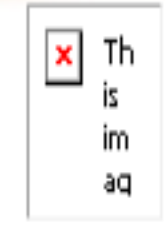
SE3910 – REAL TIME SYSTEMS

MISRA Coding Standards

INDUCTION VARIANCE

- For (I = 1; I <= 10; i++)
- {
  - A[i+1] = 0;
- }



LOOP UNROLLING

- Duplicate instructions executed in a loop to reduce the number of operations and hence the loop overhead incurred

```
For (I = 1; i<=6;i++)
{
a[i] = a[i]*8;
}
```

```
a[i] = a[i]*8;
i++;
a[i] = a[i]*8;
i++;
a[i] = a[i]*8;
i++;
```

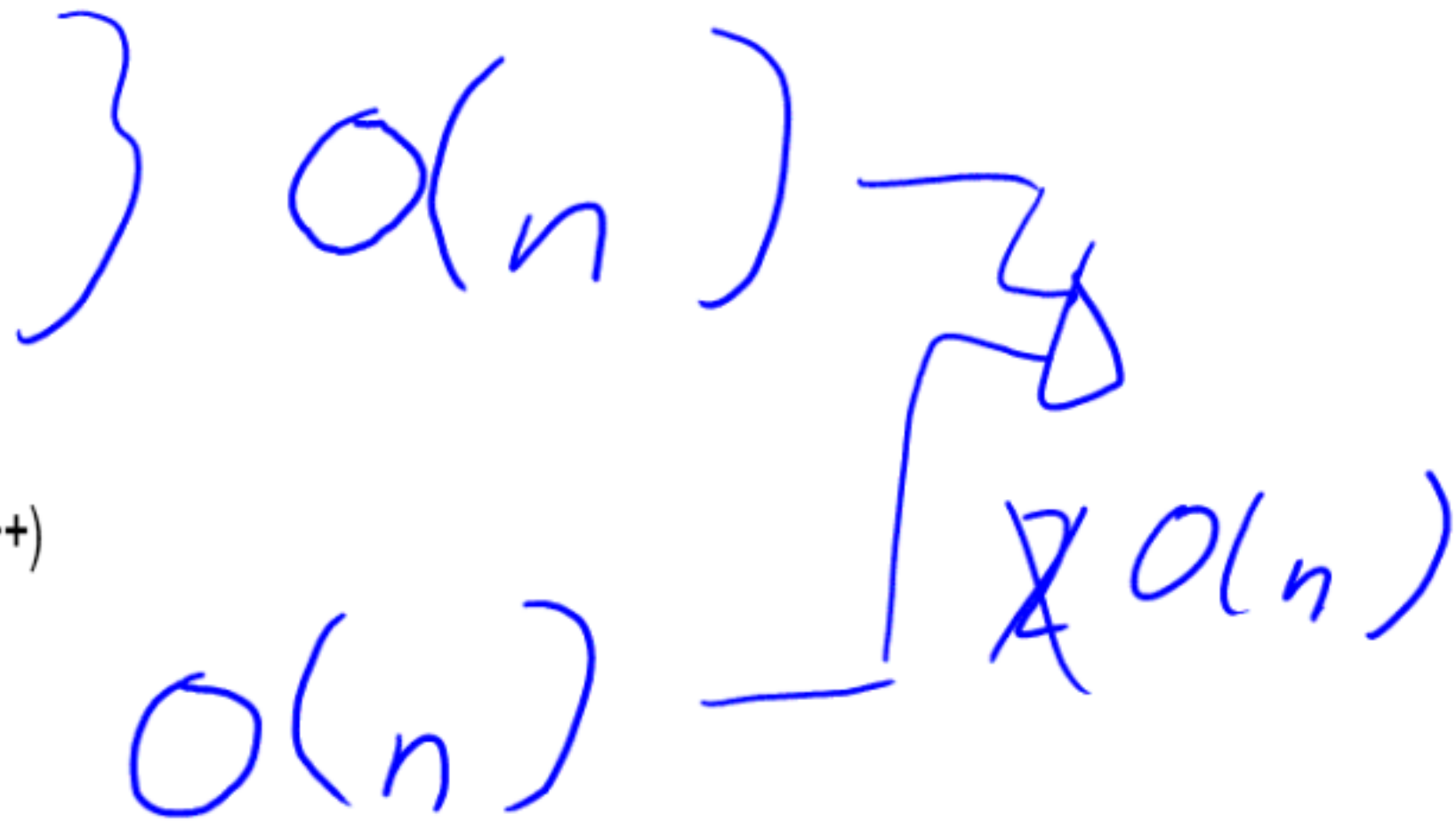
6x overall

High performance systems

- Combine similar loops together to improve performance

LOOP JAMMING

```
For (I = 1; i<=100;i++)
{
        X[i] = y[i]*8;
}
For (I = 1; I <= 100; i++)
{
        Z[i] = x[i] * y[i];
}
```



