 3.1: Average function containing a logic error

The concolic execution of the function in Figure 3.1 is going to generate values for the two inputs and the only constraint is that both values have to be string. Assuming the values generated are \(A = "\) and \(B = "\) (i.e. the empty string) the execution of the function would lead to a runtime error. However, this cannot be considered a wrong implementation nor a false positive. It is not a false positive because it is true that the function would crash with the given input. However, this scenario should not occur since it violates our implicit condition that both inputs must be convertible to integer. This situation is not interesting and it is better if it is not reported.

On the other hand, assuming both inputs are well formed (i.e. strings convertible to integer), the concolic execution would not lead to a runtime error because even though the function operates incorrectly, it does not crash.

### 3.2 Preconditions

The preconditions can solve the problem of testing a function with inputs that are not well formed. A precondition is a condition that must be fulfilled before continuing the execution of the original function under testing. If all the necessary preconditions are satisfied that means that the inputs are well formed.

One way of satisfying preconditions is to force inputs to satisfy them (e.g. if a list should have an special last element, attach it before continuing with the test). Figure 3.2 shows another way a precondition can be satisfied in order to continue executing the original function under testing, which in this case is called testme.
It is important to create tests that do not modify the original function. Therefore, the code shown in Figure 3.2 is a new function created specially for concolic testing of this program unit. This will be explained in more detail in Section 3.4.

Figure 3.3 shows the control-flow graph of the function in Figure 3.2. A yellow node represents variable bindings and calls to other functions.

This way the precondition ensures that `testme` will only be called with well formed inputs, assuming that the precondition is implemented correctly. In case the input is not well formed the precondition will not be satisfied, returning `false`. The concrete execution will not lead to a runtime error, it would follow the execution path of the branch `false`, which returns normally. This path constraint is negated and solved with a constraint solver, generating inputs that would lead the concrete execution to follow the other branch `true`. Taking advantage of the constraint solver, well formed inputs are generated to test the function `testme`.

Figure 3.2: Test using a precondition

```prolog
test(Input) ->
    case precondition(Input) of
        true ->
            %% Input is well formed
            %% Call the function to be tested
            testme(Input);
        false ->
            %% Skip input
            ok
    end.
```

Figure 3.3: Control flow graph for the function test with a precondition
-spec test_average(string(), string()) -> ok.
  test_average(A, B) ->
  case precondition(A, B) of
    true ->
      average(A, B),
      ok;
    false ->
      %% Skip input
      ok
  end.

-spec precondition(string(), string()) -> boolean().
  precondition(A, B) ->
    is_list_integer(A) andalso is_list_integer(B).

-spec is_list_integer(string()) -> boolean().
  is_list_integer([H]) when H >= $0, H <= $9 ->
    true;
  is_list_integer([H|T]) when H >= $0, H <= $9 ->
    is_list_integer(T);
  is_list_integer(_) ->
    false.

Figure 3.4: Testing function average using a precondition

Figure 3.4 shows a function test_average created to test the function in Figure 3.1. This test uses a precondition that returns true if both inputs can be converted from string to integer, meaning that the inputs are well formed; otherwise it returns false. Figure 3.5 shows the control-flow graph of the function test_average.

Figure 3.5: Control flow graph for the function test_average with a precondition
3.3 Postconditions

The postconditions can contribute to finding logic errors in a program. A post-condition is a condition that must always be true after the execution of the original function under test. The idea of a postcondition is similar to a property in property-based testing [2, 3, 4, 5].

Figure 3.6 shows how a postcondition must be satisfied after executing the original function under testing, which in this case is called testme.

```erlang
test(Input) ->
    Result = testme(Input),
    case postcondition(Input, Result) of
        false ->
            %% Runtime error
            erlang:erlang("Postcondition failed");
        true ->
            ok
        end.
```

Figure 3.6: Test using a postcondition

The code shown in Figure 3.6 is a new function created specially for the test. This will be explained in more detail in Section 3.4. Figure 3.7 shows the control-flow graph of the function in Figure 3.6.

