MATLAB EXPO 2016
KOREA
4월 28일 (목)
등록 하기 matlabexpo.co.kr
Static Analysis in C/C++ code with Polyspace

Yongchool Ryu
Application Engineer
gary.ryu@mathworks.com
Agenda

- Efficient way to find problems in Software
- Category of Static Analysis
- Code Verification with Polyspace
- Q&A
The Lifecycle:

- Remember ISO 26262 and the implied waterfall lifecycle & V-Model
- DO-178, and the implied software lifecycles, V-Model, Spiral & Waterfall
- Perhaps you’ve adopted “[Fr]Agile” methods

- Where does Static Analysis fit?
Barry Boehm’s Top 10 List of Software Defect Reduction*

1. Finding and fixing a software problem after delivery is often **100 times more expensive** than finding and fixing it during the requirements and design phase.

6. **Peer reviews** catch **60 percent of the defects**.

7. **Perspective-based reviews** catch **35 percent more defects than nondirected reviews**.

8. **Disciplined personal practices** can reduce defect introduction rates by **up to 75 percent**.

9. **About 40 to 50 percent of user programs** contain nontrivial defects.

---

The Spiral model described by Barry Boehm
Software Quality Observations From Capers Jones*

SOURCES OF QUALITY DATA

Data collected from 1984 through 2013

- About 675 companies (150 clients in Fortune 500 set)
- About 35 government/military groups
- About 13,500 total projects
- New data = about 50-75 projects per month
- Data collected from 24 countries
- Observations during more than 15 lawsuits

* Capers Jones, CTO of Namcook Analytics LLC, the presentation of Software Quality in 2013
Match efficiency in Finding Bugs

- Which method is the best match for the first efficiency graph?

1. Individual Programmers
2. Normal Test Steps
3. Static Analysis
4. Design Reviews/Code Inspections

1. 95%
2. 90%
3. 75%
4. 50%
Efficiency in Finding Bugs

- Static analysis, Inspections and testing is best

1. 95% Static Analysis
2. 90% Design Reviews/Code Inspections
3. 75% Normal Test Steps
4. 50% Individual Programmers
Quality Measurements Have Found:

SOFTWARE QUALITY OBSERVATIONS

Quality Measurements Have Found:

- Individual programmers -- Less than 50% efficient in finding bugs in their own software

- Normal test steps -- often less than 75% efficient (1 of 4 bugs remain)

- Design Reviews and Code Inspections -- often more than 65% efficient; have topped 90%

- Static analysis –often more than 65% efficient; has topped 95%

- Inspections, static analysis, and testing combined lower costs and schedules by > 20%; lower total cost of ownership (TCO) by > 45%.
How Quality Affects Software Costs

*Capers Jones*, CTO of Namcook Analytics LLC, the presentation of Software Quality in 2013
Defects affect Software Quality and Productivity

SOFTWARE QUALITY AND PRODUCTIVITY

- The most effective way of improving software productivity and shortening project schedules is to reduce defect levels.

- Defect reduction can occur through:

  1. **Defect prevention technologies**
     - Structured design and JAD
     - Structured code
     - Use of inspections, static analysis
     - Reuse of certified components

  2. **Defect removal technologies**
     - Design inspections
     - Code inspections, static analysis
     - Formal Testing using mathematical test case design
Efficient way to find problems in Software

1. **Defect prevention technologies**
   - Structured design and JAD
   - Structured code
   - Use of inspections, static analysis
   - Reuse of certified components

2. **Defect removal technologies**
   - Design inspections
   - Code inspections, static analysis
   - Formal Testing using mathematical test case design

Fix Earlier, reduce cost!

Software Development Lifecycle Phase
Source: E.Books and Y. Bittii: Software Defect Reduction Top '10 List”, IEEE Computer

- Fix Earlier, reduce cost!
- $1,102.00
- $1,136.00
- $977.00
- $455.00
- $139.00
- $0.00

Cost Per Bug

© Static Analysis

1. 65%
   - 95%
   - 100%
CATEGORY OF STATIC ANALYSIS OF USING TOOLS

- Compiler Warnings
- Bug Findings (False negative)
- Coding Rules, Code Metrics
- Formal Methods (No False negative)
- Error Prevention
- Error Detection
Types of bugs detected by Polyspace Bug Finder

Numerical
- Zero divide, overflow, shift
- Integer and float conversion overflow
- Invalid use of std. library math routine
- ...