- Combine similar loops together to improve performance

```
For (I = 1; i<=100;i++)
{
        X[i] = y[i]*8;
}
For (I = 1; I <= 100; i++)
{
        Z[i] = x[i] * y[i];
}
```

LOOP JAMMING

Z[i] = X[i] * y[i];

Removed

ROADMAP

- Today
  - Embedded Code Quality and MISRA
- Wednesday
  - Real Time Software Qualities
- Friday
  - Structured Design and Data Flow Diagrams

OBJECTIVES

- Understand the difference between static analysis and testing
- Define the halting problem
- Explain the difference between a false positive and a false negative
- Construct a primitive static analysis tool using grep
- Describe the impact of using static analysis tools over time
- Compare and contrast style guides and programming standards
- Explain the steps necessary to integrate static analysis into a development process

OBJECTIVES

- New code
- Legacy code

Static Analysis

Look @ code w/out executing

- w/a tool

Checkstyle PMD FindBugs

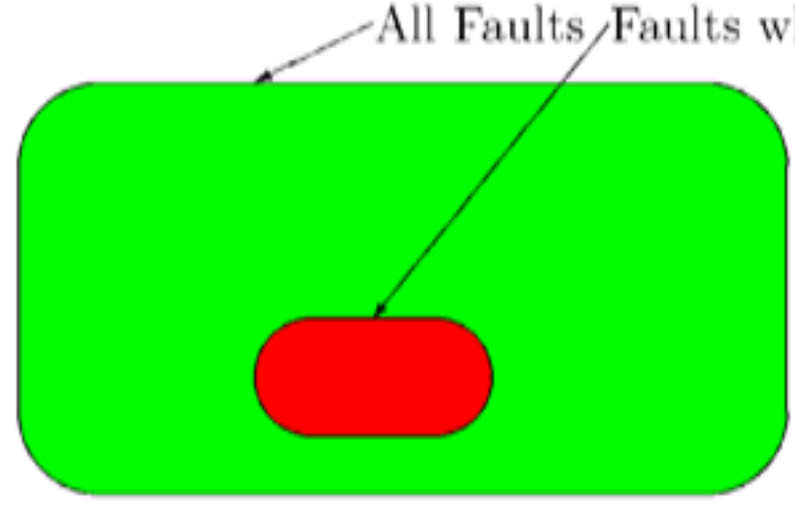
Java

INTRODUCTION

- Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [IEEE]
  - Does not involve the execution of the program
  - Software inspections are a form of static analysis
- "even well tested code written by experts contains a surprising number of obvious bugs" [Hovermeyer/Pugh]
- "Java has many language features and APIs which are prone to misuse." [Hovermeyer/Pugh]
- Static analysis tools "can serve an important role in raising the awareness of developers about subtle correctness issues. . . . prevent future bugs" [Hovermeyer/Pugh]

STATIC ANALYSIS OVERVIEW

- Similar to a spell checker or grammar checker.
- Search through code to detect bug patterns
  - error prone coding practices that arise from the use of erroneous design patterns, misunderstanding of language semantics, or simple and common mistakes.
- Static Analysis tools detect faults
  - Not all faults will fail
    - 90% of downtime comes from 10% of the faults
- Can detect many different classifications of software faults
  - Coding standards violations
  - Buffer overflows (Viega et all)
  - Security vulnerabilities (Livshits and Lam)
  - Memory leaks (Rai)
  - Timing anomalies (race conditions, deadlocks, and livelocks) (Artho)



STATIC ANALYSIS OVERVIEW (2)

- Required to claim compliance with MISRA C Standard
- Between 40% and 60% of statically detectable faults will eventually manifest themselves in the field (QA Systems)
- Has been shown to reduce software defects by a factor of six (Xiao and Pham)
- Can remove upwards of 91% of errors (R. Glass)
- Have been shown to have a 92% ROI_{time} (Schilling)
- NEW: Required by the state of New York for contracted SW

STATIC ANALYSIS OVERVIEW (4)

- Impossible to prove a software program correct in the general case
  - Manifestation of the Halting Problem.

