Static fault localization for simple bugs

(background work)

Muhammad Numair Mansur

Software Engineering Group
University of Freiburg, Germany

Supervision: Matthias Heizmann, Christian Schilling

1 Introduction

This document contains behind the scenes work of our submission in *POPL 2018*, namely "Static fault localization for simple bugs". It contains the previous/failed definitions of relevance that lead us to the final version together with examples that show why the previous definitions did not work. It also shows some examples for which the output of our algorithm is counterintuitive. The document should also serve as a reference for future work for us. This will also be the submission as my master project report.

2 Motivation

Identifying program statements that are the cause for the error is the most time consuming and tedious part of a programmer's debugging routine. This task can be made much simpler if it can be done automatically. The process of automatically finding program statements causing the error is called *fault localization*. A program might be failing because of several reasons which are not necessarily simple (a term elaborated more clearly in the paper) and easy to point out and there might be many ways to fix the bug. We, however, stick to what we call simple bugs. A bug in an error trace is called simple if the modification of a single statement can fix the error. There can be many techniques to help the user narrow down his or her search for the statements causing the error or *relevant* statements, as we call them in the paper. In this work, we present a new such technique that statically analyzes all the assigning statements in an error trace containing a simple bug and returns those, which when modified can fix this bug.
Not only did we want to help the user find simple bugs, we also wanted to formalize the notion of when an assigning statement is relevant in an error trace. Coming up with a precise formal definition took some iterations. All the previous definitions are also presented in this document along with the reasons of why they were discarded and the examples on which the inconsistency between them and our algorithm became clear.

3 Discarded definitions of relevance

We developed several definitions of relavance while trying to formally characterize the behaviour of our algorithm. We came across several versions that seemed correct at the time but in reality did not accurately and completely explain our notion of relevance and hence were discarded. We discovered this by coming up with counterexamples where these definitions did not work. This section mentions those discarded definitions and the counterexamples.
They are in the document primarily for the reason of documenting failed/incorrect versions of our current definition that might be useful in the future as we further develop our understanding of what it means that a statement is relevant

in an error trace for a simple bug.
For future reference it is easier if we name these definitions. Let us name these approaches with "check" in the end since we declare a statement relevant if they satisfy a localized check.

3.1 The restrictive-assume check

This was our first attempt to formally define a relevant assigning statement. At the time, we had two separate checks for relevance of an assignment statement and a havoc.

3.1.1 Assignment statement

An assignment statement was marked relevant, if after replacing that assignment with a havoc, some assume statement in the error trace was becoming restrictive.

Definition 1 (Restrictiveness of a statement)**.** *Let* pre *be a state formula,* π *a trace and* i *a position such that* $\pi[i]$ *is an assume statement. We call the assume statement* $\pi[i]$ restrictive *iff :*

$$sp(pre, \pi[0, i-1]) \not\Rightarrow guard(\pi[i])$$

Definition 2 (Relevance of an assignment statement)**.** *Let* π *be an error trace and* $\pi[i]$ *be an assignment statement at position* i *having the form* $x := t$*, where* x *is a variable and* t *is an expression. Let* π' *be the trace which is obtained by replacing* $\pi[i]$ *by* $havoc(x)$*. Let pre be the error precondition of* π*. The assignment statement* $\pi[i]$ *is* relevant *if there exists some assume statement at position* $j > i$ *in* π' *such that* $\pi'[j]$ *is restrictive.*

3.1.2 Havoc statement

Definition 3 (Relevance of a havoc statement)**.** *Let* π *be a feasible error trace and* $\pi[i]$ *be a Havoc statement at position* i *having the form* $havoc(x)$*, where* x *is a variable. Let* π' *be the trace which is obtained by replacing* $\pi[i]$ *by an assignment statement* $x := t$*, where* x *is the same variable as in havoc(x) and* t *is an expression. The havoc statement is* relevant *iff there exists an assignment x:=t such that* π' *is infeasible. That is:*

$$sp(true; \pi') \Rightarrow false$$

3.1.3 Examples

Where it works
Below is an example where the above check for relevance correctly marks a statement relevant.

```
1 foo()
2 {
3   x := 1;
```

```
4   y := 2;
5   z := 3;
6   assert(z > 10);
7 }
```

Lines 3 and 4 are not relevant as if we replace them with havoc, no assume in the error trace is becoming restrictive. But if we replace line 5 with $havoc(z)$, then the last (and only) assume statement in the error trace ($assume(z <= 10)$) is becoming restrictive. Hence the line with the assignment to z is relevant.

