Efficient Elimination of False Positives using Static Analysis

Tukaram Muske, Uday P. Khedker

TRDDC, Tata Consultancy Services, India
Indian Institue of Technology Bombay, India

International Symposium on Software Reliability Engineering
November 2-5, 2015
Gaithersburg, MD, USA

Background

- Static analysis - scalable but imprecise
- Model checking - precise but not scalable
- Static analysis + model checking $\Rightarrow$ better results
- False positives elimination using model checking[1]
  - Generate an assertion corresponding to each warning
  - Verification in incremental context
  - Scalable to some extent but a lot many model checking calls
  - Time consuming

[1] Hendrik Post et al. "Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking". In: *ASE*. 2008, pp. 188–197.

A Motivating Example

```c
const int arr[]
        ={0,2,5,9,14};
int var, factor; char ch;

void f1(){
 unsigned int i, j;

 i = lib1();
 j = lib2();
 var = lib3();

 if(j < 5 && i < j){
  factor= arr[j]-arr[i];
  f2();
 }
}
```

```c
int f2(){
  if(var==factor )
    f3(var);
  ...
}
```

```c
int f3(int p){
 int a, b, denom=1;
 if(ch < 5)
  denom = p;
 else
  denom = 10;

 assert(denom!=0);
 a = 100/denom;//warning
}
```

Context Expansion: Call 1

```
p = nondet_int();
ch = nondet_char();
```

Verifying the assertion

in the context of $f3$

results in

Counterexample

by assigning 0 to p

```
int f3(int p){
 int a, b, denom=1;
 if(ch < 5)
  denom = p;
 else
  denom = 10;

 assert(denom!=0);
 a = 100/denom;//warning
}
```

Context Expansion: Call 2

```
var = nondet_int();
factor = nondet_int();
ch = nondet_char();
```

```
int f2(){
  if(var==factor )
    f3(var);
  ...
}
```

```
int f3(int p){
 int a, b, denom=1;
 if(ch < 5)
  denom = p;
 else
  denom = 10;

 assert(denom!=0);
 a = 100/denom;//warning
}
```

Verifying the assertion

in the context of *f2*

results in

Counterexample

by assigning 0 to *var* and *factor*

Context Expansion: Call 3

Assertion holds when verified in the context of $f1$.

```
ch = nondet_char();
```

```
const int arr[]
        ={0,2,5,9,14};

void f1(){
 unsigned int i, j;

 i = lib1();
 j = lib2();
 var = lib3();

 if(j < 5 && i < j){
  factor= arr[j]-arr[i];
  f2();
 }
}
```

```
int f2(){
  if(var==factor )
    f3(var);
  ...
}
```

```
int f3(int p){
 int a, b, denom=1;
 if(ch < 5)
  denom = p;
 else
  denom = 10;

 assert(denom!=0);
 a = 100/denom;//warning
}
```

Motivation

- The problem
  - Large number of model checking calls
  - Hence, time consuming
- Observation
  - If **any arbitrary value** is allowed for an assertion variable at the assertion point, the assertion verification **results in a counterexample**.

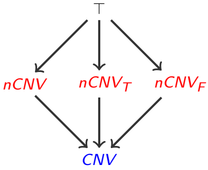
Our Approach

- Complete-range Non-deterministic Value (cnv) variables
  - Taking complete-range of non-deterministic values
  - Any arbitrary value is allowed
  - No value assignment/restriction through program code
- Complete-range Non-deterministic Value (cnv) expressions
  - $(x+10)$, $(x++)$, and $(x+y)$ **are** *cnv* expressions when $\{x,y,z\}$ are *cnv* variables.
  - $(x+10)$, and $(x++)$ **are not** *cnv* expressions when $\{y,z\}$ are *cnv* variables.
  - $(x/100)$, $(x\%2)$, and (100) in n **are not** *cnv* expressions even if x is a *cnv* variable.
- Identify redundant verification calls (RVCs) and skip them
  - using cnv variables

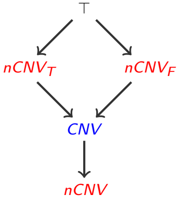
Computation of cnv variables

- Depends on
  - Context sensitivity
  - Flow sensitivity
  - May/Must reachability
  - Data and **control** dependance
- Using Data Flow Analysis
  - Lattice structure

```
void f(int x){
  //x ↦ CNV
  if(x < 10){
    //x ↦ nCNV_T
  }else{
    //x ↦ nCNV_F
  }
  //x ↦ CNV
}
```



(a) May cnv variables



(b) Must cnv variables

RVCs Identification

If all assertion variables are *cnv* variables, the call is redundant.

```
const int arr[]
        ={0,2,5,9,14};
int var, factor; char ch;

void f1(){
 unsigned int i, j;

 i = lib1();
 j = lib2();
 var = lib3();

 if(j < 5 && i < j){
  factor= arr[j]-arr[i];
  f2();
 }
}
```

```
int f2(){
  if(var==factor )
    f3(var);
  ...
}
```

```
int f3(int p){
 int a, b, denom=1;
 if(ch < 5)
  denom = p;
 else
  denom = 10;

 assert(denom!=0);
 a = 100/denom;//warning
}
```

Influence of cnv variables

```
void foo(){
 b = 0;
 if(v == 1){
  a = 10;
  b = 10;
 }
```

```
if(v == 1)
 assert(a != 0);//May Vs Must

if(b == 10)
 assert(v != 0);//O-AEP

if(x < 100)
  assert(x < 10);//Insufficiency
}
```