→ Function x [Functions]

- Most static analysis tools are unsound and incomplete.

→ Guarantees that f will always return.

STATIC ANALYSIS OVERVIEW (4)

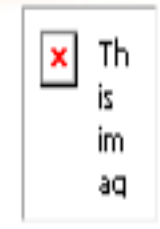
- Impossible to prove a software program correct in the general case
  - Manifestation of the Halting Problem.
- Most static analysis tools are unsound and incomplete.

False positives

False negative

STATIC ANALYSIS CLASSIFICATIONS

- General Purpose Tools
  - General purpose Static Analysis tools are those geared for general developmental usage
  - Lint, QAC, Polyspace C, JLint, Findbugs
- Security Tools
  - Static Analysis tools targeting security issues within source code
  - RATS (Rought Auditing Tool for Security), SPLint, Flawfinder
- Style Checking Tools
  - Audit software code from a stylistic standpoint ensuring consistant implementation style
  - PMD, Checkstyle
- Teaching Tools
  - Developed to help students develop better software

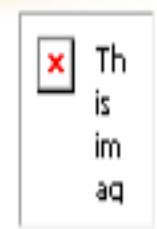


WHAT IS WRONG WITH THIS CODE?

(NOTE: THIS IS C)

```
 1: void example_routine( uint32_t x, uint32_t ok, BOOL wrong )
 2: {
 3:   if ( x = 1 )
 4:   {
 5:     printf( "X has a value of 1.\n" );
 6:   }
 7:
 8:   if ( ok == 2 )
 9:   {
10:     printf( "OK has a value of 2.\n" );
11:   }
12:
13:   if ( wrong = FALSE )
14:   {
15:     printf( "You are correct!\n" );
16:   }
17:
18:   /* if (commented=TRUE) Even though this code is commented out,
19:      the error is still shown. */
20: }
```

Assignment instead of Comparison



SIMPLE "HOME MADE" STATIC ANALYSIS TOOL USING GREP

```
 1: void example_routine( uint32_t x, uint32_t ok, BOOL wrong )
 2: {
 3:   if ( x = 1 )
 4:   {
 5:     printf( "X has a value of 1.\n" );
 6:   }
 7:
 8:   if ( ok == 2 )
 9:   {
10:     printf( "OK has a value of 2.\n" );
11:   }
12:
13:   if ( wrong = FALSE )
14:   {
15:     printf( "You are correct!\n" );
16:   }
17:
18:   /* if (commented=TRUE) Even though this code is commented out,
19:      the error is still shown. */
20: }
```

```
$ grep "if ([[:space:]]*[[:alnum:]]*[[:space:]]*=[[:space:]]*
[[:alnum:]]" error_files.c
```

SIMPLE "HOME MADE" STATIC ANALYSIS TOOL USING GREP

```
 1: void example_routine( uint32_t x, uint32_t ok, BOOL wrong )
 2: {
 3:   if ( x = 1 )
 4:   {
 5:     printf( "X has a value of 1.\n" );
 6:   }
 7:
 8:   if ( ok == 2 )
 9:   {
10:     printf( "OK has a value of 2.\n" );
11:   }
12:
13:   if ( wrong = FALSE )
14:   {
15:     printf( "You are correct!\n" );
16:   }
17:
18:   /* if (commented=TRUE) Even though this code is commented out,
19:      the error is still shown. */
20: }
```

```
$ grep "if ([[:space:]]*[[:alnum:]]*[[:space:]]*[[:space:]]*
[[:alnum:]]" error_files.c
  if ( x = 1 )
  if ( wrong = FALSE )
  /* if (commented=TRUE) Even though this code is commented out,
[wws@localhost wws]$
```

- One of the oldest and readily available static analysis tools
  - Developed initially by Bell Labs
    - C Language
    - UNIX development
    - Now available for Dos, Windows, Linux, OS/2
- Commercial version available from Gimpel Software
  - Supports value tracking, MISRA C standard compliance verification, and Effective C++ Standards
  - Analyzes C and C++ code