Figure 3.7: Control flow graph for the function test with a postcondition
The function testme is called and the result saved, collecting the necessary constraints. The concrete execution will lead to a runtime error if the postcondition is not satisfied or as a consequence of the execution of the function testme. It is assumed that the postcondition is implemented correctly and it is satisfied during the first concrete execution. The path constraint collected will be negated and solved to generate an input that can violate the postcondition, which is only possible if there is a logical error in the function testme. Otherwise, the constraint solver will not be able to generate inputs to violate the postcondition.

```erlang
-spec test_average(string(), string()) -> ok.
test_average(A, B) ->
    Result = average(A, B),
    case postcondition(A, B, Result) of
        false ->
            %% Runtime error
            erlang:error("Postcondition failed");
        true ->
            ok
    end.

-spec postcondition(string(), string(), number()) -> boolean().
postcondition(A, B, Result) ->
    %% Assuming the input is well formed at this point
    A_Int = list_to_integer(A),
    B_Int = list_to_integer(B),
    max(A_Int, B_Int) >= Result.
```

Figure 3.8: Testing function average using a precondition

Figure 3.8 shows a function test_average created to test the function in Figure 3.1. The postcondition of the function average is that the result has to be less or equal than the maximum of both inputs. Figure 3.9 shows the control-flow graph of the function test_average.
3.4 Unit Testing

The methodology presented consisting of preconditions and postconditions is modular (i.e. preconditions and postconditions are independent). Sometimes the preconditions are not necessary and can be omitted, for instance when a function is programmed defensively and it should also be tested with inputs that are not well formed. The postconditions can also be omitted if necessary.

Being able to test a function without modifying it is very important. Therefore each test is a new function wrapping up the function that is going to be tested. The test function should have the same arity as the function to be tested, both having the same input specification. The preconditions and postconditions are included in the test function.

Figure 3.10 shows a test applying the methodology with both preconditions and postconditions testing the function testme. Figure 3.11 shows the corresponding control-flow graph.
\[
\text{test}(\text{Input}) \rightarrow \\
\quad \text{case precondition}(\text{Input}) \text{ of} \\
\quad \quad \text{true} \rightarrow \\
\quad \quad \quad \text{Result} = \text{testme}(\text{Input}), \\
\quad \quad \quad \text{case postcondition}(\text{Input}, \text{Result}) \text{ of} \\
\quad \quad \quad \quad \text{false} \rightarrow \\
\quad \quad \quad \quad \quad \%\% \text{Runtime error} \\
\quad \quad \quad \quad \quad \text{erlang:}\text{error}("\text{Postcondition failed}"); \\
\quad \quad \quad \quad \text{true} \rightarrow \\
\quad \quad \quad \quad \quad \text{ok} \\
\quad \quad \quad \text{end}; \\
\quad \quad \text{false} \rightarrow \\
\quad \quad \quad \%\% \text{Skip input} \\
\quad \quad \quad \text{ok} \\
\quad \text{end}. \\
\]

Figure 3.10: Test using the methodology

Figure 3.11: Control flow graph for the function test with the methodology
-module(foo).

-spec average(string(), string()) -> number().
average(A, B) ->
  A_Int = list_to_integer(A),
  B_Int = list_to_integer(B),
  A_Int+B_Int/2.

-spec test_average(string(), string()) -> ok.
test_average(A, B) ->
  case precondition(A, B) of
    true ->
      Result = average(A, B),
      case postcondition(A, B, Result) of
    false ->
        %% Runtime error
        erlang:error("Postcondition failed");
      true -> ok
    end;
    false ->
        %% Skip input
        ok
    end.

-spec precondition(string(), string()) -> boolean().
precondition(A, B) ->
  is_list_integer(A) andalso is_list_integer(B).

-spec is_list_integer(string()) -> boolean().
is_list_integer([H]) when H >= $0, H =< $9 ->
  true;
  is_list_integer([H|T]) when H >= $0, H =< $9 ->
  is_list_integer(T);
  is_list_integer(_) -> false.

