Explaining Inconsistent Code

Muhammad Numair Mansur
Introduction

- 50% of the time in debugging
- Fault localization.
- Becomes more tedious as the program size increase.
- Automatically explaining and localizing *inconsistent code*. 
A code fragment is inconsistent if it is not a part of any normally terminating execution.

Not necessarily always a bug!

But sometimes inconsistent code results in an error.
class Example1 {
    public int doubleLoop(int x, int y) {
        int ret=0;
        for (int i=0; i<10; i++)
        {
            for (int j=0; j<10; i++) {
                ret++;
            }
        }
        return ret;
    }
}
Examples (Unreachability)

```java
class Example1 {
    public int doubleLoop(int x, int y) {
        int ret=0;
        for (int i=0; i<10; i++)
            for (int j=0; j<10; j++)
                ret++;
        return ret;
    }
}
```
```java
class Example5 {

    void foo(int i, int len) {
        if (i < 0 || i > (len - 1)) {
            throw new IllegalArgumentException();
        }
        if (i > (len - 1)) {
            return;
        }
    }
}
```
class Example5 {

    void foo(int i, int len) {
        if (i < 0 || i > (len - 1)) {
            throw new RuntimeException();

            // Unreachable: The case where this conditional evaluates to true conflicts with the other marked lines
            if (i > (len - 1)) {
                return;
            }
        }
    }
}
Our Goal

Automatically explain inconsistent code.
Our Goal

Automatically explain inconsistent code.

Inconsistent program Automata $\mathcal{A}$

Pre

Algorithm

Error Invariant Automaton $\mathcal{A}_I$

Post
Our Goal

Automatically explain inconsistent code.

Inconsistent program Automata $\mathcal{A}$

Pre

Post

Algorithm

Error Invariant Automaton $\mathcal{A}_I$
A F.A is a 5 tuple: $(Q, \Sigma, \delta, q_o, F)$

$Q$: A finite set of states.

$\Sigma$: A finite set of input symbols called an alphabet.

$\delta$: A transition function ( $\delta: Q \times \Sigma \rightarrow Q$ ).

$q_o$: initial state.

$F$: A finite set of final states.
Finite automata

Example:
Finite automata

Example:
Finite automata

Example:
Finite automata

- Transitions through the states based on the input
- True, if ends in an accepting state

input
(A sequence from the input alphabet)

F.A

Output
(accept or reject)
Finite automata

Example:

$\Sigma = \{a,b,c\}$
Input: abca
Finite automata

Example:

$\Sigma = \{a,b,c\}$

Input: abca
Finite automata

Example:

$\Sigma = \{a, b, c\}$

Input: abca
Finite automata

Example:

$\Sigma = \{a, b, c\}$

Input: abca
Finite automata

Example:

\[ \Sigma = \{a, b, c\} \]

Input: abca
Program automata

A simple and an abstract model of a program.
Program automata

A simple and an abstract model of a program.

Defined in terms of a finite automata.

State \((Q)\) = Program Location \((\text{Loc})\)

Transition \((\delta)\) = Program Statement \((\delta_\text{p})\)

Alphabet\((\Sigma)\) = A set of program statements

Initial State \((q_0)\) = Initial program Location \((\ell_0)\)

Final State \((F)\) = Final program Location \((\ell_e)\)
1: public void example(boolean b)
2: MyClass x = null;
3: if (b) {
4:   x.foo();
5: }
6: x.bar();
7: }

An example program
Program automata

1: public void example(boolean b)
2: MyObject x = null;
3: if (b) {
4:   x.foo();
5: }
6: x.bar();
7: }

**assume(b)** means that the branch of if () is taken where b is “true”

**assume(!b)** means that the branch of if () is taken where b is “not true”
1: public void example(boolean b)
2: MyObject x = null;
3: if (b) {
4:   x.foo();
5: }
6: x.bar();
7: }

An assertion on the program state that x != null
A run $\rho$ is a finite sequence of locations and statements.
\[ l_0 st_1 l_1 \ldots \ldots st_{n-1} l_n \]

A path($\rho$) $st_0 st_1 \ldots st_{n-1}$ is the path associated with a run.

A run $\rho$ is accepting if its final state is $l_e$.

A word $\pi \in \Sigma^*$ is a path if $\pi = \text{path}(\rho)$ for some accepting run $\rho$. 
Our Goal

To automatically explain inconsistent code.
Algorithm

**Input:**
- \( Pre \): precondition state formula
- \( \mathcal{A} \): program automata
- \( Post \): Postcondition state formula

**Output:**
- \( \mathcal{A}_f \): error invariant automata.

**Requires:**
- \( \mathcal{A} \) is inconsistent subject to \( Pre \) and \( Post \).

**Ensures:**
- \( \mathcal{A}_f \) explains \( \mathcal{A} \).
Algorithm

Step 1: Translate the program automata $\mathcal{A}$ into a single path of statements $\pi_\mathcal{A}$.
Algorithm

Step 1: Translate the program automata $\mathcal{A}$ into a single path of statements $\pi_\mathcal{A}$. 
Algorithm

Step 1: Translate the program automata $\mathcal{A}$ into a single path of statements $\pi_\mathcal{A}$.

