Code Coverage in Functional Concurrent Languages

Shayan Najd Javadipour

Fall 2011

Abstract

Code coverage is a measure indicating how much of the actual code is exercised in the process of testing. However, code coverage metrics are mostly defined with the focus on imperative languages. When the target is a functional language, to be more effective, these metrics should be redefined.

In this project, we studied existing code coverage metrics and then we defined an equivalent criterion specialized for functional languages. Based on this metric, we developed a tool, named *FCover*, measuring code coverage in a functional concurrent language, namely Erlang. We designed the tool in a way that its output can be used to define a stopping function for automated property based test case generation.

FCover first extracts the abstract syntax tree (AST) of the input Erlang code and then it generates an instrumented code by transforming the AST to a semantically equivalent AST. This instrumentation is used to log the data useful for measuring the code coverage. In the last step, the logged data is post-processed to generate the final coverage information.

In this report, first we describe the existing concepts related to code coverage and at the end of each section, we discuss our interpretation and implementation. Later in the second chapter, we discuss our approach for measuring code coverage in Erlang code including the necessary code transformations for the instrumentation. We also describe the application of this coverage information in automated property based test case generation.

Contents

Chapter 1

Code Coverage in Theory

Code Coverage

Code coverage is a measure indicating how much of the actual code is exercised in the process of testing. Code coverage criteria are used as a measure of test adequacy.[11]

In this project, we studied existing code coverage metrics and then we defined an equivalent criterion specialized for functional languages. This equivalent criterion arguably provides a stronger notion of code coverage and it is comparably efficient. In the following sections, we discuss this equivalent metric in more detail.

Statement Coverage

Statement coverage—also closely named *node coverage*, *line coverage*, *segment coverage* [8], *basic block coverage* and *C1* [3]—is defined as:

"A set P of execution paths, satisfies the statement coverage criterion if and only if for all nodes n in the flow graph, there is at least one path p in P such that node n is on the path p."[11]

There are several different interpretations for "node in the flow graph" in the above mentioned definition such as:

1. Statements (statement coverage)
2. A single line of code (line coverage)
3. A single basic block; block of code with no branching (basic block coverage)
4. Expressions
5. etc

Often, control flow graphs (CFG) only model the execution flow of programs without considering the exception flows. Exception flow is an execution flow initiated by exceptions, errors or exit signals. For example, the term *throw "error"* branches the main execution flow by generating an exception. A complete CFG includes both the normal execution flows and the exception flows. Based on this definition of the CFG, we can redefine node coverage which is arguably more precise.

In this project, we define a special version of node coverage targeting functional languages. Since almost everything in a functional language is an expression, comprehensive definition of “node in the flow graph” would be “expression in a program”.

The CFG starts with the *source node* and ends with the *sink node*. Neither of them have any semantical operation.

For representing the complete CFG in a functional program, the CFG should include exception flows. For this purpose, we add one extra virtual node, a fork, exactly before each node and it has an edge to the exception node. This virtual node represents the branching when an exception is generated. The exception node is a normal node representing the execution flow of the program when the exception is generated. It has an edge to the *sink node* or to the initial node of an exception handling structure.

For example, the expression $1/X$ is represented as three nodes in the CFG:

1. Virtual node: a fork with the condition $X \neq 0$. It has an edge labeled $True$ to the normal node and an edge labeled $False$ to the exception node.

2. Normal node: a node that represents the expression when there is no exception. It has an edge to the node that represents the next expression in the execution flow

3. Exception node: is a node in the exception flow that represents the generation of the exception; for example, *throw* $"Divisionbyzero"$. It has an edge to either *sink node* or to the initial node of the exception handling structure (EHS)

Here we can define two types of node coverage based on the complete graph. The first one considers the exception node as a normal node that should be covered in testing and the weaker form that does not include exception nodes in the coverage measurements. The virtual nodes are ignored in both types. Our implementation is of the first type, covering all the nodes including the exception nodes.