-spec postcondition(string(), string(), number()) -> boolean().
postcondition(A, B, Result) ->
  %% The input is well formed at this point
  A_Int = list_to_integer(A),
  B_Int = list_to_integer(B),
  max(A_Int, B_Int) >= Result.

Figure 3.12: Testing function average using the methodology
Figure 3.13: Control flow graph for the function test_average with the methodology

Figure 3.12 shows the functions `average` and `test_average` which have been used in previous sections. On this occasion the example is presented completely following the methodology. Figure 3.13 shows the corresponding control-flow graph.

The function `test_average` is going to be executed in Section 5.1 after presenting a concolic testing tool in the same chapter.

It is also important to emphasize that the same logic can be applied using more than one precondition and postcondition, it has been kept simple for the sake of illustration. However, in more complex functions it could be necessary to use many preconditions and postconditions. The structure would be exactly the same. All the preconditions have to be satisfied in order to execute the function under test. On the other hand, in case any of the postconditions is violated, a runtime error must occur.
Chapter 4

Concolic Testing in Erlang

Erlang is a dynamically typed programming language, type checking is done at run-time instead of compile-time. Erlang comes with a type specification language to define types that are later used for documentation or by testing tools.

4.1 CutEr


The heuristic used by CutEr in the concolic execution to explore different execution paths is based on path coverage. All execution paths form a search tree with a certain depth, which can be set as a boundary to stop the concolic execution. For each execution path the first decision node whose reversed label has not been visited yet, is explored. The exploration continues until all possible execution paths have been explored, or when a certain depth in the search tree has been reached. The constraint solver currently used by CutEr is Z3 [16].

Erlang type specification language is supported by CutEr. CutEr considers type specifications as preconditions of program inputs. These are considered as additional constraints that are never negated to avoid breaking a precondition.

4.2 Limited Type Language

A subset of the type specification language in Erlang is shown in Figure 4.1. Erlang specifications are never checked during compilation. However, type specifications can be used to document function interfaces or provide information for testing tools [17].
Type :: any()
    | none()
    | pid()
    | port()
    | reference()
    | []
    | atom()
    | Bitstring
    | float()
    | Fun
    | Integer
    | List
    | Tuple
    | Union
    | UserDefined

Bitstring :: <<>>
    | <<_:M>>
    | <<_:*_N>>
    | <<_:M, _:_*N>>

Fun :: fun()
    | fun((...) -> Type)
    | fun(() -> Type)
    | fun((TList) -> Type)

Integer :: integer()
    | Erlang_Integer
    | Erlang_Integer..Erlang_Integer

List :: list(Type)
    | maybe_improper_list(Type1, Type2)
    | nonempty_improper_list(Type1, Type2)
    | nonempty_list(Type)

Tuple :: tuple()
    | {}
    | {TList}

TList :: Type
    | Type, TList

Union :: Type1 | Type2

Figure 4.1: Erlang Type Language
The language, as any other type language, is limited and it cannot represent all possible constraints for all types. An illustrative example [11] is the following type declared in the calendar module of the standard library:

\[-\text{type date()} :: \{\text{Year::non\_neg\_integer()}, \text{Month::1..12}, \text{Day::1..31}\}.\]

which is used in some functions of this module. CutEr reports that these functions will fail with the input 42,4,31, which according to the specification is correct. It is not a relevant error since it is not a real date, but CutEr cannot know that.

This issue was already faced in the example shown in Figure 3.12. The function \text{average} should take strings that can be convertible to integer, but the specification language is not expressive enough. Therefore, it is only specified that the inputs have to be strings.

CutEr generates inputs that satisfy the specifications, but if they are not expressive enough then some of the inputs could be not well formed. This problem needs to be addressed for the automatic use of the tool and the preconditions explained in Section 3.2 can help CutEr to always generate well formed inputs.

### 4.3 A More Complex Example

The next problem that is going to be tested is known as the rally problem [18]. The goal is to calculate the minimum number of moves necessary for a car to finish the track without exceeding the speed limit. A track is divided in section units, each one of them with a specific speed limit. After a move the speed can be increased or decreased. The speed determines the number of moves, for each 10 km/h the car moves 1 unit. Speed can be increased or decreased by multiples of 10, and there is a maximum acceleration and maximum brake speed. This means that a car cannot accelerate (brake) at once more than the maximum acceleration (braking) speed.

