SE3910 – REAL TIME SYSTEMS

ROADMAP

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

OBJECTIVES

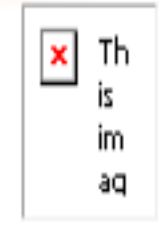
- Understand the difference between static analysis and testing
- Define the halting problem
- Explain the difference between a false positive and a false negative
- Construct a primitive static analysis tool using grep
- Describe the impact of using static analysis tools over time
- Compare and contrast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

- New code
- Legacy code

INTRODUCTION

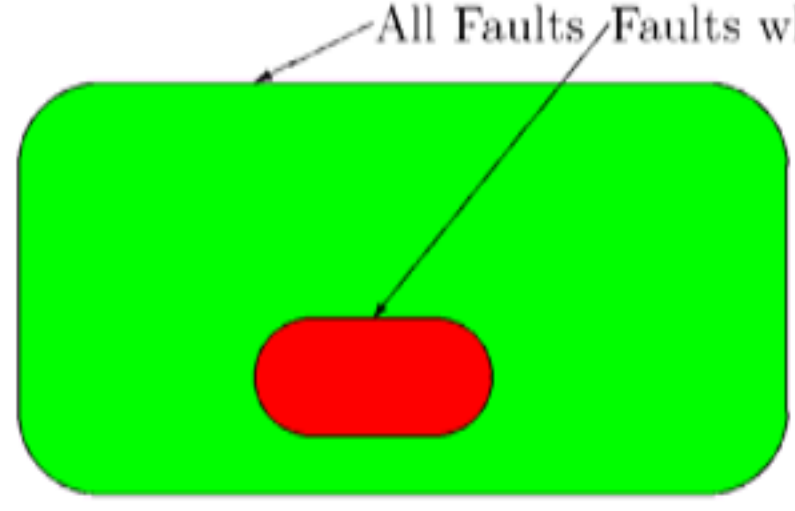
- Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [IEEE]
  - Does not involve the execution of the program
  - Software inspections are a form of static analysis

- "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)

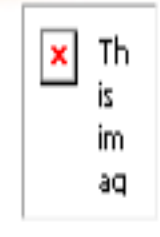


STATIC ANALYSIS OVERVIEW (2)

- 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

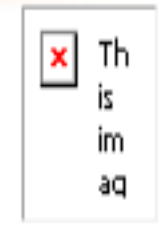
STATIC ANALYSIS OVERVIEW (4)

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



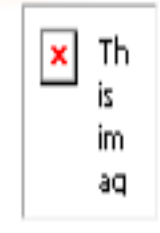
STATIC ANALYSIS CLASSIFICATIONS

- 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: }
```

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
```

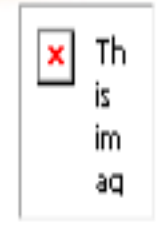
SIMPLE "HOME MADE" STATIC ANALYSIS TOOL USING GREP

```
 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
    - ALOA metrics tool is available to collect quality metrics from Lint tool.
    - XML Output readily available

LINT



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: }
```