SAMPLE BUFFER OVERFLOW FAILURE SOURCE CODE (C LANGUAGE)

```
1:  typedef unsigned short uint16_t;
2:  void update_average(uint16_t current_value);
3:
4:  #define NUMBER_OF_VALUES_TO_AVERAGE (11u)
5:
6:  static uint16_t data_values[NUMBER_OF_VALUES_TO_AVERAGE];
7:  static uint16_t average = 0u;
8:
9:  void update_average(uint16_t current_value)
10: {
11:   static uint16_t array_offset = 0u;
12:   static uint16_t data_sums = 0u;
13:
14:   array_offset = ((array_offset++) % NUMBER_OF_VALUES_TO_AVERAGE);
15:   data_sums -= data_values[array_offset];
16:   data_sums += current_value;
17:   average = (data_sums / NUMBER_OF_VALUES_TO_AVERAGE);
18:   data_values[array_offset] = current_value;
19: }
```

SAMPLE BUFFER OVERFLOW FAILURE SOURCE CODE (C LANGUAGE)

```
1:  typedef unsigned  short uint16_t;
2:  void update_average(uint16_t  current_value);
3:
4:  #define NUMBER_OF_VALUES_TO_AVERAGE  (11u)
5:
6:  static  uint16_t  data_values[NUMBER_OF_VALUES_TO_AVERAGE];
7:  static  uint16_t  average = 0u;
8:
9:  void update_average(uint16_t  current_value)
10: {
11:   static  uint16_t  array_offset  = 0u;
12:   static  uint16_t  data_sums  = 0u;
13:
14:   array_offset  = ((array_offset++)  % NUMBER_OF_VALUES_TO_AVERAGE);
15:   data_sums  -= data_values[array_offset];
16:   data_sums  += current_value;
17:   average  = (data_sums  / NUMBER_OF_VALUES_TO_AVERAGE);
18:   data_values[array_offset]  = current_value;
19: }
```

SAMPLE BUFFER OVERFLOW FAILURE

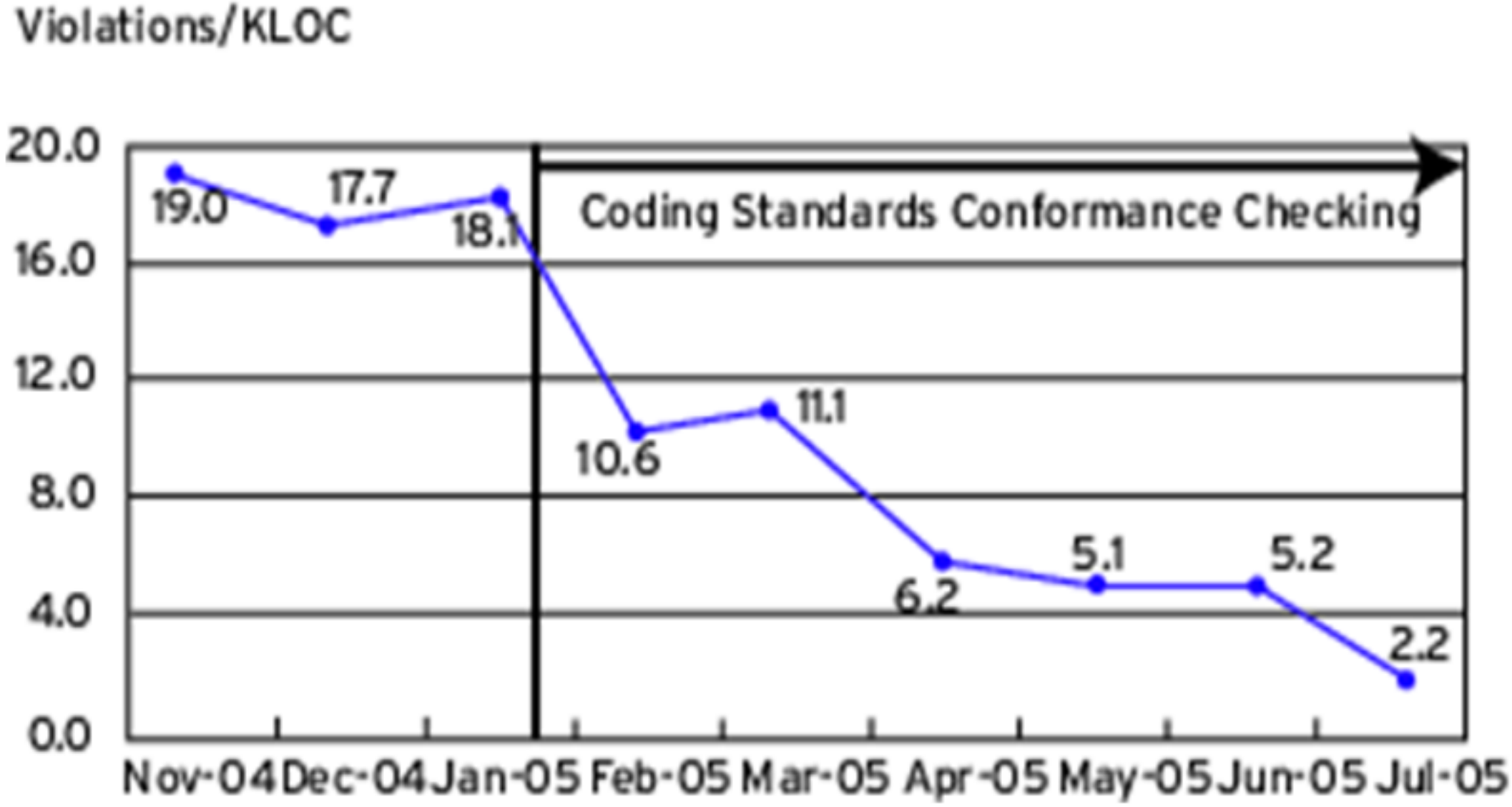
LINT OUTPUT

```
--- Module: buffer_overflow.c

array_offset = ((array_offset++) %
   NUMBER_OF_VALUES_TO_AVERAGE);


_
"*** \index{LINT}LINT: buffer_overflow.c(14) Warning 564:
   variable 'array_offset' depends on order of evaluation
   [\index{MISRA C}MISRA Rule 46]"
```

