Static fault localization for simple bugs

(background work)

Numair Mansur

University of Freiburg, Germany

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. The document also mention some examples that we came across that show why the previous definitions didn't work. It also reveals some limitations of our defintion of relevance by showing some more examples. The document should also serve the purpose of a reference for our future work in this area. This will also be my submission for my master project report.

2 Motivation

(Read the paper and write the motivation again)
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*. There can be many techniques to help the user to narrow down his or her search for *relevant* statements. We in this work present a new such technique that statically analyze all the assigning statements in an error trace and return those that are playing a direct role in taking the program exeuction to the error state. Many such techniques already exist in the literature that return error-relevant statements based on different criterias and strategies. In this work we focus on program statements that change the state of the program and analyze if that change can have an effect on the reachabilty of the error. A program might be failing because of several reasons which are not necessarily simple(?) and easy to point out and there might be many ways to fix the error. We however in this work 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. Not only did we want to help 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 and the curent definition

of relevancy was reached after some time and effort. All the previous definitions are also presented in this document along with the reasons of why they were wrong and the examples on which they failed.
(Maybe also talk about how we developed the algorithm in the previous semester).
(anything more ?)

3 Discarded definitions of relevance:

While developing a formal definition for relevance, we came across several versions that seemed correct at the time but in reality didn't accurately and completley explain our notion of relevance and hence were discarded. We discovered this by coming up with counter examples where these definitions didn't work. This section mentions those discarded definitions and the counter examples.
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. Since we declare a statemnet relavent if they satisfy a check, hence lets name these definitions with a check.

3.1 The restricitve-assume check:

We defined two seperate checks for checking the relevance of an assignment statement and a havoc statement. ***Explain a bit more here why?***

3.1.1 Assignment statement:

An assignment statement was marked relevant, if we replace that assginment with a havoc and some assume statement in the error trace was becoming restrictive.

Definition 1 (Restrictivness 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 $\pi[i]$* restricitve *iff :*

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

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 Ψ 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 for π' and Ψ.*

3.1.2 Havoc statement:

Definition 3 (Relevancy 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 $x := t$ be an assignment statement where x is the same variable as in $havoc(x)$ and t is an expression. The havoc statement $\pi[i]$ is relevant if there exists an assignemnt $x := t$ such that if we replace the havoc in π with $x := t$ to get a new trace π', then the following implication holds:*

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

3.1.3 Examples:

Where it works:
Below is an example where the above check for relevancy 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 assume statement ($assume(z <= 10)$)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 trace 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 $havoc(y)$ have no effect on the restrictivness of an already restrictive error trace. Hence it should not be relevant here. However, clearly this statement have an effect on the rechability of the error.

Also talk about the proofs. (maybe not ? chris did not like the idea)

3.2 The blocking-executions check:

We adopted this definition for relevance when we realized that we should take into account the "amount" of restrictivness 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 ϵ 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 exits an assume statement in the trace π at position j such that $s_j \not\models guard(\pi[j])$.*

Definition 6 (Relevance)**.** *Let β 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 β' 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 blocked executions than the trace before the replacement, i.e if $\beta \subsetneq \beta'$.

3.2.1 Examples:

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

```
procedure foo()
{
  y := 42;
  havoc x;
  assume(x >= 0 && y >= 23);
  assert(false);
}
```

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

```
procedure foo()
{
  y := 10;
  havoc x;
  assume(x > 0);
}
```

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

4 Final version of Relevance

4.1 Introduction:

An assignment statement ($x := t$ or $havoc(x)$) is responsible in an error trace if the assigned value matters for the reachability 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 responsible.

4.2 Formal defintion:

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 $s_i, s_{i+1} \vDash T$, 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 $s_i, s_{i+1} \vDash T$ where T is the transition formula of $\pi[i]$ and there exists an assume statement in the trace π at position j such that $s_j \nvDash guard(\pi[j])$*

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 has a blocking execution.*

Algorithm 1 Relavance of an assigning statement

```
1: procedure Relevance
2:     $trace \leftarrow$ Error trace $\pi$ of length $n$
3:     $relevantStatements \leftarrow [\ ]$
4:     for $i = n$ to 1 do
5:         $Q \leftarrow \neg wp(false; trace(i+1, n))$
6:         $P \leftarrow wp(Q; trace(i)) \cap sp(true; trace(1; i-1))$
7:         if $P \nsubseteq wp(Q; havoc(x))$ then
8:             $relevantStatements.append(trace(i))$
    return $relevantStatements$
```

In the algorithm , we check the relevance of a statement by checking if the triple $(P, \pi[i], \neg Q)$ is unsatisfiable and $\pi[i]$ is in the unsatisfiable core. We can do this by checking if $P \nsubseteq WP(Q; havoc(x))$.

