Erlang Code Evolution Control

David Insa¹, Sergio Pérez¹, Josep Silva¹, Salvador Tamarit²

¹Universitat Politècnica de València
²Universidad Politécnica de Madrid

LOPSTR 2017 12/10/2017
Content

• Introduction
• The technique in detail
  • Type Analysis Phase
  • Test Case Generation Phase
  • Comparison Phase
• Recording the trace
• SecEr tool
• Conclusions & Future Work
Introduction

• In **debugging** programmers use **breakpoints** to observe the values of an expression during an execution

• Is this feature available in **testing**?

• It would be useful to focus the test cases on an **specific point** without code modifications

**OUR PROPOSAL**

- Introduce the ability to specify **Points Of Interest (POI)** in the context of testing

- A technique to **compare two equivalent POIs** in different versions of the same program for Erlang
Introduction

**Old Version**

\[ \text{main}(X,Y) \rightarrow \]
\[ A = X + Y, \]
\[ D = X - Y, \]
\[ A * D. \]

**New Version**

\[ \text{main}(X,Y) \rightarrow \]
\[ A = \text{add}(X,Y), \]
\[ D = \text{sub}(X,Y), \]
\[ A * D. \]

1. **Identify a POI** and a set of **input functions**
2. A **test suite** is automatically **generated**
   
   Each test case contains:
   - A **call** to an input function with specific arguments
   - The **sequence of values** the POI is evaluated to (trace)
3. Each test case is passed against the new version and both traces are compared
4. A **report** of the **success or failure** of the test cases is provided

**We have implemented our approach for Erlang in a tool named SecEr**
Introduction

Old Version

\[ \text{main}(X,Y) \rightarrow A = X + Y, B = X - Y, A \times B. \]

New Version

\[ \text{main}(X,Y) \rightarrow A = \text{add}(X,Y), B = \text{sub}(X,Y), A \times B. \]

1. **Identify** a POI and a set of **input functions**

2. **A test suite** is automatically **generated**

   Each test case contains:
   - A **call** to an input function with specific arguments
   - The **sequence** of values the POI is evaluated to (trace)

3. **Each test case is passed against the new version** and both traces are **compared**

4. **A report** of the **success or failure** of the test cases is provided

*We have implemented our approach for Erlang in a tool named *SecEr*"
Content

• Introduction
• The technique in detail
  • Type Analysis Phase
  • Test Case Generation Phase
  • Comparison Phase
• Recording the trace
• SecEr tool
• Conclusions & Future Work
The technique in detail

Type Analysis Phase → Test Case Generation Phase → Comparison Phase → Final Report

Input → Test Case
The technique in detail
Type analysis phase
Type analysis phase

1.- The inferred types refer to the whole function

Solution: Consider each clause independently and refine the result to types per clause

\[ g(\emptyset, \emptyset) \rightarrow \ldots \quad \text{“-spec } g(\emptyset|1, \emptyset|1) \rightarrow \ldots” \]

\[ f(\emptyset, [A, B]) \rightarrow \text{“-spec } f(1|2, [1|2|5|6]) \rightarrow \ldots” \]

2.- The length of the list is unknown

\[ f(1, [1, 2, 5, 6]) \rightarrow f(1, [1, 2, 5, 6]) \]

3.- The repeated-variable restriction is ignored

\[ f(1, [2, 5]) \rightarrow f(1, [2, 5]) \]
prop_identity() ->
  ?FORALL(X, any(), id(X) = X).

id(X) -> X.
2.- The length of the list is unknown
   Solution: Traverse the list parameters of the clause element by element

3.- Repeated variable relation is lost
   Solution: Store the values of already treated variables
Type analysis phase

1. Input
   → .erl
   → Fun
   → TypEr + new code
   → PropEr + new code
   → Input
   → CutEr + new code
   → Inputs

2. Input
   → concolic gen
   → Inputs
   → CutEr Inputs
   → Inputs
   → Runtime Errors
The technique in detail

Type Analysis Phase → Test Case Generation Phase → Comparison Phase → Final Report
Test case generation phase

- Input
- input selector
- Input
- execute
- Trace
- New?
- T
- mutating gen
- New Input
- F
- random gen
- New Input
- add

Continue until a limit is reached
Store all the generated test cases (Input + Trace)
The technique in detail
Comparison phase

Test Case selector → Test case → execute → Trace2

.code instrumentator

Result → Final Report

Test case selector

Test case

Input

Trace1

.add /

Comparison Result

.Result analyzer

Test Case

.POI

.erl2

Result

Final Report
Content

- Introduction
- The technique in detail
  - Type Analysis Phase
  - Test Case Generation Phase
  - Comparison Phase
- Recording the trace
- SecEr tool
- Conclusions & Future Work
Recording the trace

• There are several tools for tracing executions in Erlang
• None of them allows us to collect the trace of patterns
• Debuggers will not provide a value for a POI if it is inside an expression whose evaluation fails

OUR PROPOSAL

➢ Collect the traces as a side effect when executing the code
➢ Approach based on message passing to a tracing server
  ➢ The code needs to be instrumented (4 STEPS)
Recording the trace (1 & 2)

1. Obtain and annotate the Abstract Syntax Tree of the program. Annotate each node with two lists of variables:
   - Variables being bound in its subtree
   - Variables that were already bound when reaching the node

2. Find the selected POI in the AST with a top-down traversal:
   - Store the current traversed path with tuples of the form (Node,ChildIndex)
   - The result is a path that yields directly to the POI
3. Analyze the location of the POI
   • Expressions: Add a send command to inform the tracing server
   • Patterns: Need special treatment

\[
\{1, B, 3\} = \{1, 2, 4\} \quad \text{2}
\]