- Fault manifesting itself as a failure depends upon the compiler's handling of source code!
  - Some compilers may handle code properly.
  - Other compilers may cause failure to occur.
  - Compiler options may effect behavior.
    - Especially true of optimization flags

IMPACT OF SA OVER TIME

(DR. DOBBS, *JUNE 16, 2006* **CODE QUALITY IMPROVEMENT**)



ADDING SA TO DEVELOPMENT PROCESS

- 1. Develop a coding standard and style guides
  - Style guide is not a necessity to use SA effectively
  - There may be multiple style guides
- 2. Automate compliance checking with the standard
- 3. Add SA Compliance checking to review process

STYLE GUIDES

- Provides stylistic guidance for developing source-code modules.
- Items to define include:
  - Copyright notices
  - requisite commenting
  - Indentation
  - naming conventions
  - Any other stylistic items
- Can raise significant debate amongst software engineers
- Can be automated by providing templates to automatically format code in conformance with the style guide
  - Eclipse, JEdit, CodeWright all support style templates.

CODING STANDARD

- Defines which coding constructs can and can not be used in a project.
  - Should predominantly be enforceable through static analysis methods
  - Should include general best practices as well as past experiences within the domain

- Example rule:
  - *"All variables shall be assigned a value before being used in any operation"*
  - Statically detectable
  - Can be easily understood by a programmer.

- Defined deviation procedure
  - With every coding standard, there will be a need for an occasional deviation.
  - All deviations should be reviewed in a formal setting (peer review, formal review, walkthrough, etc.)

- Standards Exist to use as a baseline
  - MISRA C
  - High Integrity C++

- Tabs should be used as the correct method of indentation. Tabs should be 4 spaces in width.
- Avoid lines longer than 80 characters as these cause problems on smaller displays and terminals.
- Lines that must wrap should be broken only at the following points:
  - After a comma
  - Before an operator
  - Prefer higher-level breaks over lower-level breaks
  - Align the new line with the beginning of the expression at the same level on the previous line.
  - If all else fails, use an indent of 8 spaces.