Static memory
- Array access out of bounds
- Null pointer
- ...

Concurrency
- Data races (atomic, non-atomic)
- Deadlocks
- ...

Dynamic memory
- Memory leaks
- Use of previously freed pointer
- Unprotected dynamic memory allocation
- ...

Programming
- Invalid use of = or == operator
- Declaration mismatch
- ...

Dataflow
- Write without further read
- Non-initialized variable
- ...

Language support
- C
- C++

Full list of run-time checks in Polyspace Code Prover

**C run-time checks**
- Unreachable Code
- Out of Bounds Array Index
- Division by Zero
- Non-Initialized Variable
- Scalar and Float Overflow (left shift on signed variables, float underflow versus values near zero)
- Initialized Return Value
- Shift Operations (shift amount in 0..31/0..63, left operand of left shift is negative)
- Illegal Dereferenced Pointer (illegal pointer access to variable of structure field, pointer within bounds)
- Correctness Condition (array conversion must not extend range, function pointer does not point to a valid function)
- Non-Initialized Pointer
- User Assertion
- Non-Termination of Call (non-termination of calls and loops, arithmetic expressions)
- Known Non-Termination of Call
- Non-Termination of Loop
- Standard Library Function Call
- Absolute Address
- Inspection Points

**C++ run-time checks**
- Unreachable Code
- Out of Bounds Array Index
- Division by Zero
- Non-Initialized Variable
- Scalar and Float Overflow
- Shift Operations
- Pointer of function Not Null
- Function Returns a Value
- Illegal Dereferenced Pointer
- Correctness Condition
- Non-Initialized Pointer
- Exception Handling (calls to throws, destructor or delete throws, main/tasks/C_lib_func throws, exception raised is not specified in the throw list, throw during catch parameter construction, continue execution in __except)
- User Assertion
- Object Oriented Programming (invalid pointer to member, call of pure virtual function, incorrect type for this-pointer)
- Non-Termination of Call
- Non Termination of Loop
- Absolute Address
- Potential Call
- C++ Specific Checks (positive array size, incorrect typeid argument, incorrect dynamic_cast on reference)

www.mathworks.com/help/codeprover/results.html
Not all bugs can be statically proven

- **All Bugs**
  - e.g., divide by zero, overflow, illegal pointer dereferences
  - e.g., if(x=y) vs. if(x==y), memory leaks, partial array access

- **Statically Detectable**

- **Provable**

---

**Polyspace Bug Finder**

**Polyspace Code Prover**
How do Bug Finder results differ from Code Prover results?

**Bug Finder**

- **Nothing Found**
- **Probable Bug**

**Code Prover**

- **Green - Reliable**
- **Orange - Vulnerability**
- **Grey - Unreachable / Dead**
- **Red - Faulty**

**Purple - coding rule violations**
To prove the absence of errors, the Polyspace verification accounts for all possible execution paths using abstract interpretation.

```c
signed char x, y;

x = random();

if (x > 0) {
    x = 5;
}
else if (x != 0) {
    y = 100 / x;
}

printf("%d", x);
```
Results from Polyspace Code Prover

- Start with C/C++ source code

```c
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }

    i = get_bus_status();

    if (i >= 0) {
        *(p - i) = 10;
    }
}
```
Results from Polyspace Code Prover

- Source code painted in **green**, **red**, **gray**, **orange**

```c
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }

    i = get_bus_status();

    if (i >= 0) {
        *(p - i) = 10;
    }
}
```
Results from Polyspace Code Prover

static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }

    i = get_bus_status();

    if (i >= 0) {
        *(p - i) = 10;
    }
}
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }

    i = get_bus_status();

    if (i >= 0) {
        *(p - i) = 10;
    }
}
Results from Polyspace Code Prover

```c
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }

    i = get_bus_status();

    if (i >= 0) {
        *(p - i) = 10;
    }
}```
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }

    i = get_bus_status();

    if (i >= 0) {
        *(p - i) = 10;
    }
}
Results from Polyspace Code Prover