- Is RVCs identification accurate?
  - *may-cnv* variables Vs *must-cnv* variables
  - Impact of over-approximition of execution paths (O-AEP)
  - Insufficiency of *cnv* variables
- Define two parameters
  - Precision $= \dfrac{\text{number of } \textbf{\textit{correctly}} \text{ identified RVCs}}{\text{total number of identified RVCs}}$
  - Recall $= \dfrac{\text{number of } \textbf{\textit{correctly}} \text{ identified RVCs}}{\text{number of actual calls violating the assertions}}$

Effect on False Positives Elimination

- Define two parameters
  - Time saving $= 1 - \frac{\text{Time taken in } eFPE}{\text{Time taken by } FPE_{orig}}$
  - Elimination loss $= 1 - \frac{\text{False positives eliminated in } eFPE}{\text{False positives eliminated by } FPE_{orig}}$
- Trade-off
  - Precision Vs Recall
  - Time saving Vs Elimination loss

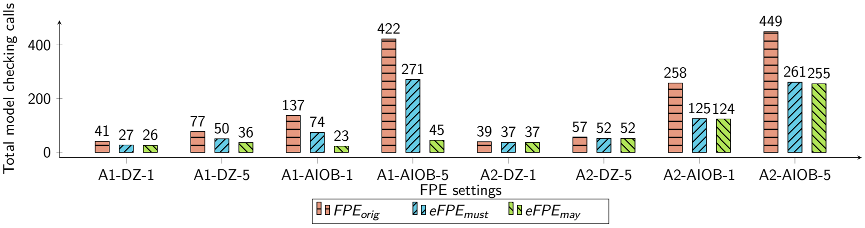
Experimental Set up

- *cnv* variables computation - implementation in TCS ECA[2]
- CBMC as the model checker[3]
- Two applications (50 KLOC and 40 KLOC)
- Two properties (DZ and AIOB)
- False positives elimination in three settings
  - FPE_{orig}, $eFPE_{may}$, and $eFPE_{must}$
- Each setting with two context levels
  - maxCCL $=1$, and maxCCL $=5$
- Assumptions
  - assertions are reachable
  - code is sliced and every slice corresponds to a single assertion

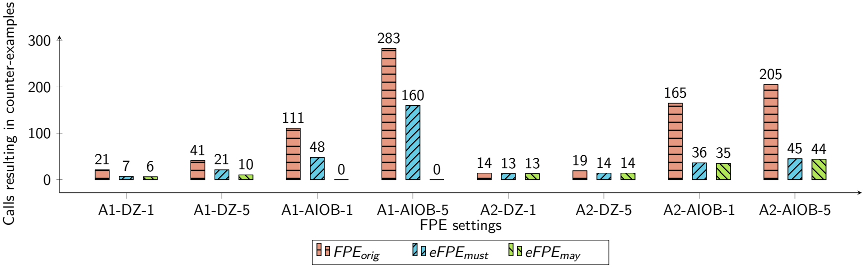
[2] *TCS Embedded Code Analyzer (TCS ECA)*. . http://www.tcs.com/offerings/engineering_services/Pages/TCS-Embedded-Code-Analyzer.aspx.

[3] CBMC. http://www.cprover.org/cbmc/.

Experimental Results

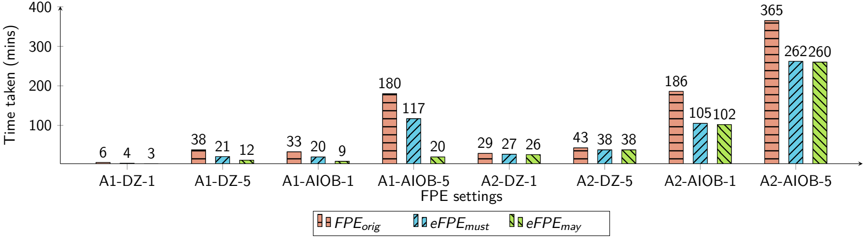


Total model checking calls

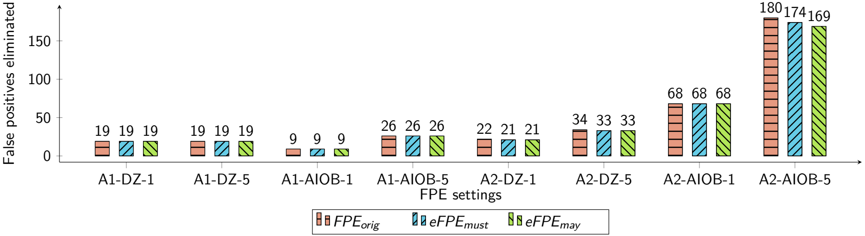


Model checking calls resulting in counter-examples

Experimental Results



False positives elimination time



False positives eliminated by FPE settings

Observations

- RVCs identification
  - 76% recall with 97.3% precision
  - Total model checking calls reduced by 49.49%
  - Calls resulting in counterexamples
    - original-FPE=58%, efficient-FPE=31%
- False positives elimination time reduced by 39.28%
  - Missed elimination of 21/754 false positives
- Trade-off:
  - Efficiency 39.28% Vs Elimination loss 2.78%
  - Failed cases require manual reviewing
- Constraining over One Vs All assertion variables
  - both choices applicable in practice

Summary

- The problem
  - large number of model checking calls
  - poor performance in false positives elimination
- Our Solution
  - Introduced a concept of *cnv* variables
  - Identification of redundant verification calls
- Experimental evaluation
  - Trade-off: Efficiency (39.28%) Vs Elimination loss (2.78%)
  - Spectrum of trade-offs
    - May Vs Must *cnv* variables
    - Single Vs Multiple variables
  - Choice depending on requirement in practice

Thank you!

Questions or Suggestions?

at t.muske@tcs.com