The preconditions required for this problem are: the end of the track is indicated with the element \{0,0\}, the maximum number of units in a track is 10,000, speed is always a positive multiple of 10 smaller than 250.

Appendix A shows a correct implementation to solve the rally problem. Figure 4.2 shows part of the code containing an error, because the speed chosen is always the limit of the current part of the track. This is wrong, because there is a maximum acceleration that cannot be violated.
-spec rally(speed(), speed(), track()) -> moves().

rally(MaxA, MaxB, Track) ->
   {Moves, _AccSpeed} = rally(MaxA, MaxB, Track, 0, 0, []),
   Moves.

-spec rally(speed(), speed(), track(),
            speed(), moves(), [speed()]) -> {moves(), [speed()]}.

rally(_MaxA, _MaxB, [], _CurrentSpeed, Moves, Acc) ->
   {Moves, Acc};

rally(_MaxA, _MaxB, [{0,0}], _CurrentSpeed, Moves, Acc) ->
   {Moves+1, Acc};

rally(MaxA, MaxB, Track = [{_N,SpeedLimit} | _SubTrack],
      _CurrentSpeed, Moves, Acc) ->
     %% Correct code:
     %% HighestSpeed = CurrentSpeed + MaxA,
     %% SpeedCandidate =
     %%   case HighestSpeed >= SpeedLimit of
     %%     true ->
     %%       SpeedLimit;
     %%     false ->
     %%       HighestSpeed
     %% end,
     SpeedCandidate = SpeedLimit, %% Wrong code
     {NewTrack, NewSpeed} =
     optimal_speed(Track, MaxB, SpeedCandidate),
     NewAcc = Acc ++ [NewSpeed],
     rally(MaxA, MaxB, NewTrack, NewSpeed, Moves+1, NewAcc).

---

Figure 4.2: Function rally with a bug
Figure 4.3 shows functions `test_rally_precondition` and `test_rally` created to test the rally implementation. Preconditions related to the speed and the maximum number of units in a track are going to be satisfied automatically with the types specified (i.e., speed, units, track and moves). CutEr understands this language and it is going to generate inputs that satisfy this preconditions. On the other hand, the precondition that the last element of a track must be \{0,0\} is not cover with the specified types. One way of satisfying a precondition, as mentioned in Section 3.2, is to force inputs to satisfy it. In this scenario, all tracks have the precondition that their last element has to be \{0,0\}, which it is impossible to specify using the type language in Erlang. Therefore, each track is forced to satisfy that precondition appending the required last element before calling the original function under test.

The function `test_rally_precondition` in Figure 4.3 guarantees that the function `rally` is going to be called with well-formed inputs. On the other hand, the function `test_rally` is also setting a postcondition. The speed difference between the previous speed and the next one can be at most the maximum acceleration.
-spec test_rally_precondition(speed(), speed(), track()) -> ok.
  test_rally_precondition(MaxA, MaxB, Track) ->
    %% Precondition: The last element of the track must be {0,0}
    WFTrack = Track ++ [{0,0}],
    rally(MaxA, MaxB, WFTrack),
    ok.

-spec test_rally(speed(), speed(), track()) -> ok.
  test_rally(MaxA, MaxB, Track) ->
    %% Precondition: The last element of the track must be {0,0}
    WFTrack = Track ++ [{0,0}],
    {_Moves, Acc} = rally(MaxA, MaxB, WFTrack, 0, 0, []),
    %% Postcondition: Cannot accelerate more than MaxA
    InitSpeed = 0,
    case postcondition(MaxA, InitSpeed, Acc) of
        false ->
            erlang:error("Postcondition failed");
        true ->
            ok
        end.