It can be composed of many atomic statements.
Algorithm

Example:

This was the first step in getting the final result, an error invariant automata.
Error Invariant Automaton

- An abstraction of the program, that only mentions the statements and facts that are relevant for understanding the cause of the inconsistency.
- The irrelevant statements are first summarized as first order logical formulas and then eliminated.
- These formulas are called error invariants.
- An error invariant captures the reason of abnormal program termination.
- So, at a high level, an Error Invariant Automaton replaces code which does not contribute to the inconsistency with a suitably chosen invariant.

Let's see this in practice on a fragment of code.
1: public TaskDialog(Tast task)
~: . . . . .
6: txtDescription.setTask(task.getDescription());
~: . . . . .
16: if (notification)
    {
        . . . .
    }
~: . . .
27: chbRegular.setEnabled(task == null);
~: . . .
1: public TaskDialog(Tast task)  

6: txtDescription.setTask(task.getDescription());

16: if (notification)  
    {  
        . . . . . .
    }

27: chbRegular.setEnabled(task == null);

No Effect on inconsistency

assert ( task != null )

Arbitrary code

No effect on task == null

An assertion that task might be null

No Effect on inconsistency
1: public TaskDialog(Tast task)  
   \(\text{assert ( task != null )}\)  
   \(\text{Arbitrary code}\)  
   \(\text{No effect on task == null}\)  
   \(\text{No Effect on inconsistency}\)  

6: txtDescription.setTask(task.getDescription());  

16: if (notification)  
   \{  
   \}  

27: chbRegular.setEnabled(task == null);  
   \(\text{An assertion that task might be null}\)  
   \(\text{No Effect on inconsistency}\)
Error Invariant Automaton

1: public TaskDialog(Tast task)
   ~: ....... line 1 - 5 No Effect on inconsistency
6: txtDescription.setTask(task.getDescription());
   ~: ....... line 6 assert ( task != null )
16: if (notification)
    { line 7 - 26 Arbitrary code
      .......
    }
   ~: .......
27: chbRegular.setEnabled(task == null);
   ~: .......
   line 27 An assertion that task might be null
   line 28 - end No Effect on inconsistency

\( \mathcal{L}_{1-6} \) true
\( \mathcal{L}_{7-26} \) task != null
\( \mathcal{L}_{28-\text{end}} \) false
An error trace is a sequence of statements $\pi = st_0 st_1 ... st_n$, together with $pre$ and $post$. $pre$ describes the initial state and $post$ is an assertion that is violated. That means, in an error trace

$$Pre \land PF(\pi) \land Post$$

is unsatisfiable.
An error trace is a sequence of statements $\pi = st_0, st_1, \ldots, st_n$, together with $pre$ and $post$.
$pre$ describes the initial state and $post$ is an assertion that is violated.
That means, in an error trace

$$Pre \land PF(\pi) \land Post$$

is unsatisfiable.
Example:

$$True \land assume(task = null) \land assume(task \neq null) \land True$$
An error invariant for a position $i \in [0,n+1]$ in an error trace is a first order logical formula such that:

- The conjunction of the first order logical formulas for each statement implies $I_i$.
- $I_i$ and the conjunction of the remaining formulas is unsatisfiable.
In the previous work, the authors introduced a function \( \text{ErrInv}(pre, \overline{A}, post) \) which given an error trace, computes:

\[
l_0, s_{i_1}, l_1, s_{i_2}, \ldots, s_{i_k}, l_k
\]

Such that,

\( s_{i_1}, s_{i_2}, \ldots, s_{i_k} \) is a subsequence of \( \overline{A} \) and \( l_j \) is an inductive invariant for the position \( i_j \) and \( i_{j+1} \).
We say that an error invariant is inductive for position $i < j$ if: 

$\mathcal{C} \xrightarrow{I} \mathcal{C}_i \xrightarrow{I} \mathcal{C}_j \xrightarrow{I} \mathcal{C}$
We say that an error invariant is inductive for position $i < j$ if:
We say that an error invariant is inductive for position $i < j$ if:

$I$ is called an inductive error invariant.
Error Invariant Automaton

An Error Invariant Automaton is an inconsistent program automaton with a mapping $I$ from locations $\mathcal{L}$ of $\mathcal{A}_I$ to state formulas, such that for all locations $\ell$, $I(\ell)$ is an error invariant for $\ell$. 
Now, after applying step 1 we got a single path \( \pi_A \).