Where it fails

```
1 foo()
2 {
3   x := 1;
4   y := 2;
5   z := 3;
6   havoc z;
7   assert(z > 10);
8 }
```

In the above example, every statement is now relevant. If we replace any of the assigning statements with *havoc*, the assume statement in the end is restrictive and not necessarily because of the replacement with havoc but because of the last $havoc(z)$ statement in the program. Hence in this program line 3 and 4 are also relevant.

Another example where this approach fails is:

```
1 procedure main()
2 {
3   y := 42;
4   havoc x;
5   assume(x >= 0 && y >= 23);
6   assert(false);
7 }
```

Here replacing the assignment statement $y := 42$ with $havoc(y)$ has no effect on the restrictiveness of an already restrictive error trace. Hence it should not be relevant here. However, clearly this statement is also responsible for the bug.

3.2 The blocking-executions check

We adopted this definition for relevance when we realized that we should take into account the "amount" of restrictiveness of an assume statement instead of just considering if it is getting restrictive or not in a binary fashion.

Definition 4 (Execution)**.** *Let π be an error trace of length n. An* execution *of π is a sequence of states $s_0, s_1, ..., s_n$ such that $(s_i, s_{i+1}) \models T$, where T is the transition formula of $\pi[i]$.*
Let ξ represent the set of all possible executions of the error trace.

Definition 5 (Blocking Execution)**.** *An execution of a trace π of size n is called a* blocking execution *if there exists a sequence of states $s_0, s_1, ..., s_j$ where $i < j \leq n$ such that $(s_i, s_{i+1}) \models T[i]$, where $T[i]$ is the transition formula of*

$\pi[i]$ *and there exists an assume statement in the trace* π *at position* j *such that* $s_j \not\Rightarrow guard(\pi[j])$.

Definition 6 (Relevance)**.** *Let* $\mathcal{B}$ *represent the set of all blocking executions of a trace* π*. Let there be an assignment statement of the form* $x := t$ *at position* i*. Let* π' *represent the trace that we get after replacing* $\pi[i]$ *with a havoc statement of the form* $havoc(x)$ *and let* $\mathcal{B}'$ *represent the set of all blocking executions for* π'*.*
We say that the assignment statement $\pi[i]$ *is relevant if the trace after the replacement has strictly more blocking executions than the trace before the replacement in the following sense*

$$\mathcal{B} \subsetneq \mathcal{B}'$$

.

3.2.1 Examples

Where it works
Consider the failed example from the restrictive-assume check.

```
1 procedure foo()
2 {
3   y := 42;
4   havoc x;
5   assume(x >= 0 && y >= 23);
6   assert(false);
7 }
```

The blocking-executions check now correctly says that $y := 42$ is relevant since changing it to havoc gives us more blocking executions than before.
Where it fails

```
1 procedure foo()
2 {
3   y := 10;
4   havoc x;
5   assume(x > 0);
6 }
```

In this example, the statement $y := 10$ clearly has nothing to do with the error. But changing it to havoc gives us more blocking executions than before and according to the blocking-executions check, it is wrongly marked as relevant, too.

4 Algorithmic modification

We also noticed that we need a slight modification in the algorithm if we want to make it completley consistent with our notion of relevance. Consider the following example:

```
procedure foo()
{
  x := 5;
  y := 7;
  assume(x != 5 && y != 7)
}
```

with an error trace:

```
  x := 5     y := 7     assume (x == 5 || y == 7)
```

According to our final definition of relevance, $y := 7$ should not be relevant as no value v exists such that if we replace $y := 7$ with $y := v$, the rest of the trace (starting in the state $x = 5$) has a blocking execution.