4.3 Examples

Only put the examples here where the previous approaches failed.
Consider an error trace $x := 1, y := 2, z := 3, havoc(z), assume(z <= 10)$ obtained from the following program:

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

In the example above only the havoc statement at line 6 is relevant. One might assume that the assignment statement involving z might also be relevant. But if there exists an execition $\mathcal{E}$ of the trace, there is no assignemnt to z at this point such that we have a blocked execution of the subtrace from this assignment onwards starting with a state taken from the execution $\mathcal{E}$ just before the assignment $z := 3$.
Consider another example:

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

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

(more examples to follow)

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 , which assigns a new value to some variable x . Let $P = \neg WP(False; \pi[i, n]) \cap SP(True; \pi[1, i-1])$ be a set of bireachable states at position i and $Q = \neg WP(False; \pi[i + 1, n])$ be the coreachable 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

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

Obviously all the transitions from the states in $WP(Q; havoc(x))$ ends up in Q. Relevancy of $\pi[i]$ implies that there is a state in $s \in P$ 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 is becoming blocking. Also, from our assumption, it is clear that there exits an execution till P, since P is not empty. □

5 Algorithm modification:

We also noticed an error in our algorithm. 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 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 is becoming restrictive.

But our previous algorithm marks it as relevant. Because in our previous algorithm, 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 $sp(true, \pi[1, i - 1])$. The complete algorithm can ofcourse be found in our POPL submission.

[Check it again with the paper. maybe some error in the formula]

6 Limitations of our definition

This section mentions two known examples that reveal the weakness of our approach and we should keep them in mind as we further develop the notion of relevance. For furture reference, let us name these examples as follows.

6.1 Two havocs problem:

Conside the following program:

```
procedure foo()
{
  havoc x;

  // some other code

  havoc y;
  assume(x>0 && y>0)
}
```

Our algorithm only considers second havoc as relevant. The example can be found in ultimate repository's toy examples for Error localization.

6.2 Relevant havoc problem:

Consider the following program:

```
procedure foo(a,b)
{
    havoc x;
    if(x > 10){
      x := x+1;
    }
    else {
      x := x-1;
    }
    assert a > b;
}
```

Our algorithm considers havoc x relevant even tough fixing this statement will not fix the error.

7 Comparison with other approaches

7.1 Error Invariants [1]

The biggest limitation with the error invariants approach is their inability to deal with havocs and the assert false problem.(Maybe explain in the appendix assert false problem). The error invariant approach require a big unsatisfiable forumla.
Generalization / Characterization of the assert false problem.
Consider the following example:

```
foo()
{
  p1 := -1;
  p2 := -1;
  i := 1; This is the problem here
  while(i < 10){
    if(i==0){
      p1 := 0;
      p2 := 0;
    }
    assert( p1!= -1):
    i++;
  }
}
```

In the above program, the error invariant will say only havoc x is responsible for the error. But we say something differently.
POTENTIAL BUG IN THE IMPLEMENTATION !! Check later

7.2 Flow Sensitive Fault localization [3]

In the paper flow sensitive fault localization, the authors try to take into account the control flow of the program and propose a flow senstive error localization technique which, they hope, not only explain the cause of the error but also explain 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. There are however serious shortcomings with this approach of finding relevant statements to begin with.
The flow sensitive error trace formula is a modification of the error trace forumla, such 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 analyse assume statements directly. However, we are still able to find the assignment statements that cause the guard of the relevant assume statemets to hold. Lets look at an example where the flow sensitive approach and our algorithm gives exactly the same result.

```
foo()
{
  x := 1;
  y := input - 42;
  if (y < 0) {
    x := 0;
  }
  assert(x != 0);
}
```

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

(Example where we give different results ?)

8 Future work

8.1 Call Return

8.2 Security Analysis

References

[1] E.Ermis, M. Schaf, and T. Wies. Error Invariants. In FM'12, pages 338–353. Springer, 2012.

[2] M. Schaf, D. Schawrtz, T. Wies. Explaining Inconsistent Code. In Joint meeting of the European Software Engineering conference and the Symposium on the Foundations of Software Engineering, ESEC/FSE'13, pages: 521 - 531, Saint Petersburg, Russian Federation. August 18-26,2013

[3] J. Christ, E. Ermis, M. Schaf, and T. Wies. Flow-sensitive fault localization. In VMCAI, volume 7737, pages 189–208, Berlin, Heidelberg, 2013. Springer