In a functional language, every expression should return a value and there is no looping. Therefore, the CFG of a functional language is acyclic. In fact, there is looping in functional languages. For example, recursion is a way to simulate looping. But the main point is that in unit testing we focus on one single function. In that function, we consider function calls—even self calls—as abstract expressions. These abstract expressions either terminate and return a value, or do not terminate at all. In the case of nontermination

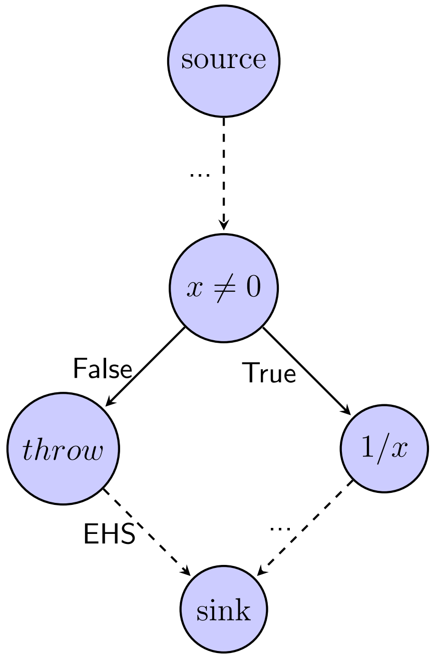

Figure 1.1: a CFG with virtual node

we can consider a new exception flow but usually they can be merged with the existing exception flow. If we merge the two, a nontermination is equivalent to the case that the callee returns an exception.

We also implemented some form of optimization. We cut the CFG on the specific cut-points and that would break the CFG into smaller parts; the distinct set of connected nodes between the two consecutive cut-points. Each of these smaller parts form a tree[1].

In a complete CFG, we define cut-points by:

1. Immediately after the *source node* is a cut-point

2. Immediately before the *sink node* is a cut-point

3. Immediately after the first node in the beginning of each branch, including the exceptional branches[2]

To achieve full node coverage, we need to make sure that all the leaves of these smaller parts are visited. That is because each leaf in these trees can uniquely define a path in the tree and when the execution flow reaches the beginning of the path, all the expressions in the path are executed.

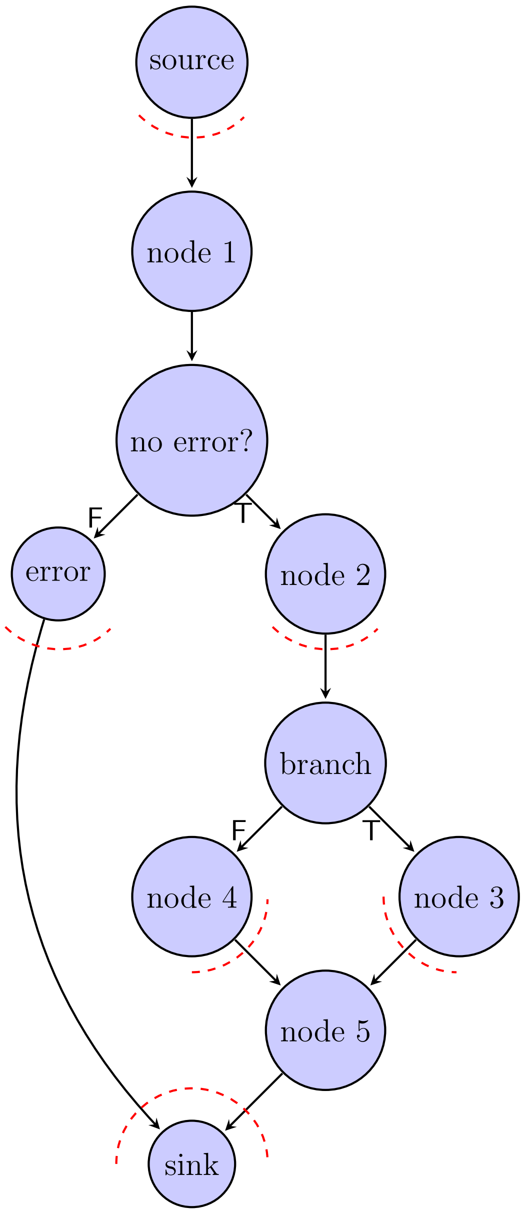

Figure 1.2: a sample CFG and cut-points

In summary, in our implementation we monitor the execution of:

1. the exception nodes

2. the fist node after the *source node*

3. the nodes at the beginning of each branch

Decision Coverage

Along with *all-edges coverage*, *C2* and *decision-decision-path testing*; *decision coverage* and *branch coverage* are closely related code coverage metrics.