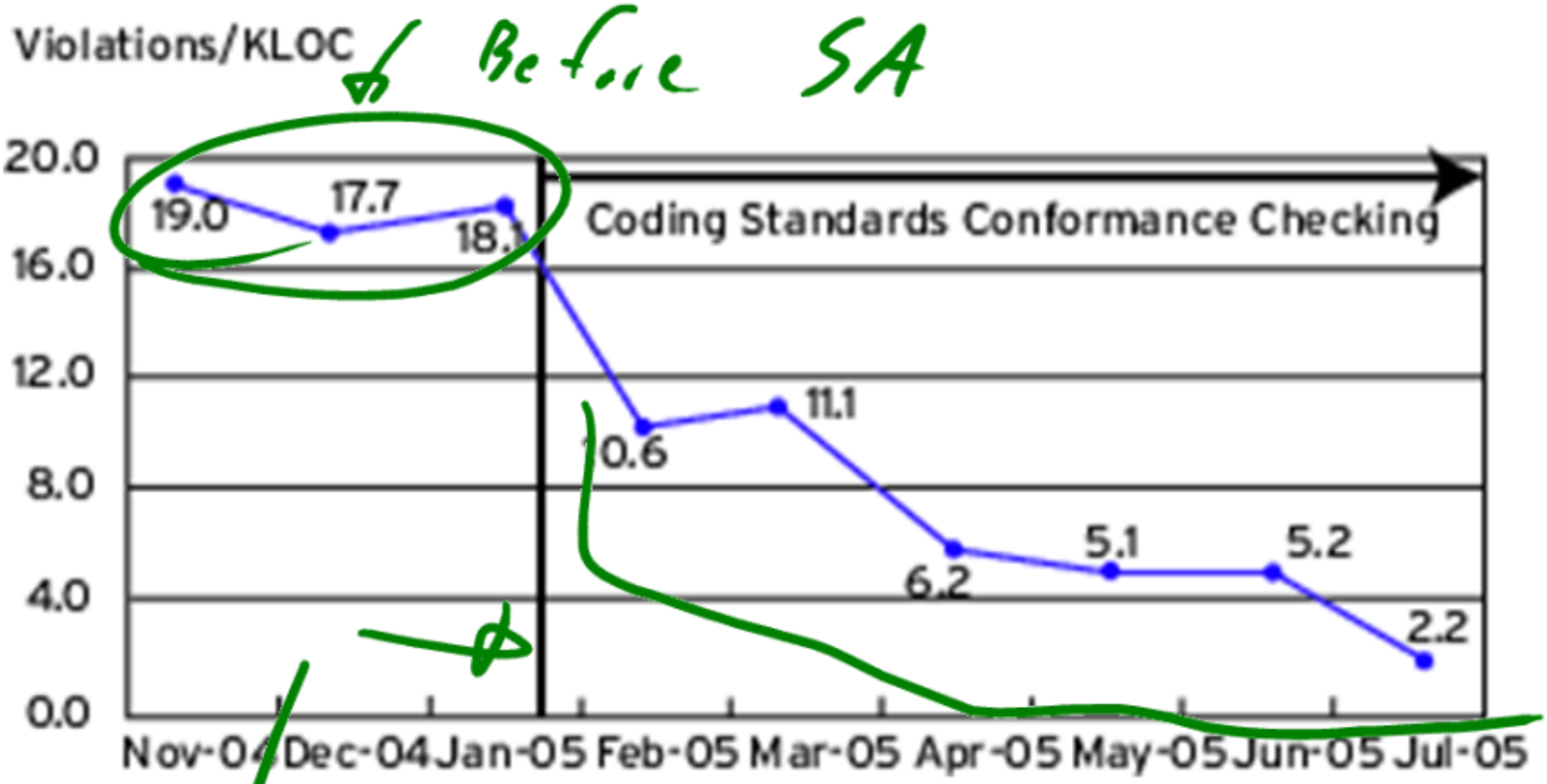
SAMPLE BUFFER OVERFLOW FAILURE
LINT OUTPUT

```
--- 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



Before SA

After

Huge reduction

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!)
  - Comma-separated arguments in a list
  - All binary operators (except ".") ; Unary operators should never be separated.
  - The expressions in a "for" statement, including the for-each version.
  - A typecast and the variable name it affects.

MISRA C AND MISRA C++

SE3910 REAL TIME SYSTEMS

Th
is
im
aq

MISRA C / C++ BACKGROUND

- Guidelines discuss general problems in software engineering
  - note C and C++ do not have as much error checking as other languages do.
- MISRA C
  - Subset of the C language.
  - Based on the ISO/IEC 9899:1990 C standard
  - Every MISRA C program is a valid C program.
  - 141 rules that constrain the C language.
- MISRA C++
  - Subset of the ISO/IEC 14882:2003 C++ standard.
  - 228 rules

EXAMPLE RULES

Rule 0–3–2	(Required)	If a function generates error information, then that error information shall be tested.

Rationale

A function (whether it is part of the standard library, a third party library or a user defined function) may provide some means of indicating the occurrence of an error. This may be via a global error flag, a parametric error flag, a special return value or some other means. Whenever such a mechanism is provided by a function the calling program shall check for the indication of an error as soon as the function returns.

Note, however, that the checking of input values to functions is considered a more robust means of error prevention than trying to detect errors after the function has completed.

Example

