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 approaches that lead us to the final version. The document also mention some examples that we came across that show why the previous approaches didn't work. The document should also serve the purpose of a reference if we forget something or some stuff like that. i dont know
This document will also serve as a submission for my master project.
bla bla bla...some more stuff about the document.

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:

This section mentions the defintions for relevancy that are pre-versions(?) of the definition that is in the final POPL submission. These defintions have been discarded due to their inability to correctly and completely explain our notion of relevancy. They are in the document primarily for the reason of documenting failed/incorrect versions of our 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.

3.1 The restricitve-assume check:

I think here we have to talk assignemtn and havoc relevancy seperately. and maybe also talk about the proofs.

3.1.1 Approach

In this approach, we said that we replace an assignment with a havoc and if some assume in the trace is becoming restrictive then the assignment statement is becoming restrictive. What was the criteria for the havoc again ?

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 a 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 Example where it works

```
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 restrictive.

3.1.3 Example 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.

3.2 The blocking-executions check:

3.2.1 Approach

We adopted this definition for relevance when we discovered 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.

Definition 3 (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 4 (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*

3

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

Definition 5 (Relevancy of an assignment statement). *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 $\beta \subsetneq \beta'$.

3.2.2   Example where it works

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

This definition now correctly says that $y := 42$ is relevant since changing it to havoc gives us more blocking executions then before.

3.3   Example where it fails

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

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 to this defintion, it is wrongly marked as relevant too.

3.4   SP Intersection check(Implementational detail)

bla bla
(Also tell here, how we treated the havoc relevancy and normal relevancy seperatlely)

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 6 (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 7 (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 8 (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 π 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

I don't think there is a need to put examples here. 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:*

$$P \nsubseteq WP(Q, havoc(x))$$

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

$$P \nsubseteq WP(Q; havoc(x))$$

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:

$$P \nsubseteq WP(Q; havoc(x))$$

"$\Leftarrow$"
$\pi[i]$ is relevant, if:
$$P \nsubseteq WP(Q; havoc(x))$$

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 Examples where our algorithm does not work

add the examples which expose some problems.
i can think of atleast 2 at the moment. 2 havoc problem.
another one was that big one. can't remember the name at the moment. but definatley there.

6 Comparison with other approaches

6.1 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:

```
1  foo()
2  {
3    p1 := -1;
4    p2 := -1;
5    i := 1; This is the problem here
6    while(i < 10){
7       if(i==0){
8          p1 := 0;
9          p2 := 0;
10      }
11      assert( p1!= -1):
12      i++;
13    }
14 }
```

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

6.2 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 formula, 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.

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

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 ?)

7 Future work

7.1 Call Return

7.2 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