Target expressions
   • Pattern-matching
   • List comprehension
   • Expressions with clauses:
     • if
     • case
     • functions
     • etc.
Recording the trace (3)

Divide the AST path into two sub-paths:

- **PathBefore**: Root -> deepest target expression
- **PathAfter**: First child of the target expression -> POI
Recording the trace (4)

4. Perform the actual instrumentation
   • Traverse the PathBefore
   • Transform the code following a rule according to PathAfter
   • Traverse PathBefore backwards to update the AST

Five exclusive rules to instrument expressions
   • LEFT_PM (pattern-matching)
   • PAT_GEN_LC (list comprehensions)
   • CLAUSE_PAT (pattern in expressions with clauses)
   • CLAUSE_GUARD (guard in expressions with clauses)
   • EXPR (expressions)
Recording the trace (4)

\[(\text{LEFT\_PM}) \quad p = e \rightarrow p = \begin{array}{l} np = e, \text{tracer!\{add, npoi\}, np end} \\
  \text{if} \\
  (p = e, _) = \text{last}(\text{PathBefore}) \\
  \land (_, \text{pos}(p)) = \text{hd}(\text{PathAfter}) \\
  \text{where} \\
  (_, \text{npoi}, np) = \text{pfv}(p, \text{PathAfter})\end{array}\]
Content

• Introduction
• The technique in detail
  • Type Analysis Phase
  • Test Case Generation Phase
  • Comparison Phase
• Recording the trace
• SecEr tool
• Conclusions & Future Work
SecEr command:
$ ./secer -f FILE -li LINE - var VARIABLE [-oc OCCURRENCE]
- f FILE -li LINE - var VARIABLE [-oc OCCURRENCE]
[-funs INPUT_FUNCTIONS] -to TIMEOUT

SecEr tool

Old_Version_POI

New_Version_POI
SecEr tool

$ ./secer -f happy0.erl -li 9 - var Happy -oc 1
-f happy1.erl -li 18 - var Happy -oc 1
-funs [main/2] -to 15
SecEr tool

$ ./secer -f happy0.erl -li 9 -var Happy -oc 1
   -f happy1.erl -li 18 -var Happy -oc 1
   -fun [main/2] -to 15

Function: main/2
---------------------------
Generated test cases: 320
Both versions of the program generate identical traces for the point of interest
---------------------------
SecEr tool

1 -spec main(pos_integer(), pos_integer()) ->
2   [pos_integer()].
3 main(N, M) ->
4   happy_list(N, M, []).
5
6 happy_list(_, N, L) when length(L) == N ->
7   lists:reverse(L);
8 happy_list(X, N, L) ->
9   Happy = is_happy(X),
10  if Happy ->
11     happy_list(X + 1, N, [X|L]);
12   true ->
13     happy_list(X + 1, N, L) end.
14
15 is_happy(1) -> true;
16 is_happy(4) -> false;
17 is_happy(N) when N > 0 ->
18   N As Digits =
19     [Y - 48 || Y <- integer_to_list(N)],
20   is_happy(
21     lists:foldl(
22       fun(X, Sum) ->
23       (X * X) + Sum
24     end,  
25     0,  
26     N As Digits));
27 is_happy(_) -> false.

1 is_happy(X, XS) ->
2   if
3     X == 1 -> true;
4     X < 10 -> false;
5     true ->
6       case member(X, XS) of
7         true -> false;
8         false ->
9           is_happy(sum(map(fun(Z) -> Z*Z end,
10             [Y - 48 || Y <- integer_to_list(X)])),
11             [X|XS])
12     end
13     end.
14 happy(X, Top, XS) ->
15   if
16     length(XS) == Top -> sort(XS);
17     true ->
18       Happy = is_happy(X,[]),
19       Happy of
20         true -> happy(X + 1, Top, [X|XS]);
21         false -> happy(X + 1, Top, XS)
22     end
23     end.
24
25 -spec main(pos_integer(), pos_integer()) ->
26   [pos_integer()].
27 main(N, M) ->
28   happy(N, M, []).
SecEr tool

```
$ ./secer -f happy0.erl -li 9 -var Happy -oc 1
   -f happy1.erl -li 18 -var Happy -oc 1
   -fun [main/2] -to 15

Function: main/2
----------------------
Generated test cases: 251
Mismatching test cases: 22 (8.76%)
All mismatching results were saved at: ./results/main_2.txt
--- First error detected ---
Call: main(4,1)
happy0 trace (9,Happy,1): [false,false,false,true]
happy1 trace (18,Happy,1): [false,false,false,false,false,false,false,true]
```
Content

• Introduction
• The technique in detail
  • Type Analysis Phase
  • Test Case Generation Phase
  • Comparison Phase
• Recording the trace
• SecEr tool
• Conclusions & Future Work
Conclusions

• Combination of Erlang existing tools and mutation to improve the result
• New approach to automatically check the behaviour preservation between versions

{1,B,3} = {1,2,4}

New tracing process that allows for placing a POI in patterns, guards and expressions

{1,POI,FV} = {1,2,4},
tracer ! POI,
{1,POI,FV}
end
Future Work

Future Work

• Adapt the approach to deal with indeterminism
• Increase the information stored in traces to report non-functional properties such as efficiency
• Allow for the specification of a list of POIs instead of a single POI
• Make the tool compatible with tests previously defined by the user
Thank you for your attention!!

Any question?