But our previous algorithm marked the statement as relevant because P was computed as $pre(true, \pi[i, n])$, where $\pi[i]$ is the assignment statement under consideration. Hence P would be $true$ and Q would be $x = 5 \vee y = 7$. Obviously $P \not\subseteq wp(Q; havoc(y))$, where $wp(Q, havoc(y))$ is $x = 5$. Therefore, the assignment is marked relevant.
We fixed this error in the algorithm by computing P not as $pre(true, \pi[i, n])$ but as the intersection of $pre(true, \pi[i, n])$ and $sp(true, \pi[0, i-1])$.

5 Final version of Relevance

5.1 Introduction

An assigning statement ($x := t$ or $havoc(x)$) is responsible in an error trace if the assigned value matters for the reachabilty of the error. If the error state is reachable for any possible value of the variable x, then we say that the assignment to x at this location in the error trace is not relevant. Also, in the final version, we have one definition for both the assignment statement and havoc.

5.2 Formal definition

Definition 7 (Execution)**.** *Let π be an error trace of length n. An execution of π is a sequence of states $s_0, s_1, ..., s_n$ such that $(s_i, s_{i+1}) \models T$, where T is the transition formula of $\pi[i]$.*

Definition 8 (Blocking Execution)**.** *An execution of a trace π of size n is called a* blocking execution *if there exists a sequence of states $s_0, s_1, ..., s_j$ where $i < j \leq n$ such that $(s_i, s_{i+1}) \models T$ where T is the transition formula of $\pi[i]$ and there exists an assume statement in the trace π at position j such that $s_j \not\models guard(\pi[j])$*

Definition 9 (Relevance of an assigning statement)**.** *Let $\pi = \langle st_1, ..., st_n \rangle$ be an error trace of length n where st_i is an assigning statement at position i that assigns a new value to some variable x. The statement st_i is relevant if there exists an execution $s_1, ..., s_{n+1}$ of π and some value v such that every execution of the trace $\langle x := v; \pi[i+1, n] \rangle$ starting in s_i is blocking.*

Algorithm 1 Relevance of an assigning statement

```
1: procedure RELEVANCE
2:     trace ← Error trace π of length n
3:     relevantStatements ← [ ]
4:     for i = n to 1 do
5:         Q ← ¬wp(false; trace[i + 1, n])
6:         P ← wp(Q; trace[i]) ∩ sp(true; trace[0; i − 1])
7:         if trace[i] is an assigning statement and P ⊈ wp(Q; havoc(x)) then
8:             relevantStatements.append(trace[i])
       return relevantStatements
```

Theorem 1 (Equivalence of relevance)**.** *Let $\pi = \langle st_1, ..., st_i, ..., st_n \rangle$ be an error trace of length n and $\pi[i]$ be an assigning statement at position i that assigns a new value to some variable x. Let $P = \neg WP(false; \pi[i, n]) \cap SP(true; \pi[0, i-1])$ be a set of bi-reachable states at position i and $Q = \neg WP(false; \pi[i+1, n])$ be the co-reachable states at position $i+1$. The statement $\pi[i]$ is relevant iff:*

$$P \not\subseteq WP(Q, havoc(x))$$

Proof. Let $\mathcal{D}$ be the domain of the variable x.
"$\Rightarrow$"
If $\pi[i]$ is relevant, then we have to show that

$$P \not\subseteq WP(Q; havoc(x))$$

Obviously all the transitions from the states in $WP(Q; havoc(x))$ end up in Q. Relevance of $\pi[i]$ implies that there is a state s in P such that there is a transition from s to $\neg Q$. That would mean:

$$P \not\subseteq WP(Q; havoc(x))$$

"$\Leftarrow$"
$\pi[i]$ is relevant, if:

$$P \not\subseteq WP(Q; havoc(x))$$

We know that $WP(Q; havoc(x))$ is the set of states from which all transitions end up in Q. The above non-implication shows the existence of a state s in P such that $s \notin WP(Q; havoc(x))$ from which there is a transition to $\neg Q$. This shows the existence of a value $v \in \mathcal{D}$ that we can assign to x such that if we replace $\pi[i]$ with $x := v$, then every execution starting from s is becoming blocking. Also, from our assumption, it is clear that there exists an execution till s, since P is not empty. □