```
extern void fn3 ( int32_t i, bool & flag );

int32_t fn1 ( int32_t i )
{
   int32_t result  = 0;
   bool  success = false;

   fn3 ( i, success );      // Non-compliant - success not checked
   return result;
}

int32_t fn2 ( int32_t i )
{
   int32_t result  = 0;
   bool  success = false;

   fn3 ( i, success );      // Compliant - success checked
   if ( !success )
   {
      throw 42;
   }
   return result;
}
```

See also

Rule 19–3–1

(void) doit(5);

I am consciously ignoring the return value.

EXAMPLE RULES

6.2.7 Comments

Rule 2–7–1	(Required)	The character sequence /* shall not be used within a C-style comment.

Rationale

C++ does not support the nesting of C-style comments even though some compilers support this as a non-portable language extension. A comment beginning with /* continues until the first */ is encountered. Any /* occurring inside a comment is a violation of this rule.

Example

Consider the following code fragment:

```
/* some comment, end comment marker accidentally omitted
Perform_Critical_Safety_Function(X);
/* this "comment" is Non-compliant */
```

In reviewing the code containing the call to the function, the assumption is that it is executed code.

Because of the accidental omission of the end comment marker, the call to `Perform_Critical_Safety_Function` will not be executed.

```
/* This function does
   something
function [redacted]doit()
{

}
/* end function */
```

Rule 2–10–2	(Required)	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

RULES

Rationale

If an identifier is declared in an inner scope and it uses the same name as an identifier that already exists in an outer scope, then the innermost declaration will "hide" the outer one. This may lead to developer confusion.

The terms outer and inner scope are defined as follows:

- Identifiers that have file scope can be considered as having the outermost scope.
- Identifiers that have block scope have a more inner scope.
- Successive, nested blocks, introduce more inner scopes.

EXAMPLE

Example

Brackets ← Same

```
int16_t i;
{
   int16_t i;   // This is a different variable
                // This is Non-compliant
   i = 3;       // It could be confusing as to which i this refers
}
void fn ( int16_t i )   // Non-compliant
{
}
```

Rule 3–2–3 (Required) A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Rationale

Having a single declaration of a type, object or function allows the compiler to detect incompatible types for the same entity.

Normally, this will mean declaring an external identifier in a *header file* that will be included in any file where the identifier is defined or used.

Example

```
// header.hpp
extern int16_t a;

// file1.cpp
#include "header.hpp"

extern int16_t b;

// file2.cpp
#include "header.hpp"

extern int32_t b;  // Non-compliant - compiler may not detect the error
int32_t a;         // Compliant     - compiler will detect the error
```

RULES

Rule 15.1 The *goto* statement should not be used

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

It is not specifically required that you not do this.

Rationale

Unconstrained use of *goto* can lead to programs that are unstructured and extremely difficult to understand.

EXAMPLE

In some cases a total ban on *goto* requires the introduction of flags to ensure correct control flow, and it is possible that these flags may themselves be less transparent than the *goto* they replace. Therefore, if this rule is not followed, the restricted use of *goto* is allowed where that use follows the guidance in Rule 15.2 and Rule 15.3.

See also

Rule 9.1, Rule 15.2, Rule 15.3, Rule 15.4

Rule 13.4 The result of an assignment operator should not be *used*

C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15, 18; Undefined 32] [Koenig 6]

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Amplification

This rule applies even if the expression containing the assignment operator is not evaluated.

Rationale

The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code;
- It introduces additional *side effects* into a statement making it more difficult to avoid the undefined behaviour covered by Rule 13.2.

Example

```
x = y;                          /* Compliant                                  */
a[ x ] = a[ x = y ];            /* Non-compliant - the value of x = y
                                 * is used                                   */

/*
 * Non-compliant - value of bool_var = false is used b
 * bool_var == false was probably intended
 */
if ( bool_var = false )
{
}
```