Branch coverage is defined as:

"A set P of execution paths satisfies the branch coverage criterion if and only if for all edges e in the flow graph,there is at least one path p in P such that p contains the edge e."[11]

Decision coverage is defined as:

"Decision Coverage - Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once."[9]

And *decision* is defined as:

"A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition."[9]

Where *condition* is:

"A Boolean expression containing no Boolean operators."[9]

In functional languages, beside Boolean expressions[3], pattern matchings are used to form conditional clauses. To have a more comprehensive coverage metric, patterns should be considered a form of Boolean expressions. In Fcover, we transform a pattern p of variable X to a Boolean expression $match(p, X)$. Since patterns are not often first-class entities in functional languages, *match* is a meta-level function.

In the definition of decision coverage all the Boolean expressions are included while branch coverage considers only the ones immediately before each fork.

---

[1]Binary tree, since decisions are Boolean

[2]Each exceptional branch has only one node which is the exception node.

[3]They are often called guards.

The complete CFG of a functional program has a special property: if you visit all the nodes you have visited all the edges. Therefore, for such CFG, node coverage and branch coverage are equivalent.

FCover provides the specified node coverage; consequently, it also provides branch coverage.

In FCover, we add a monitor on top of the Boolean expressions in each conditional clause. Monitors, or loggers, observe and log the value of the Boolean expression at runtime. So later on, after preprocessing, we can check from the logged data whether all of these Boolean expressions were evaluated to both $True$ and $False$. However, there are difficulties in implementing decision coverage for dynamic type languages since "Boolean expression" is not defined. For Erlang, in our view, it is satisfactory to cover only the Boolean expressions inside control flow statements.

Condition Coverage

Condition coverage, also closely referred to as *predicate coverage*, is defined as follows:

"Condition coverage requires that each condition in a decision take on all possible outcomes at least once (...), but does not require that the decision take on all possible outcomes at least once. In this case, for the decision (A or B) test cases (TF) and (FT) meet the coverage criterion, but do not cause the decision to take on all possible outcomes. As with decision coverage, a minimum of two tests cases is required for each decision."[7]

As mentioned in the previous section, patterns should be considered a form of Boolean expression. Each sub-pattern of a pattern forms a separate condition, in a similar way that subexpressions of a Boolean expression do.

In this work, we decompose a decision into conditions and observe their values at execution time. Condition coverage can be computed by post-processing this logged data. It is done by comparing the logged values of each condition with the expected permutations. However, due to the difficulties mentioned in the previous section, we limit ourselves to the conditions in each conditional clause.

Multiple Condition Coverage

Multiple condition coverage is defined as follows:

"A test set T is said to be adequate according to the multiple-condition-coverage criterion if, for every condition C, which consists of atomic predicates $(p1, p2, ..., pn)$, and all the possible combinations $(b1, b2, ..., bn)$ of their truth values, there is at least one test case in T such that the value of pi equals bi, $i = 1, 2, ..., n$." [11]

In our terminology, a "condition" is a decision and "predicate" is a condition.

Multiple condition coverage grows exponentially in respect to the number of conditions in a decision, but it is easy to calculate. In practice, the number of conditions in a decision is often so low that it does not affect performance.

In this project, we calculate this metric by post-processing the logged data. It is done by verifying, for each decision, whether all the possible combinations in the truth table exist. This coverage metric is also extended to include patterns.

Condition / Decision Coverage

C/D Coverage is a mixture of condition coverage and decision coverage. It is defined as:

"Condition/decision coverage combines the requirements for decision coverage with those for condition coverage. That is, there must be sufficient test cases to toggle the decision outcome between $True$ and $False$ and to toggle each condition value between $True$ and $False$. Hence, a minimum of two test cases are necessary for each decision. Using the example $(AorB)$, test cases (TT) and (FF) would meet the coverage requirement. However, these two tests do not distinguish the correct expression $(AorB)$ from the expression A or from the expression B or from the expression $(AandB)$."[7]

In FCover, C/D coverage is done by post-processing the logged data in the same way as decision coverage and condition coverage. This coverage metric is also extended to include patterns.

