Dynamic errors

by
Ramraj S ME.,
Assistant Professor
Department of Software Engineering

16 - Mar -2016

Agenda

- Static analysis vs Dynamic analysis
- Dynamic errors
- challenges in analysis
- PREfix - tool support
- PREfix analysis

Static analysis vs Dynamic analysis

- Static Analysis is analysis done on source code without actually executing it.E.g., Syntax errors are caught by static analysis.
- Dynamic Analysis is analysis done as a program is executing and is based on intermediate values that result from the programs execution.E.g., A division by 0 error is caught by dynamic analysis.

Current static analysis methods are inadequate for:

- Dead Variables: Detecting unreachable variables is unsolvable in the general case.
- Arrays: Dynamically allocated arrays contain garbage unless they are initialized explicitly.

Dynamic errors

- Invalid pointer reference
- Fault storage location
- use of unintialized memory
- improper operation on resources

- c,c++ - 90 percentage of these kind errors are due to interaction of multiple functions
- compilers - only static errors, no memory leaks
- Lint - static analyzer
  - only with in procedure
  - cross function errors are not reported
- Annotation checkers - LCLint, Aspect, Extended, Static Checker - Need anotation from user for verification
- Abstract Interpretation - not feasable for large size of programs
- purify - debugging tool - work with test case

challenges for analysis

- False Negatives
  - Looking only in one function and miss errors across functions.
- False Positives
  - Reporting errors that cant really occur
  - Engineering effort (e.g. ESC/Java)
  - Requiring extensive program specifications
  - Execution overhead
  - Monitoring program may be impractical
  - Only as good as your test suite

PREfix

An error detection tool for C and C++, called PREfix, was built based on simulation technique.

- Handle hard aspects of C-like languages - Pointers, arrays, unions, libraries, casts
- Don't require user annotations- Build on language semantics
- Avoid false positives - Use path-sensitive analysis
- Give the user good feedback - Why might an error occur? Show the user an example execution

PREfix Analysis

- Explore paths through function
- For each path:
- Symbolically execute path - Determine facts true along the path
- Compute a guard- What must be true for the path to be taken
- Compute constraints- Preconditions for successful execution of path
- Compute result - What is true of the return value?

```
char *f(int size) {
    char * ptr;
    if (size > 0)
        ptr=(char*)malloc(size);
    if (size == 1)
        return NULL;
    ptr[0] = 0;
    return ptr;
}
```

```
f (param size)
alternate 0
    guard size <= 0
    constraint initialized(size)
        ARRAY ACCESS ERROR: ptr not initialized
alternate 1
    guard size == 1
    constraint initialized(size)
    fact ptr==memory_new(size)
    result return==NULL
        MEMORY LEAK ERROR:
        memory pointed to by ptr is not reachable
        through externally visible state
alternate 2
    guard size > 1
    constraint initialized(size)
    fact ptr==NULL
        ARRAY ACCESS ERROR: ptr is NULL
alternate 3
    guard size > 1
    constraint initialized(size)
    fact ptr==memory_new(size)
    fact ptr[0] == 0
    result return == memory_new(size)  && return[0] == 0
alternate 4...
```

Figure : Analysis

Interprocedural

```
void exercise_deref() {
  int v = 5;
   int x = deref(&v);
   int y = deref(NULL);
   int z = deref((int *) 5);
}
```

- Are there errors in this code?

```
int deref(int *p) {
    if (p == NULL)
        return NULL;
    return *p;
}
```

- Begin

deref (param p)

```
int deref(int *p) {
    if (p == NULL)
        return NULL;
    return *p;
}
```

- Use of p

deref (param p)

constraint initialized(p)

Figure :

```
int deref(int *p) {
    if (p == NULL)
        return NULL;
    return *p;
}
```

- Split path on value of p

```
deref (param p)
    alternate return_0
        guard p==NULL
        constraint initialized(p)
```

```
int deref(int *p) {
    if (p == NULL)
        return NULL;
    return *p;
}
```

- Return statement

```
deref (param p)
    alternate return_0
        guard p==NULL
        constraint initialized(p)
        result return==NULL
```

```
int deref(int *p) {
    if (p == NULL)
        return NULL;
    return *p;
}
```

- Consider other path

```
deref (param p)
    alternate return_0
        guard p==NULL
        constraint initialized(p)
        result return==NULL
    alternate return_X
        guard p != NULL
        constraint initialized(p)
```

```
int deref(int *p) {
    if (p == NULL)
        return NULL;
    return *p;
}
```

- Dereference of p

```
deref (param p)
    alternate return_0
        guard p==NULL
        constraint initialized(p)
        result return==NULL
    alternate return_X
        guard p != NULL
        constraint initialized(p)
        constraint valid_ptr(p)
```

References I

[1] Frank Buechner *"TestCaseDesignUsingTheClassificationTreeMethod "*

Thank you