```
/* Non-compliant even though bool_var = true isn't evaluated          */
if ( ( 0u == 0u ) || ( bool_var = true ) )
{
}

/* Non-compliant - value of x = f() is used                           */
if ( ( x = f ( ) ) != 0 )
{
}

/* Non-compliant - value of b += c is used                            */
a[ b += c ] = a[ b ];

/* Non-compliant - values of c = 0 and b = c = 0 are used             */
a = b = c = 0;
```

See also

Rule 13.2

Rule 13.4 The result of an assignment operator should not be *used*

C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15, 18; Undefined 32] [Koenig 6]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies even if the expression containing the assignment operator is not evaluated.

Rationale

The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code;
- It introduces additional *side effects* into a statement making it more difficult to avoid the undefined behaviour covered by Rule 13.2.

Example

```
x = y;                          /* Compliant                              */
a[ x ] = a[ x = y ];            /* Non-compliant - the value of x = y
                                 * is used                               */

/*
 * Non-compliant - value of bool_var = false is used but
 * bool_var == false was probably intended
 */
if ( bool_var = false )
{
}
```

```
/* Non-compliant even though bool_var = true is to evaluate   */
if ( ( 0u == 0u ) || ( bool_var = true ) )
{
}

/* Non-compliant - value of x = f() is used                   */
if ( ( x = f ( ) ) != 0 )
{
}

/* Non-compliant - value of b + c is used                     */
a[ b + c ] = a[ b ];
```

```
/* Non-compliant - values of c = 0 and b = c = 0 are used     */
a = b = c = 0;
```

See also

Rule 13.2

Rule 6–2–1	(Required)	Assignment operators shall not be used in sub-expressions.

Rationale

Assignments used in a sub-expression add an additional side effect to that of the full expression, potentially resulting in a value inconsistent with developer expectations. In addition, this helps to avoid getting = and == confused.

Example

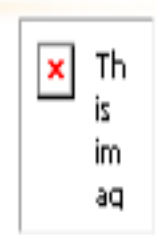
EXAMPLES

```
x = y;
x = y = z;                  // Non-compliant
if ( x != 0 )               // Compliant
{
   foo ( );
}
bool b1 = x != y;           // Compliant
bool b2;
b2 = x != y;                // Compliant
if ( ( x = y ) != 0 )       // Non-compliant
{
   foo ( );
}
if ( x = y )                // Non-compliant
{
   foo ( );
}
if ( int16_t i = foo ( ) )  // Compliant
{
}
```

x=y=z;

if((x=y)!=0)

if(x=y)



Rule 5–0–13	(Required)	The *condition* of an *if-statement* and the *condition* of an *iteration-statement* shall have type *bool*.

Rationale

If an expression with type other than *bool* is used in the *condition* of an *if-statement* or *iteration-statement*, then its result will be implicitly converted to *bool*. The *condition* expression shall contain an explicit test (yielding a result of type *bool*) in order to clarify the intentions of the developer.

Exception

A condition of the form *type-specifier-seq declarator* is not required to have type *bool*.

This exception is introduced because alternative mechanisms for achieving the same effect are cumbersome and error-prone.

Example

```
extern int32_t * fn  ( );
extern int32_t   fn2 ( );
extern bool      fn3 ( );

while ( int32_t * p = fn ( ) )     // Compliant by exception
{
   // Code
}

// The following is a cumbersome but compliant example
do
{
   int32_t * p = fn ( );
   if ( NULL == p )
   {
      break;
   }
   // Code...
}
while ( true );                    // Compliant