Modified Condition / Decision Coverage (MCDC)

This criterion is part of the standard "DO-178B", and states:

"Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions."[9]

There are three important points in the implementation of MC/DC coverage:

1. How to handle shortcut logical operators
2. How to handle multiple/nested conditional control flows
3. How to handle repeated condition in a decision, also tautologies/contradictions

Moreover, this method should be extended to include patterns.

There are different ways to handle shortcut logical operators in functional programs:

1. Treating all the shortcut operators the same as their non-shortcut equivalents
2. Considering separate internal decisions for the operands[10]
3. Keeping conditions constant if they are not executed due to a shortcut operator[5]

Since decisions have no side effects in Erlang and errors/exceptions are equivalent to the Boolean value $False$, we can ignore the shortcut behavior of the operators and compute coverage assuming they are normal Boolean operators.

For measuring the MCDC coverage, in this project, an exhaustive algorithm is implemented to calculate whether a test suite passes the coverage. The input of the algorithm is the logged data. In this project we treat shortcut operators as normal operators and in case of chronological dependency, we interpret "logically undefined" as logical $False$. A condition named *child* is chronologically dependent on the condition named *parent* where *child* is meaningless if *parent* does not hold. For example, "$element(1, X)$" in the following code requires "$is\_tuple(X)$" to hold:

```
foo(X)->
    if is_tuple(X) andalso element(1,X)==2 -> ok;
        true -> false
end.
```

In case of multiple and nested control flow statements, the problem is how to include contexts in calculations. By contexts, we can determine whether a condition is repeated and in that case, what is its value. Having this information, we can compute coverage treating the repeated condition as a

Boolean constant with the value extracted from the context (environment). A simpler solution is to consider each nested decision as a separate decision with extra conditions indicating their contextual preconditions. For example, in “if” with multiple guards, the second guard is evaluated only if the first guard does not hold. In that case, we add negation of the first guard condition with shortcut operator “andAlso” to the decision of the second guard.

It is possible that conditions are not completely independent from each other and there are also chances of repeated conditions. Two solutions to this problem are defined as follows:

“Unique Cause MCDC: A Form of MCDC which allows for masking to be used only in the case of coupled conditions to show a conditions independence. Otherwise, only the condition of interest is allowed to change between the two truth vectors of the independence pair. The conditions change is (generally) the unique cause of the change in the expressions outcome, hence the name.”[4]

“Masking MCDC: A form of MCDC that allows all possible forms of masking to be used to show a conditions independence.”[4]

Where “masking” is defined as:

“The process of setting the RHS (LHS) operand of an operator to a value such that changing the LHS (RHS) operand of that operator does not change the value of the operator. For an AND operator, masking of the RHS (LHS) can be achieved by holding the LHS (RHS) False. Recall from Boolean algebra that $XANDFalse = FalseANDX = False$ no matter what the value of X is. For an OR operator, masking of the RHS (LHS) can be achieved by holding the LHS (RHS) True. Recall from Boolean algebra that $XORTrue = TrueORX = True$ no matter what the value of X is.” [4]

In this project, after observing the values of the conditions in each decision at the execution time, we compute MCDC coverage by post-processing the logged data. With our algorithm we can find conditions that do not even have one pair of test cases to determine their independence. Hence, we can identify tautologies; contradictions, decision independent from conditions; and dummy conditions, conditions that their value does not change decisions. There are several optimized algorithms in the literature that can generate test cases for the complete MCDC coverage.

Path Coverage

Path coverage is a complex, yet powerful, form of code coverage. Because of the implementation complexities and practical difficulties, there are several weaker variations of this coverage that are more useful in practice.

Path Coverage is defined as follows:

"A set P of execution paths satisfies the path coverage criterion if and only if P contains all execution paths from the begin node to the end node in the flow graph."[11]

Path coverage is not implemented in this project due to its complexity and inefficiency.