5.3 Examples

Consider the failed examples from previous definitions and let us analyze them again under our new definition.

```
1 foo()
2 {
3   x := 1;
4   y := 2;
5   z := 3;
6   havoc z;
7   assert(z > 10);
8 }
```

In the example above which contains only one error trace and where the restrictive-assume check fails, only the havoc statement at line 6 is now relevant according to our new definition.

Consider the example where the blocking-executions check failed:

```
1 foo()
2 {
3   y := 10;
4   havoc x;
5   assert(x > 0);
6 }
```

Only $havoc(x)$ is now relevant since there is no assignment to y in line 3 that can make an existing execution of the error trace blocking.

6 Possibly counter-intuitive cases

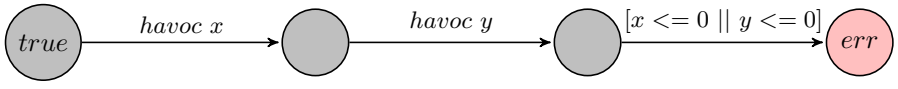
This section mentions three examples with counter-intuitive results when analyzed by our algorithm. All the examples can be found in ULTIMATE repository's examples folder for Error localization. For future reference, let us name these examples as follows:

6.1 Two-havoc case

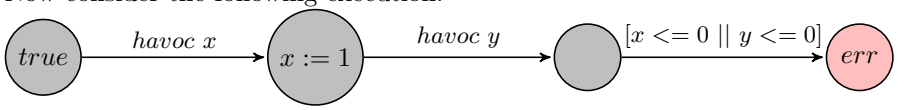
Consider the following program:

```
1 procedure foo()
2 {
3   havoc x;
4   havoc y;
5   assert(x>0 && y>0)
6 }
```

Our algorithm only consider the second havoc as relevant. This seem rather surprising at first, but consider an execution of the error trace which starts from *true*.



In this execution, no value v exists such that if we replace $\langle havoc\ x \rangle$ with $x := v$, every execution of the trace $\langle x := v;\ havoc\ y;\ assume(x <= 0\ ||\ y <= 0) \rangle$ starting in the state $true$ is blocking. Hence $havoc\ x$ is indeed not relevant. Now consider the following execution.



In this execution, a value v exists ($v >= 1$), such that if we replace $\langle havoc\ y \rangle$ with $y := v$, every execution of the trace $\langle y := v;\ assume(x <= 0\ ||\ y <= 0) \rangle$ starting in the state $x := 1$ is blocking. Hence it becomes clear why $havoc\ y$ is relevant.

6.2 Relevant-havoc case

Consider the following program:

```
1  procedure foo(a,b)
2  {
3      havoc x;
4      if(x > 10){
5        x := x+1;
6      }
7      else {
8        x := x-1;
9      }
10     assert a > b;
11 }
```

Our algorithm considers $havoc\ x$ as relevant. The program have two error traces with both of them having an assume statement with the guard involving the variable x. Therefore, there exists a value v, such that if we replace $havoc\ x$ with $x := v$, all the executions for the rest of the trace are becoming blocking.

6.3 Two-assignment case

Consider the following example:

```
1  procedure foo()
2  {
3      x := 10;
4      y := 10;
5      assert x < 10 && y < 10;
6  }
```

In this case none of the assignment statements are relevant since the assume statement in the end would have a disjunction and changing only one assignment statement does not make the execution blocking.