-spec postcondition(speed(), speed(), [speed()]) -> boolean().
  postcondition(_MaxA, _PreviousSpeed, []) ->
    true;
  postcondition(MaxA, PreviousSpeed, [Speed | Rest]) ->
    case PreviousSpeed + MaxA >= Speed of
        true ->
            postcondition(MaxA, Speed, Rest);
        false ->
            false
        end.

Figure 4.3: Code to test function rally
Chapter 5

Results

The results in this chapter were obtained using the following system requirements:

- CutEr version 0.1: Concolic testing tool [11].
- Z3 version 4.4.2: Constraint solver [16].
- Erlang/OTP version 18.7.2: Programming language [15].
- Python version 2.7.10: Programming language [19].

5.1 Average Function

At this point, CutEr can be used to test the function in the example shown in Figure 5.1. First the original function is tested automatically and then it is tested implementing the proposed methodology. In order to define a different depth, explain in the search tree than the default one the option -d is used.

Firstly, running CutEr on the original function average:

```bash
$ cuter foo average \"[\"0\", \"0\"]\"
```

where `cuter` is the name of the script provided with the tool and its arguments are the name of the module, the function to test and an input for the given function. In case the input of the function is not well formed, it will be reported by CutEr. The output as a result of executing this command is:

```
Testing foo:average/2 ...
<more dots and x’s here deleted>
=== Inputs That Lead to Runtime Errors ===
#1 foo:average("", ")
#2 foo:average("-", ")
#3 foo:average("+", ")
#4 foo:average([45,0], ")
```
As expected, CutEr reported errors that are strings not convertible to integer. The inputs lead to runtime errors, but they are irrelevant to us because such strings should never be inputs. On the other hand, other more interesting errors (e.g. `foo:average("2", "1")`) are not found because there is no postcondition to help CutEr.

Running CutEr on the function `test_average` in Figure 3.4 (i.e. only using a precondition) generates the following output:

```
$ cuter foo test_average ' ["0", "0"] ' -d 100

Testing foo:test_average/2 ...

.x...x.xxxxxxxxx.x...xxx.xx.xxx...xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...

No Runtime Errors Occurred

No runtime errors occurred because the function under test is only executed with well formed inputs (i.e. strings convertible to integer). However, there is a logic error in the `average` function that has not been reported yet.

Finally, running CutEr on the function `test_average` with the whole implemented methodology seen in Figure 3.12:

```
$ cuter foo test_average ' ["0", "0"] ' -d 50

Testing foo:test_average/2 ...

........xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...

```
foo: test_average("9", "9")

=== Inputs That Lead to Runtime Errors ===
#1 foo: test_average("9", "9")

CutEr reports a runtime error that is particularly interesting because it is violating the postcondition. Executing the input reported in an Erlang shell:

```
$ cuter rally rally '[(20, 10), [[5, 20], [0,0]]]' -d 80
```

Testing rally:rally/3 ...

CutEr reports two inputs that lead to runtime errors. However, a closer look reveals that one of the preconditions is not met: the last element of a track must be \(\{0,0\}\). The reported inputs are irrelevant because the inputs are not well formed.
Running CutEr on the function `test_rally_precondition` in Figure 4.3 generates the following output:

```
$ cuter rally test_rally_precondition `[20, 10, {[5, 20]}]` -d 80

Testing rally:test_rally_precondition/3 ...
.xx.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxx..xxxxxxxxxxxxxxxxxxxxxxxxxxxx
<more dots and x's here deleted>
No Runtime Errors Occurred
```

No runtime errors occurred now that the function is only executed with well formed inputs. However, there is a logic error in the rally function that has not been reported yet.

Finally, running CutEr on the function `test_rally` with the whole implemented methodology.