Bellow follows a list of some of the coverage metrics based on path coverage:

Basis Path Coverage

In this metric, execution paths that are in the same class and only differ in the number of loops (recursions), are identified. Then, complete basis path coverage demands testing at least one path in each class.

JJ-Path Coverage (LCSAJ Coverage)

This criterion determines if all jump to jump paths (LCSAJs) have been executed.

LCSAJ can be defined as:

"An LCSAJ consists of a body of code through which the flow of control may proceed sequentially and which is terminated by a jump in the control flow. The hierarchy $TERi$, $i = 1, 2, ..., n, ...$ of criteria starts with statement coverage as the lowest level, followed by branch coverage as the next lowest level."[11]

Data Flow Coverage

It is a form of path coverage that only includes the sub-paths from variable bindings to the subsequent references of the variables.

Predicate Coverage

It is defined as: "We say that predicate coverage has been achieved if all possible combinations of truth values corresponding to the selected path have been explored under some test. Predicate coverage is clearly stronger than branch coverage. If all possible combinations of all predicates under all interpretations are covered, we have the equivalent of total path testing. Just as there are hierarchies of path testing based on path segment link lengths, we can construct hierarchies based on different notions of predicate coverage."[3]

All-Definition Coverage

The definition is as follows: "A set P of execution paths satisfies the all-definitions criterion if and only if for all definition occurrences of a variable x such that there is a use of x which is feasibly reachable from the definition, there is at least one path p in P such that p includes a subpath through which the definition of x reaches some use occurrence of x."[11]

All-Uses Coverage

All-Uses Coverage is described as: "A set P of execution paths satisfies the all-uses criterion if and only if for all definition occurrences of a variable x and all use occurrences of x that the definition feasibly reaches, there is at least one path p in P such that p includes a subpath through which that definition reaches the use." [11]

All Definition-Use-Paths (All DU-Paths)

The definition is as follows: "A set P of execution paths satisfies the all-du-paths criterion if and only if for all definitions of a variable x and all paths q through which that definition reaches a use of x, there is at least one path p in P such that q is a subpath of p, and q is cycle-free or contains only simple cycles." [11]

N-Length Sub-path Coverage

N-Length Sub-path Coverage determines whether each path with length N is executed.

Required k-Tuples Criteria

It is defined as: "A set P of execution paths satisfies the required k-tuples criterion, $k > 1$, if and only if for all j-dr interactions L, $1 < j \leq k$, there is at least one path p in P such that p includes a subpath which is an interaction path for L."[11]

Other Code Coverage Metrics

Function Coverage

If all the functions or subroutines in the program are called, we achieve complete function coverage. In this project we implement the general form of it, Entry/Exit Coverage.

Entry/Exit Coverage

If all the functions (or subroutines) in the program are called (entry) and return a value (exit), complete entry/exit coverage is achieved. We add two observation points (logging points) to each function, one in the beginning and one in the exit. After execution of the test cases we can measure the coverage based on the logged data.

Call Coverage / Call Pair Coverage

With the assumption that most bugs lie in the interfaces between code blocks, this method determines whether all the function calls have been executed. In this project for each function call there is an exception node and normal node that should be covered. If there is no exception indicating that function cannot be called, 100% call coverage is achieved, otherwise each of these exceptions reveal a function-call bug.

Loop coverage (recursion coverage)

This coverage metric determines whether each loop in the program has been executed zero, one or multiple times. In functional programs we can replace this coverage metric with "Recursion Coverage". Recursion coverage determines whether recursion happened zero, one or multiple times. In this project the recursion coverage can be achieved by searching the logged data to check if there are zero, one or multiple records of function-entry for the same function name.

Relational Operator Coverage

This metric measures whether every expression with comparison operators is tested with its boundary values. This metric is not yet implemented in the tool.

Table Coverage

Table coverage indicates whether each entry in a particular array has been referenced.[2] This metric is not implemented in the tool.

Race Coverage

Monitors the code at execution time and reports which parts of the code have been executed with multiple threads (processes). This information helps discovering race conditions. This tool does not support this metric.