**Step 2:** Apply \( \text{ErrInv}(\pi_A) \)

\[
\text{ErrInv}(\pi_A) = \text{ErrInv}(\text{Pre}, \pi_A \text{Post}) = l_0 \text{st}(l_{i1}) \ldots \text{st}(l_{ik}) l_k.
\]
Now, after applying step 1 we got a single path $\pi_A$.

**Step 2:** Apply $\text{ErrInv}(\pi_A)$

$\text{ErrInv}(\pi_A) = \text{ErrInv}(Pre, \pi^A, Post)$

$= l_0\text{st}(l_{i_1}) \ldots \ldots \text{st}(l_{i_k})l_k.$
Now, after applying step 1 we got a single path $\pi_A$.

**Step 2:** Apply $\text{ErrInv}(\pi_A)$

$$\text{ErrInv}(\pi_A) = \text{ErrInv}(\text{Pre, } \pi_A, \text{Post}) = I^0 \cdot \text{st}(I^1) \cdot \ldots \cdot \text{st}(I^k) \cdot I^k.$$
Example:

\[ I_{1-6} \]

\[ I_{7-26} \]

assume(task !=null)

assume(task =null)

\[ I_{28-n} \]
Example:

Algorithm

\[
\begin{align*}
\ell_{1-6} \quad \quad &\quad \quad \text{true} \\
\ell_{7-26} \quad \quad &\quad \quad \text{assume(task !=null)} \\
\ell_{28-n} \quad \quad &\quad \quad \text{assume(task =null)} \\
\end{align*}
\]

\[
\begin{align*}
\ell_{1-6} \quad \quad &\quad \quad \text{false} \\
\ell_{7-26} \quad \quad &\quad \quad \text{assume(task !=null)} \\
\ell_{28-n} \quad \quad &\quad \quad \text{assume(task =null)} \\
\end{align*}
\]
Algorithm

Step 3:
The locations covered with an inductive error invariant can be collapsed into a single location.
Step 4: For each remaining non-atomic statement, apply the algorithm recursively to all these smaller automata.
Step 4: For each remaining non-atomic statement, apply the algorithm recursively to all these smaller automata.
Algorithm

Step 4: For each remaining non-atomic statement, apply the algorithm recursively to all these smaller automata.
Step 4: For each remaining non-atomic statement, apply the algorithm recursively to all these smaller automata.

In case of the location with invariant $I_2$, the invariant is inconsistent with respect to pre and post conditions.
**Algorithm**

**Step 4:** For each remaining non-atomic statement, apply the algorithm recursively to all these smaller automata.

In case of the location with invariant $I_2$
Algorithm

**Step 4:** For each remaining non-atomic statement, apply the algorithm recursively to all these smaller automata.

In case of the location with invariant $I_2$, recursively...
Algorithm

**Step 4**: For each remaining non-atomic statement, apply the algorithm recursively to all these smaller automata.
Algorithm

1: public TaskDialog(Tast task)
~: . . . . . .
6: txtDescription.setTask(task.getDescription());
~: . . . . . .
16: if (notification)
    {
        . . . . . .
    }
~: . . . . . .
27: chbRegular.setEnabled(task == null);
~: . . . . . .

The approach was tested on 6 real world examples. For each of these examples, Error invariant automatas were generated using the technique introduced. All the generated error invariant automatas represented real world inconsistencies with no false alarms. Running time to prove inconsistency using unsat ranged from 0.008 seconds in one of the experiments to 0.019 seconds.
Usability testing was also conducted on 11 programmers. Half of the test subjects were shown the full programs, while the other half were just shown the error invariant automata. All candidates took 1 hour and 6 minutes to identify the bug. For the full programs without error invariant automata: 51 minutes. With error invariant automata: 17 minutes.
Conclusion

The experiments indicate that EIA provide useful visual assistance to spot inconsistencies.

EIA can also be used for fault localization on a single trace and thus provide a general tool to assist programmers in debugging.
References

Martin Schaf, Daniel Schawrtz, Thomas Wies
**Book Title:** Joint meeting of the European Software Engineering conference and the Symposium on the Foundations of Software Engineering, ESEC/FSE’13, Saint Petersburg, Russian Federation, August 18-26, 2013  
**url:** [http://dl.acm.org/citation.cfm?id=2491411](http://dl.acm.org/citation.cfm?id=2491411)  
**Title:** Explaining Inconsistent Code, pages: 521 - 531


Jurgen Christ, Evren Ermis, Martin Schaf, Thomas Weis
**Book Title:** Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings  
**Title:** Flow-Sensitive Fault Localization, pages: 189 - 208