```
$ cuter rally test_rally `[20, 10, {[5, 20]}]` -d 80 -s 4

Testing rally:test_rally/3 ...
.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
rally:test_rally(10, 100, {[18,180]})
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
<more dots and x's here deleted>
=== Inputs That Lead to Runtime Errors ===
#1 rally:test_rally(10, 100, {[18,180]})
<more inputs here deleted>
```

CutEr reports a runtime error that is violating the postcondition. In order to execute the rally function with the reported input it is necessary to append the last element `{0,0}`. CutEr does not report the input with `{0,0}` as last element, because the precondition was met explicitly in the test. Executing the input reported in an Erlang shell:
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4]
[async-threads:10] [hipe] [kernel-poll:false]

Eshell V7.3 (abort with \^[G]
1> rally:rally(10, 100, [{18,180}, {0,0}]).
2> rally:test_rally(10, 100, [{18,180}]).
** exception error: "Postcondition failed"
in function rally:test_rally/3 (rally.erl, line 112)

Executing the test with the given input generates an exception because the postcondition failed.
Chapter 6

Discussion

The discussions in this chapter focus on the current limitations of CutEr, multiple postconditions and integration in existing projects.

6.1 CutEr

CutEr version 0.1 is still under heavy development. A concolic testing tool such as CutEr needs to execute symbolically as much code as possible in order to have more constraints to generate inputs that exercise different execution paths. At this moment, some built-in functions (BIFs) in Erlang are not supported symbolically. The examples shown in this report were prepared to only use functions that CutEr can execute symbolically in order to get the most out of it. This limitation made impossible to test more complex programs because some functions were not supported symbolically. As a result of this thesis some bugs in CutEr were reported.

CutEr is a powerful testing tool that can find corner cases that would be very difficult to find using other testing techniques. However, the scope of this thesis is to show a methodology to provide the tool with more information to find logic errors avoiding not well formed inputs. A complete demonstration of the power of CutEr cannot be covered in this project.

6.2 Multiple Postconditions

The examples shown in this thesis contain only one postcondition. However, one may need many postconditions to fully test an unit. Each postcondition adds more complexity to the search tree of constraints and slows the execution. In such a case it is more efficient to create different tests testing different postconditions.
### 6.3 Integration

One of the great advantages of using concolic testing tools such as CutEr is that they work out of the box. On the other hand, other testing tools (e.g. property-based testing tools) are not fully automatic requiring more testing expertise. The methodology proposed in this thesis only requires knowledge of the programming language used. Preconditions and postconditions are written using standard functions without having to create special generators or properties as in the case of property-based testing tools.

Furthermore, unit tests can become concolic tests easily. It is only necessary to modify an existing unit test to take arguments that are going to be generated by the concolic testing tool. Thus, a unit test becomes a concolic test with a higher testing coverage.

### 6.4 Related Work

The problem of testing functions with well formed inputs was already addressed [11] in a similar way with a previous validation of the inputs. To the best of my knowledge there is no other previous work defining a methodology in concolic testing. However, property-based testing techniques [2, 3, 4, 5] follow a similar structure. In property-based testing inputs are randomly generated with a manually written generator. On the other hand, in the proposed methodology preconditions only validate whether an input is well formed or not. However, properties in property-based testing are similar to postconditions in the methodology here proposed.
Chapter 7

Conclusions and Future Work

Concolic testing is a powerful technique that combines concrete and symbolic execution to generate inputs that exercise different execution paths increasing testing coverage.

Concolic testing tools are automatic but may report irrelevant errors if the inputs used to test are not well formed. Moreover, logic errors may not be reported without providing more information.