Object Code Related Coverages

There are several metrics to check the quality of test cases at the object code level, in the compiler or directly after compilation of the code. Since the tool works with high-level code, these metrics are not implemented.

Fault Based Coverages

There are several metrics determining the quality of test suites and their code coverage by adding some forms of error inside the code. This tool does not support these metrics.

Chapter 2

Our Approach

Coverage Information

In FCover, raw coverage information is a list of program points and their corresponding information. Since later on we use the tool as a stop function, we collect special information for the different categories of program points to be able to identify identical test cases. Program points are relative positions inside a function tracked globally using a central state counter. During the AST transformation, the transformer requests a new program point from the counter and assigns it to the logger to embed it inside the augmented code. For example:

```
fooFunc () ->
        bar.
```

Has multiple program points including:

1. Function Entry
2. Function Exit
3. etc

A path would be an ordered list of these logged tuples. For the above mentioned example, it would be something like:

```
[ {ProcessID# , fooFunc, programPoint 1
     ,Line 2,{functionEntry}}
   ,...
   ,{ProcessID# , fooFunc, programPoint n
     ,Line 2,{functionExit}}]
```

The first part of each element is the *PID* of the process executing the test case. The second part is the name of the function and the third is the relative program point in the function. The line number of the original code is in the fourth part. The fifth part contains some detailed information about each program point.

Code Transformation

To be able to log required and useful information, we inject loggers to different parts of the original code. We transform the code, via AST transformation, into a semantically equivalent instrumented version. In the following sections, we list and discuss these transformations.

Enclosing Errors

In the section about statement coverage, we mentioned that our interpretation of node in the complete control flow graph includes branched execution flows due to errors and exceptions. To log information related to exceptions we wrap each error producing, or exception producing, language construct inside a block implemented by try-catch to log the exception information. Below follows a list of possible expressions[1] and indication of whether they generate errors at runtime if the arguments do not return exceptions (errors):

<table>
<tr><th>Expression Type</th><th>Possibility of Run-Time Error</th></tr>
<tr><td>literal</td><td>No</td></tr>
<tr><td>P = E</td><td>Yes</td></tr>
<tr><td>variable V</td><td>No</td></tr>
<tr><td>tuple skeleton {E_1, ..., E_k}</td><td>No</td></tr>
<tr><td>empty list []</td><td>No</td></tr>
<tr><td>cons skeleton [E_h | E_t]</td><td>No</td></tr>
<tr><td>binary constructor <<V_1:Size_1/TSL_1, ..., V_k:Size_k/TSL_k>></td><td>Yes</td></tr>
<tr><td>E_1 Op E_2, where Op is a binary operator</td><td>Yes</td></tr>
<tr><td>Op E_0, where Op is a unary operator</td><td>Yes</td></tr>
<tr><td>#Name{Field_1=E_1, ..., Field_k=E_k}</td><td>No</td></tr>
<tr><td>E_0#Name{Field_1=E_1, ..., Field_k=E_k}</td><td>Yes</td></tr>
<tr><td>#Name.Field</td><td>No</td></tr>
<tr><td>E_0#Name.Field</td><td>Yes</td></tr>
<tr><td>catch E_0</td><td>No</td></tr>
<tr><td>E_0(E_1, ..., E_k)</td><td>No</td></tr>
<tr><td>E_m:E_0(E_1, ..., E_k)</td><td>Yes</td></tr>
<tr><td>list comprehension [E_0 || W_1, ..., W_k], where each W_i is a generator or a filter</td><td>Yes</td></tr>
<tr><td>binary comprehension <<E_0 || W_1, ..., W_k>>, where each W_i is a generator or a filter</td><td>Yes</td></tr>
<tr><td>begin B end, where B is a body</td><td>No</td></tr>
<tr><td>if Ic_1 ; ... ; Ic_k end, where each Ic_i is an if clause</td><td>Yes</td></tr>
<tr><td>case E_0 of Cc_1 ; ... ; Cc_k end, where E_0 is an expression and each Cc_i is a case clause</td><td>Yes</td></tr>
</table>

