1 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 *error localization*. There can be many techniques to help the user to narrow down his or her search for *responsible* statements. We in this paper present a new such technique that analyze all the assigning statements in a failing program execution 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-responsible 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. Let us look at a simple example that can help us see this a bit more clearly.

(add a simple but good example here that makes it for us to see the application of our approach)

2 Responsible

Definition 1 (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 2 (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 $s_j \not\Rightarrow guard(\pi[j])$*

Definition 3 (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.*

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

2.1 Examples

Consider an error trace $x := 1, y := 2, z := 3, havoc(z), assume(z <= 10)$ obtained from the following program:

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: $relevance \leftarrow checkUnsatCore(P, trace(i), Q)$
8: **if** $relevance =$ "$unsat$" and $trace(i)$ in "$unsatCore$" **then**
9: $relevantStatements.append(trace(i))$
return $relevantStatements$

```
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. □

3 Comparison with other approaches

3.1 Error Invariants [1]

Consider the following example:

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

According to our algorithm, only line number 4 is relevant since only at this location there exits an assignment to x such that the trace from here on is getting blocked. But at line 3, no such assignment exists. According to the approach using error invariants, the assignment at line 4 is not relevant since this statement have no effect on the error invariant and the error invariant remains the same (inductive error invariant).
Intuitively, it is not so helpful for the user if line 3 is marked relevant. Because even if that line is *fixed*, the program is still ending up in an error state. However, fixing line 4 also fixes the program.
Consider another example:

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

According to our algorithm, only the second assignment to x is relevant for the error, since an assignment to x at this point can make an existing execution for the rest of the trace blocking. On the first look it seems that the assignment at line 3 should also be relevant, because it can make an existing execution blocking for the rest of the trace but we can always assign z such a value that it can overide the effect of the assignment at line 3 and make the execution unblocking again.
The error invariant algorithm is not smart enough to recognize that and it says that both of the assigning statements to x are relevant. Since it bases its judgement on changing error invariants which are changing after line 3 and 4 both.
We however believe that our result is more helpful to the user since changing this one line can fix the error in the program.

3.2 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.

```
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 ?)

4 Previous/Failed approaches:

4.1 Replace with havoc and an assume is restrictive

4.1.1 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 4 (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 5 (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.*

4.1.2 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.

4.1.3 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.

4.2 Approach with blocking executions

4.2.1 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 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 $s_i, s_{i+1} \models T$, where T is the transition formula of $\pi[i]$.*
Let ϵ represent the set of all possible executions of the error trace.

Definition 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 $s_i, s_{i+1} \models T[i]$, 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 $s_j \not\Rightarrow guard(\pi[j])$.*

Definition 8 (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 $\beta \subsetneq \beta'$.

4.2.2 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.

4.2.3 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.

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