1 Relevance

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 $s_j \nvDash guard(\pi[j])$*

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.*

Algorithm 1 Relavance of an assigning statement

```
procedure RELEVANCE
    $trace \leftarrow$ Error trace $\pi$ of length $n$
    $relevantStatements \leftarrow [\ ]$
    for $i = n$ to 1 do
        $Q \leftarrow \neg wp(false; trace(i+1, n))$
        $P \leftarrow wp(Q; trace(i)) \cap sp(true; trace(1; i-1))$
        $relevance \leftarrow checkUnsatCore(P, trace(i), Q)$
        if $relevance =$ ”$unsat$” and $trace(i)$ in ”$unsatCore$” then
            $relevantStatements.append(trace(i))$
    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 $P \not\subseteq WP(Q; havoc(x))$.

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. □

2 Comparison with other approaches

2.1 Error Invariants approach

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 can also fix the program.

3 Previous/Failed approaches:

3.1 Replace with havoc and an assume is restrictive

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 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.*

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

```
foo()
{
  x := 1;
  y := 2;
  z := 3;
  havoc z;
  assert(z > 10);
}
```

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:

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

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 Approach with blocking executions

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 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 $\beta \subsetneq \beta'$.

3.2.2 Example where it works

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

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

3.2.3 Example where it fails

```
procedure main()
{
  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 to this defintion, it is wrongly marked as relevant too.