try B catch Tc_1 ; ... ; Tc_k end, where B is a body and each Tc_i is a catch clause	No
try B of Cc_1 ; ... ; Cc_k catch Tc_1 ; ... ; Tc_n end, where B is a body, each Cc_i is a case clause and each Tc_j is a catch clause	Yes
try B after A end, where B and A are bodies	No
try B of Cc_1 ; ... ; Cc_k after A end, where B and A are a bodies and each Cc_i is a case clause	Yes
try B catch Tc_1 ; ... ; Tc_k after A end, where B and A are bodies and each Tc_i is a catch clause	No
try B of Cc_1 ; ... ; Cc_k catch Tc_1 ; ... ; Tc_n after A end, where B and A are a bodies, each Cc_i is a case clause and each Tc_j is a catch clause	Yes
receive Cc_1 ; ... ; Cc_k end, where each Cc_i is a case clause	No
receive Cc_1 ; ... ; Cc_k after E_0 -> B_t end, where each Cc_i is a case clause, E_0 is an expression and B_t is a body	Yes
fun Name / Arity	No
fun Module:Name/Arity	No
fun Fc_1 ; ... ; Fc_k end where each Fc_i is a function clause	No
query [E_0 || W_1, ..., W_k] end, where each W_i is a generator or a filter	We do not support
E_0.Field, a Mnesia record access inside a query	We do not support
parenthesized expressions (E_0)	No

To wrap all the above mentioned constructs which may raise an exception, we use the same code block. An example of transformed code for expression $1 * 9$ at line 3 of function f would be:

```
try
    1 * 9
catch
    exit:VarUnique2 ->
        rareLoggerName !
          {self(), f, 2, 3, {exception}},
        exit(VarUnique2);
    error:VarUnique2 ->
        rareLoggerName !
          {self(), f, 2, 3, {exception}},
        erlang:error(VarUnique2);
    VarUnique2 ->
        rareLoggerName !
          {self(), f, 2, 3, {exception}},
        throw(VarUnique2)
```

```
end.
```

The catch part has three cases, one for each possible category of exceptions. It first executes the expression and if there is no exception, it returns the value. If exceptions are generated, it catches them, logs them and re-throws them again. As mentioned before there is a central logger server named “rareLoggerName” that logs messages sent from different parts of the instrumented code. Here, the recorded message is “exception”.

Function/Abstraction Transformation

Functions are first transformed to a set of case clauses representing their pattern matching behaviors and then the case clause transformation is applied on the result of the first transformation. Also in the first transformation, we have to design a wildcard pattern to return the “match_failure” error specific to the function instead of the case specific one. There are loggers at the beginning and at the end of the function body to log function entry and function exit. For example, function definition:

```
f([X])->X.
```

is first transformed into:

```
f(XUnique1) ->
    begin
        ExpUnique_2 = {XUnique1},
         case ExpUnique_2 of
          {X} ->  X;
          _ -> erlang:error(function_clause)
        end
      end.
```

and then in the second stage, the case expression is transformed. In the first stage, the arguments of the function are all wrapped in a tuple, over which the pattern matching happens and a special clause is added to generate “function match_failure” error instead of “case match_failure”.

If Transformation

Loggers are placed on top of the “if” structure to observe the value of the conditions in decisions. For example:

```
if
  X < 1 -> v1;
```

```
  X > 1 -> v2
end
```

is transformed into an equivalent code of this pseudo-code:

```
begin
  loggerForIfClause,
  loggerForIfClause,
  if
    X < 1 ->
            loggerForClauseEntry
            , v1;
    X > 1 ->
            loggerForClauseEntry
            , v2
  end
end
```

The first logged message includes textual representation of the decision and its actual value. There is also another logger, in the beginning of the clause, logging clause entry. It is possible to have exceptions in the decision part of the control flow statements; an error case in Erlang is equal to *False*. Therefore, the actual transformation of:

```
if 2 > 1 -> 1 end
```

would be:

```
begin
  rareLoggerName !
    {self(), f, 2, 3,
        {"try 2 > 1 catch _:_ -> false end"
          , try
                2 > 1
            catch
                _:_ -> false
            end}},
  if 2 > 1 ->
      rareLoggerName ! {self(), f, 3,
                3, {clauseEntry}}
      , 1
  end
end
```

Case Transformation

For each case clause there is a logger on top of the case block to log the conditions in the decisions of each clause.