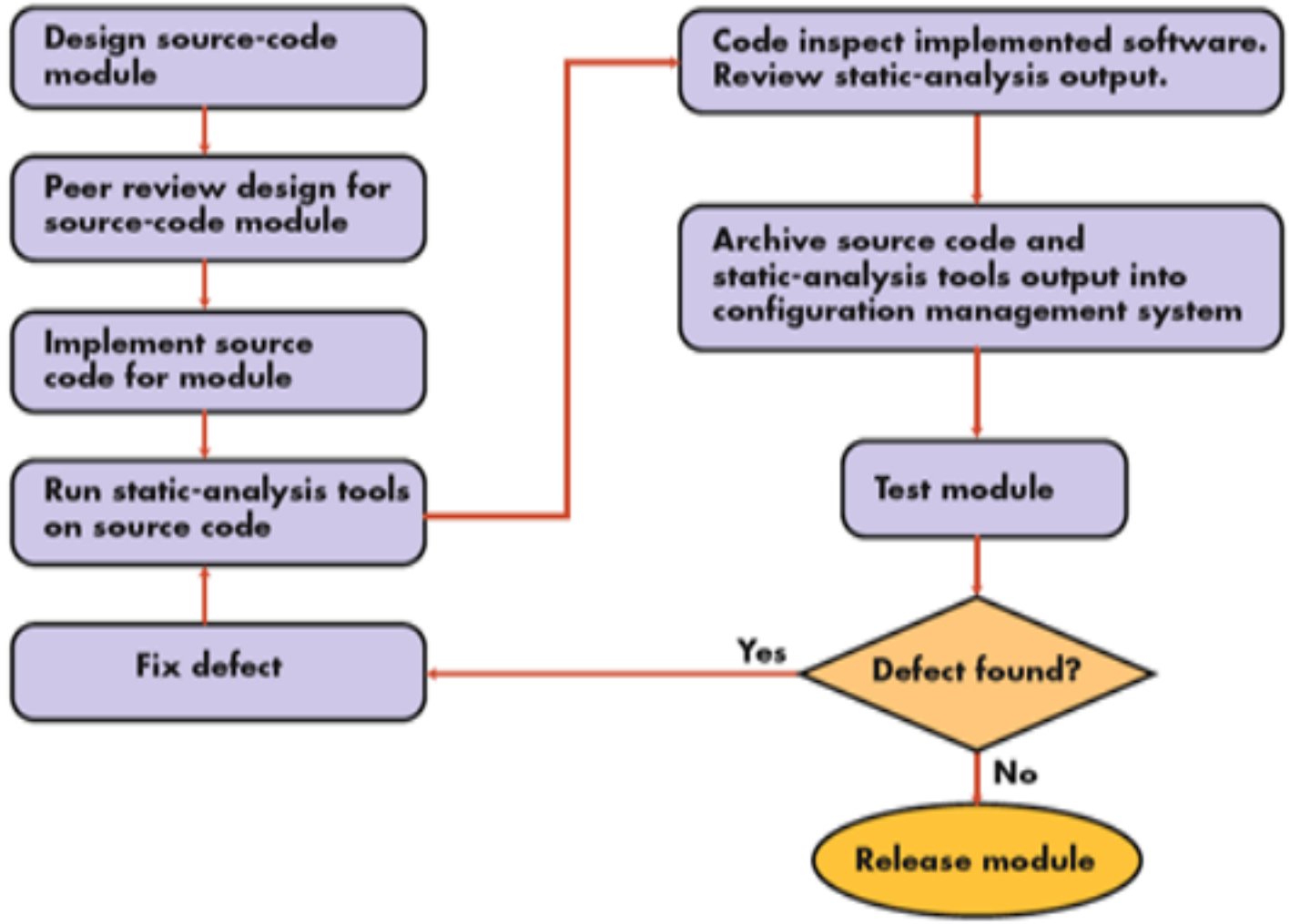
- Two blank lines should be added between:
  - Sections of a source file.
  - Class and interface definitions.
- One blank line should be added between:
  - Methods.
  - Local variables in a method and its first statement.
  - Before a block or single-line comment.
  - Between logical sections inside a method to improve readability.
- A blank space should be added between:
  - A keyword (such as if, while, for) and its opening parentheses. (Not method names!)

MISRA C / C++

- Misra C/C++ (Motor Industry Software Reliability Association)
- Rules first introduced in 1998
- Revised in 2004: 141 rules for C
- Revised to cover C++ in 2008 (mostly derived from JSF rules): 228 rules
- Widely used in motor vehicle industry
- Some support in popular embedded compilers
- Closed standard