In this thesis a methodology is proposed where preconditions always guarantee that the inputs used to test a function are well formed. The function under testing is only executed with inputs that satisfy all the preconditions. Furthermore, logic errors can be found setting postconditions that produce a runtime error in case the result of the function under testing does not satisfy any postcondition.

The results obtained with the concolic testing tool CutEr show that irrelevant errors (i.e. inputs not satisfying all the preconditions) are not reported. Moreover, CutEr is able to find logic errors reporting inputs that violate a defined postcondition.

Once CutEr supports more built-in functions (BIFs) it would be necessary to apply the described methodology in a bigger code base. It would be particularly interesting to study the performance of the proposed design in a bigger code base, since preconditions and postconditions make the search tree grow increasing its complexity.

Another challenge would be to obtain results applying this methodology in different programming languages. Each programming language is different and the methodology may not cover all the necessities. Moreover, there are specific features in every language where a more specific design could lead to finding more errors in a code base.
Chapter 8

Bibliography


Appendix A

Source Code

I. rally.erl

-module(rally).

-compile(rally).

-compile(export_all).

%% Brief explanation of the algorithm:
%%
%% 1. Choose the highest speed possible.
%%
%% 2. Moves all the units at that speed. Check if it is breaking
%% any limit. If a limit is broken then step 4.
%%
%% 3. Start braking as much as possible to see if it is possible to
%% continue without breaking limits (i.e. If there is enough
%% “time” to decrease the speed). If a limit is broken then
%% step 4.
%%
%% 4. Try next candidate speed (i.e. current speed minus 10).
%% Go back to step 2.
%%
%% 5. Once the speed chosen does not break any limit, the car
%% moves. Go back to step 1 until the track is completed.

-type speed() :: 10 | 20 | 30 | 40 |
50 | 60 | 70 | 80 |
90 | 100 | 110 | 120 |
130 | 140 | 150 | 160 |
170 | 180 | 190 | 200 |
210 | 220 | 230 | 240.
type units() :: 1..10000.
type track() :: [{units(), speed()},...].
type moves() :: non_neg_integer().

-spec rally(speed(), speed(), track()) -> moves().
rally(MaxA, MaxB, Track) ->
    {Moves, _AccSpeed} = rally(MaxA, MaxB, Track, 0, 0, []),
    Moves.

-spec rally(speed(), speed(), track(),
            speed(), moves(), [speed()]) -> {moves(), [speed()]}.
rally(_MaxA, _MaxB, [], _CurrentSpeed, Moves, Acc) ->
    {Moves, Acc};
rally(_MaxA, _MaxB, [[0,0]], _CurrentSpeed, Moves, Acc) ->
    {Moves+1, Acc};
rally(MaxA, MaxB, Track = [[_N,SpeedLimit] | _SubTrack],
       _CurrentSpeed, Moves, Acc) ->
    HighestSpeed = _CurrentSpeed + MaxA,
    SpeedCandidate =
      case HighestSpeed >= SpeedLimit of
      true ->
        SpeedLimit;
      false ->
        HighestSpeed
      end,
    {NewTrack, NewSpeed} =
      optimal_speed(Track, MaxB, SpeedCandidate),
    NewAcc = Acc ++ [NewSpeed],
    rally(MaxA, MaxB, NewTrack, NewSpeed, Moves+1, NewAcc).

-spec optimal_speed(track(), speed(), speed()) ->
    {track(), speed()}.
optimal_speed(Track, MaxB, Speed) ->
    NumOfMoves = Speed div 10,
    case move(Track, NumOfMoves, Speed) of
    overlimit ->
        optimal_speed(Track, MaxB, Speed-10);
    SubTrack ->
        case braking(SubTrack, MaxB, Speed) of
        true ->
            {SubTrack, Speed};
        false ->
            optimal_speed(Track, MaxB, Speed-10)
        end
    end
end.
-spec braking(track(), speed(), speed()) -> boolean.
  braking([], _MaxB, _Speed) ->
    true;
  braking(Track, MaxB, Speed) ->
    SpeedBraking = Speed - MaxB,
    case SpeedBraking =< 10 of
      true ->
        true;
      false ->
        NumOfMovesBraking = SpeedBraking div 10,
        case move(Track, NumOfMovesBraking, SpeedBraking) of
          overlimit ->
            false;
          SubTrack ->
            braking(SubTrack, MaxB, SpeedBraking)
        end
      end
    end.

-spec move(track(), units(), speed()) -> track() | overlimit.
  move([(0, 0)], _NumOfMoves, _Speed) ->
    [];
  move([(N,V) | SubTrack], NumOfMoves, Speed) ->
    case Speed =< V of
      true ->
        if N =:= NumOfMoves ->
          SubTrack;
        N > NumOfMoves ->
          [(N-NumOfMoves, V) | SubTrack];
        true ->
          MoveLeft = NumOfMoves-N,
          move(SubTrack, MoveLeft, Speed)
        end;
      false ->
        overlimit
    end.