For example:

```
case 1 of
  1 -> v1
  2 -> v2
end
```

is transformed into the equivalent code of this pseudo-code [1]:

```
begin
        ExpUnique_2 = 1,
        case ExpUnique_2 of
          1 ->
               loggerForCaseClause
          _ ->
              loggerForCaseClauseNegative
        end,
        case ExpUnique_2 of
          2 ->
               loggerForCaseClause
          _ ->
              loggerForCaseClauseNegative
        end,
        case ExpUnique_2 of
          1 ->
              loggerForEntry
              , v1
          2 ->
              loggerForEntry
               ,v2
        end
  end
```

Catch Transformation

Catch clauses are translated into nested catch expressions, in order to log the proper information. For example:

[1] and also enclosed in try/catch

```
try
  1
catch
  A -> 2  ;
  B -> 3
end.
```

The catch part is transformed into the equivalent code of this pseudo-code:

```
try
  1
catch
  error:UniqueX -> LogClauseHere
                 , try
                     erlang:error(A)
                   catch
                     A-> 2;
                     B->3
                   end;
  exit:UniqueX  -> LogClauseHere
                 , try
                     erlang:exit(A)
                   catch
                     A-> 2;
                     B->3
                   end;
  UniqueX       -> LogClauseHere
                 , try
                     erlang:throw(A)
                   catch
                     A-> 2;
                     B->3
                   end;
end.
```

Try Transformation

There are several different forms of “try” expressions in Erlang. The transformation of “try” block, which includes “of” selection, is a complex process. For example:

```
try exp1 of
  [] -> a;
  _ -> b
catch
   _:_ -> c
end
```

is transformed into the equivalent code of this pseudo-code[2]:

```
case  ( try
           {ok, begin exp1 end }
        catch
           transformed catch with its return
            value enclosed in tuple {error, ?}
        end) of
  {ok, V } -> case clauses representing try
  "of_selection" clauses
  {error, E } -> E
end
```

Testing Approach: Stop Function

We write tests in terms of the properties that are used by Quickcheck[6] to automatically generate test cases. In automatic test generation, we need a stop function to determine when the generated test cases are satisfactory enough and to stop the process of the test case generation afterwards. A stop function needs to identify which test cases are equivalent and it is done by defining features. Two test cases are equivalent if they have the same features. Quickcheck can do feature based testing for us. For this purpose, we connect FCover to Quickcheck. Features are determined by the coverage information.

Below follows the process:

1. FCover parses the code and generates the Abstract Syntax Tree (AST).

2. The AST is traversed by the tool and transformed to a new instrumented AST.

3. The tool compiles and loads the new AST.

4. Quickcheck generates and executes a test case.

[2]with another stage to transform the cases in generated code

5. FCover calculates the raw coverage information and passes it to Quickcheck.
6. Quickcheck keeps repeating the last two steps by feature based approach until it stops and returns the test suite.
7. To shrink the test suite, coverage information of the generated test suite is post-processed to identify and remove the equivalent test cases.

Bibliography

[1] Erlang Run-Time System Application (ERTS) User's Guide-Version 5.8.5. The abstract format.

[2] J. Andersson. Automatic test vector generation and coverage analysis in model-based software development. 2005.

[3] B. Beizer. *Software testing techniques*. Dreamtech Press, 2002.

[4] JJ Chilenski. An investigation of three forms of the modified condition decision coverage criterion. Technical report, Report DOT/FAA/AR-01/18, Federal Aviation Administration, USA, 2001.

[5] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[6] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Acm sigplan notices*, volume 35, pages 268–279. ACM, 2000.

[7] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.

[8] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14:868–874, June 1988.

[9] Position Paper CAST-10 (Certification Authorities Software Team). What is a decision in application of modified condition/decision coverage (mc/dc) and decision coverage (dc)?, June 2002.

[10] SC-190/EROCAE WG-52. Final annual report for clarification of do-178b software considerations in airborne systems and equipment certification, October 2001.

[11] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29:366–427, December 1997.