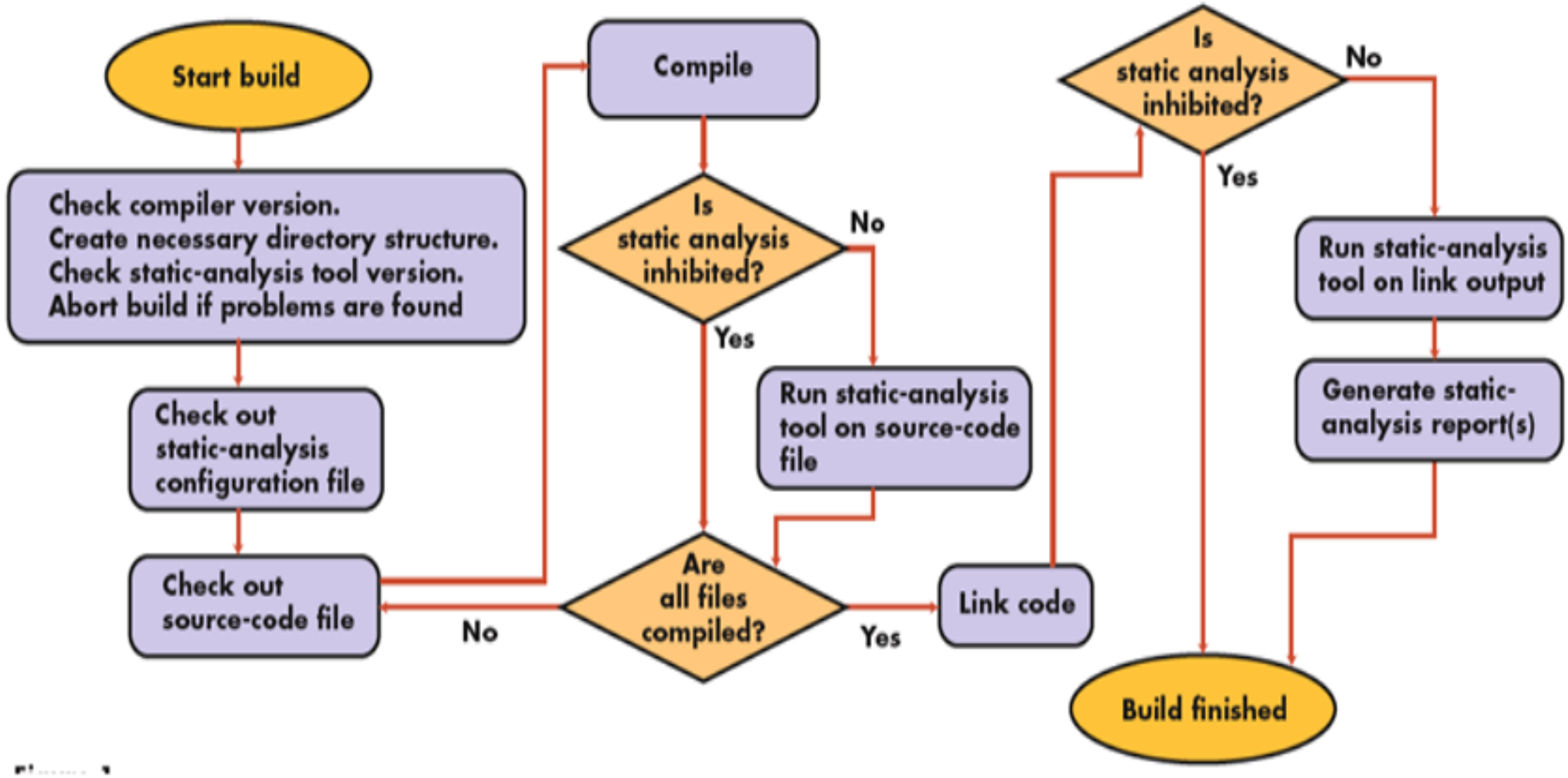
SW DEVELOPMENT PROCESS INCORPORATING STATIC ANALYSIS

This software-development process segment incorporates static analysis



AUTOMATE CODING STANDARD COMPLIANCE

Sample flowchart of an automated build script incorporating static analysis



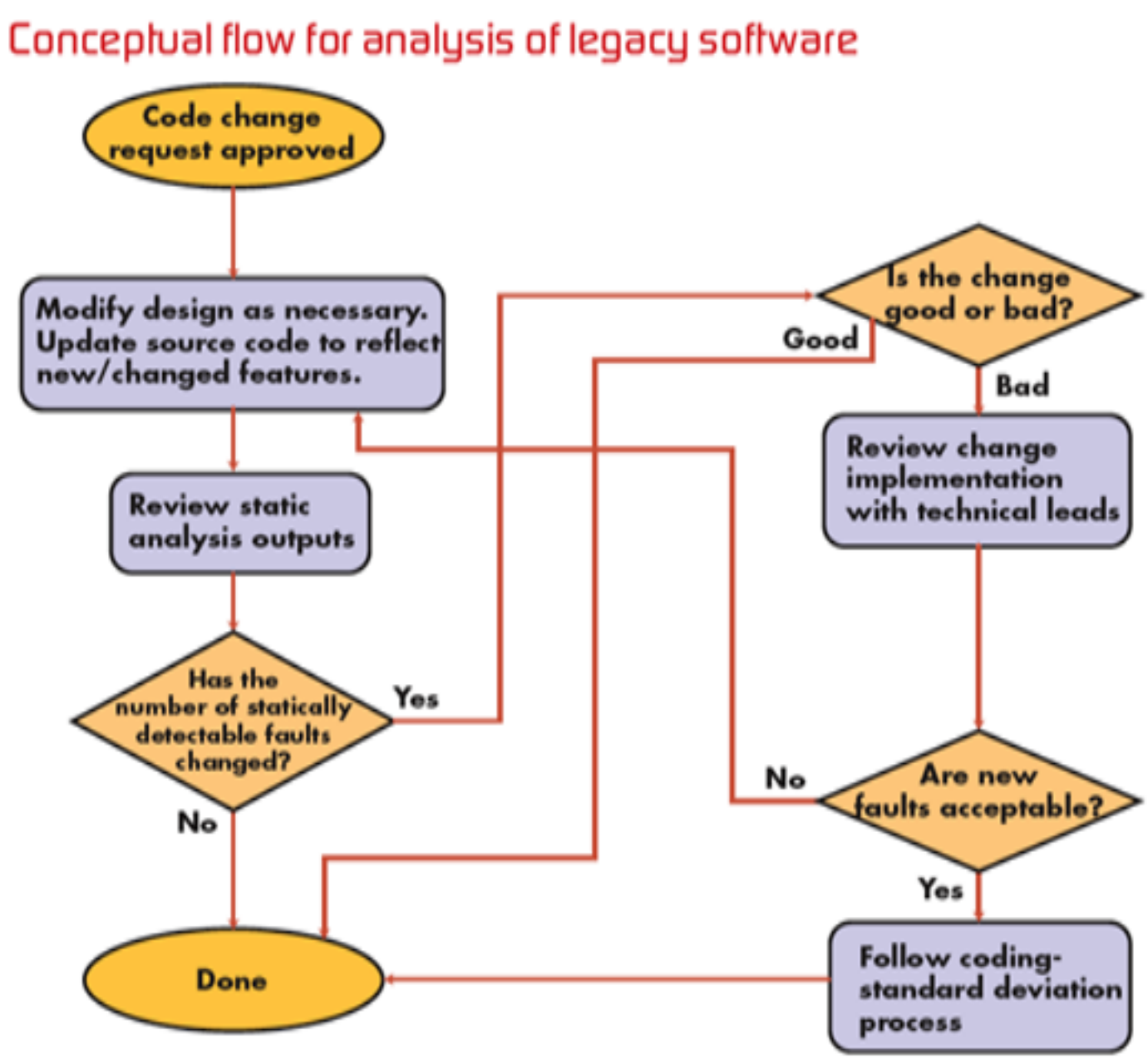
LEGACY CODE INTEGRATION

- Applying to existing code base can be challenging
- Success depends upon
  - age of the code
  - engineer's programming style
  - paradigms used
  - Diligence of engineers applying static analysis
- Many projects have abandoned SA when the first run of the tool generates 100,000 or more warnings.

- Treat each statically detectable fault as a bug fix.
  - Each time a fault is removed, there's the possibility of injecting a more serious fault into the module.
  - The worst thing would be to attempt to repair a false-positive that was statically detected as a fault and inject a failure.
  - This must be done diligently, as each Statically Detectable fault could be a catastronphic failure in the making
    - Ariane V

- With legacy code, the most important information to track isn't necessarily the presence of statically detectable faults, but the change in the number of faults as revisions are made to that code.

LEGACY SOFTWARE FLOWCHART



RESOURCES

- Integrate static analysis into a software development process
  - http://www.embedded.com/shared/printableArticle.jhtml?articleID=193500830
- NIST SAMATE - Software Assurance Metrics And Tool Evaluation Project
  - http://samate.nist.gov/index.php/Main_Page
- Static Source Code Analysis Tools for C
  - http://www.spinroot.com/static/