7 Known problems

Currently, our implementation in ULTIMATE fails to give expected results when array modification is involved in the program. This is due to the fact that we havoc the whole array at the moment and not one element. Due to this problem, if any assignment to an array element is relevant according to our algorithm, all the assignments that modify any element of the same array will also be relevant. The following example will make this problem clearer:

```
1 foo()
2 {
3   havoc j;
4   i := j + 1;
5   a[i] := 5;
6   a[j] := 7;
7   assert(a[i] == 8);
8 }
```

The implementation marks both assignment statements modifying the array elements as relevant.

8 Comparison with other approaches

8.1 Error Invariants

One weakness of the *Error Invariants* [2] approach is its inability to deal with havoc statements. The other weakness is related to the fact that generating error invariants require an unsatisfiable trace formula. This leads to a limitation in the case when there is an assert false at the end of that unsatisfiable trace formula. In this case, all the interpolants can be chosen as true and none of the statements would be considered relevant. We call this the *assert-false problem*. Following is a very simple example where this problem becomes easy to see:

```
1 foo()
2 {
3   x := 1;
4   if(x < 10)
5   {
6     assert false;
7   }
8 }
```

All the interpolants can be chosen as true and $x := 1$ would not be considered relevant in this case. The following example is a characterization of the assert-false problem. It is a modified version of an example that we used in the *POPL* submission.

```
1 foo()
2 {
3   p := -1;
4   i := 1; //  Bug: i should be 0
5   while(i < 10){
6     if(i==0){
```

```
7         p := 0;
8       }
9       assert( p != -1):
10      i++;
11    }
12  }
```

In this example, only line 3 would be considered relevant according to the error invariants approach. Where as the bug is that the user actually forgot to initialize i with 0. Our algorithm was however able to catch this problem.

8.2 Flow Sensitive Fault localization

In the paper *Flow Sensitive Fault Localization* [1], the authors try to take into account the control flow of the program and propose a flow sensitive error localization technique which, they hope, not only explains the cause of the error but also explains why the statements leading to the error were executed. They do this by modifying the so called *error trace formula* to something which they call a *flow sensitive error trace formula*. An error trace formula is just a conjunction of the trace formula of the error trace together with the error precondition and the correctness assertion. Information about the relevant statements can be obtained from the proof of unsatisfiability of the error trace formula.

The flow sensitive error trace formula is a modification of the error trace formula, in a sense that it now keeps track of dependencies between statements and the branching conditions that are relevant for the reachability of these statements in the control flow graph of the program.
The main focus of the authors here is on finding the relevant assume statements and the statements that cause the guard of the relevant assume statements to hold. We on the other hand only focus on the statements that cause the state of the program to change and do not analyze assume statements directly. However, we are still able to find the assignment statements that cause the guard of the relevant assume statements to hold. Let us look at an example where our algorithm can already find the statements which the authors can find only after modifying there formula.

```
1 foo()
2 {
3   x := 1;
4   y := input - 42;
5   if (y < 0) {
6     x := 0;
7   }
8   assert(x != 0);
9 }
```

Both the flow sensitive approach and our algorithm say that the assignment statements $y := input - 42$ and $x := 0$ are relevant for the error.

9 Future work

9.1 Call/Return statements

We yet have to see the behavior of our algorithm in the context of procedure calls and define when a call and return statement should be relevant in an error trace. This would require some deeper study and would therefore be considered in future work.

9.2 Security Analysis

An interesting problem that we might be able to solve is to distinguish security bugs from non-security bugs. According to our initial understanding, a bug is a security bug iff an user input statement can cause the program execution to reach the error state.
We list some possible criteria for a bug to be called a security bug.

1. There is some reachable location where a program reads an input.
2. There is some input value such that continuing from this location we definitely reach the error.
3. There is some input that continuing from this location we do not reach the error.

From condition 2 and 3, we can conclude that the input is somehow relevant for the error. This is of course just our initial understanding on this topic, which may not be accurate and further work is needed to formally define a security bug which would involve analyzing in detail the examples that contain security bugs.

References

[1] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. Flow-sensitive fault localization. In *VMCAI*, volume 7737 of *LNCS*, pages 189–208. Springer, 2013.

[2] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM*, volume 7436 of *LNCS*, pages 187–201. Springer, 2012.