```
static void pointer_arithmetic (void) {
  int array[100];
  int *p = array;
  int i;
  for (i = 0; i < 100; i++) {
    *p = 0;
    p++;
  }
  if (get_bus_status()>0) {
    if (get_oil_pressure()>0) {
      *p = 5;
    } else {
      i++;
    }
  }
  i = get_bus_status();
  if (i >= 0) {
    *(p - i) = 10;
  } else {
    (p - i) = 10;
  }
}
```
Results from Polyspace Code Prover

static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;
    for (i = 0; i < 100; i++) {
        *p = 0;
        p ++;
    }
    if (get_bus_status () > 0) {
        if (get_oil_pressure () > 0) {
            *p = 5;
        } else {
            i ++;
        }
    }
    i = get_bus_status () ;
    if (i >= 0) {
        * (p - i) = 10 ;
    }
}
signed char intoverflow(int val)
{
    signed char first;
    signed char second;
    signed int result;
    signed int result1;

    if (val < 10) {
        first = 20;
        second = 100;
    } else {
        first = 60;
        second = 100;
    }

    result = first + second;

    result1 = result + 1;

    result = (signed int)((signed int)first + (signed int)second)>>24;

    return (signed char)(result+result1);
}
Who should use the tools?

- **Specification**
  - **Software Engineers**
  - **Quality Assurance Engineers**

- **Design**
  - **Software Architects/Engineers**
  - **Quality Engineers**

- **Implementation**
  - **Software Engineers**

- **SW Unit Tests**
  - **Software Engineers**
  - **Quality engineers**

- **SW Acceptance tests**
  - **Quality Assurance Engineers**

- **SW Integration tests**
  - **Software Architects/Engineers**
  - **Quality Engineers**

Legend:
- **BF**: Bug Finder
- **CP**: Code Prover
Software Quality Objectives (SQO)

- Specify software quality levels in Polyspace
  - Identify when a file, module, or component achieves desired quality level

- Define customizable thresholds based on
  - Software metrics
  - Code rule violations
  - Number of red, gray, oranges

- Use SQO as a process guide
  - Practical plan for an incremental adoption of tools and process changes to meet quality objectives
Dashboard for management view

With top-level rollup, trends, and pass/fail objectives
Save time by using both Bug Finder and Code Prover

Improvements every 6 months

Fix Earlier, reduce cost!

EARLY & OFTEN!
Q & A
New features in R2015b and R2016a

- **Full support** of MISRA-C:2012 rules
- **MISRA 2012 Directives**
  - New MISRA 2012 Directives 4.5 and 4.13
  - Improve support of directive 4.3
- When you want the MISRA 2012 checker to be applied to C90 only, you can tick ‘Respect C90 Standard’
  - It may have some side effects on compilation

- Polyspace Code Prover:
  - MISRA C:2012 rules 22.1 to 22.4 and rule 22.6 are not supported
New features in R2015b and R2016a

- Polyspace Bug Finder defects: now 140 defects!
  - 81 new defects with new defect categories:
    - Programming,
    - C++,
    - Security,
    - Resources management
  - Additional:
    - Improved precision on memory leaks
**Use cases**

**Implementation**
- Find local bugs
- Find MISRA violations
- Find “un-testable” functions
- Perform Code Reviews

**Design**

**Specification**

**SW Acceptance tests**
- Measure SW quality
- Quality report generation

**SW Integration tests**
- Find integration bugs
  - Declaration mismatches
  - Data race on shared variables
  - Global variables usage

**SW Unit Tests**
- Quality gate
  - Find runtime errors / unused code
  - Prove absence of runtime errors on modules
  - Justify MISRA violations

**Legend**
- **BF**: Bug Finder
- **CP**: Code Prover