while ( int32_t length = fn2 ( ) ) // Compliant by exception
{
   // Code
}

while ( bool flag = fn3 ( ) )      // Compliant
{
   // Code
}

if ( int32_t * p = fn ( ) )        // Compliant by exception
if ( int32_t length = fn2 ( ) )    // Compliant by exception
if ( bool flag = fn3 ( ) )         // Compliant
if ( u8 )                          // Non-compliant
if ( u8 && ( bool_1 <= bool_2 ) )  // Non-compliant
for ( int32_t x = 10; x; --x )     // Non-compliant
```

EXAMPLES

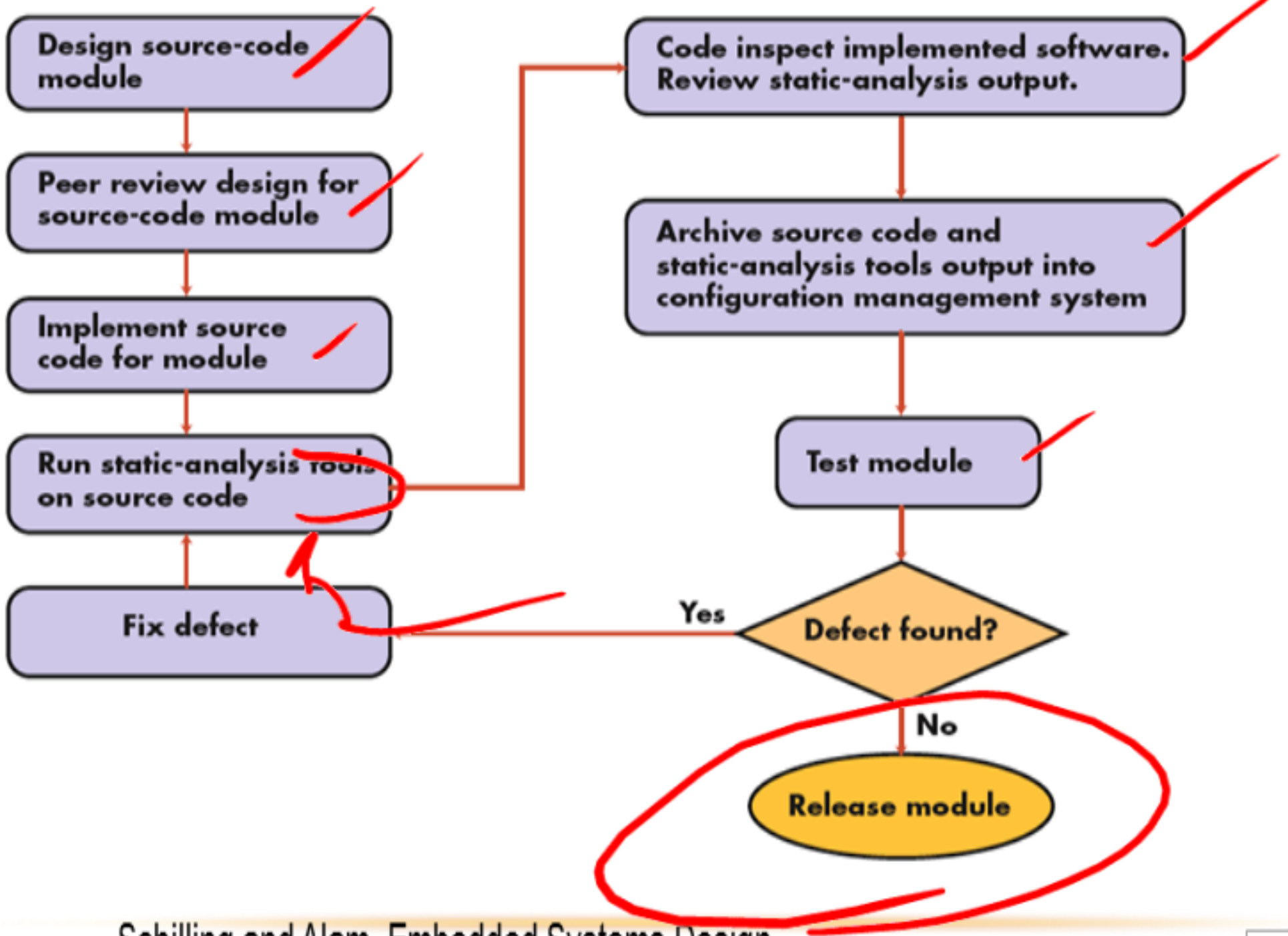
if(u8)

SE3910 REAL TIME SYSTEMS

im
aq

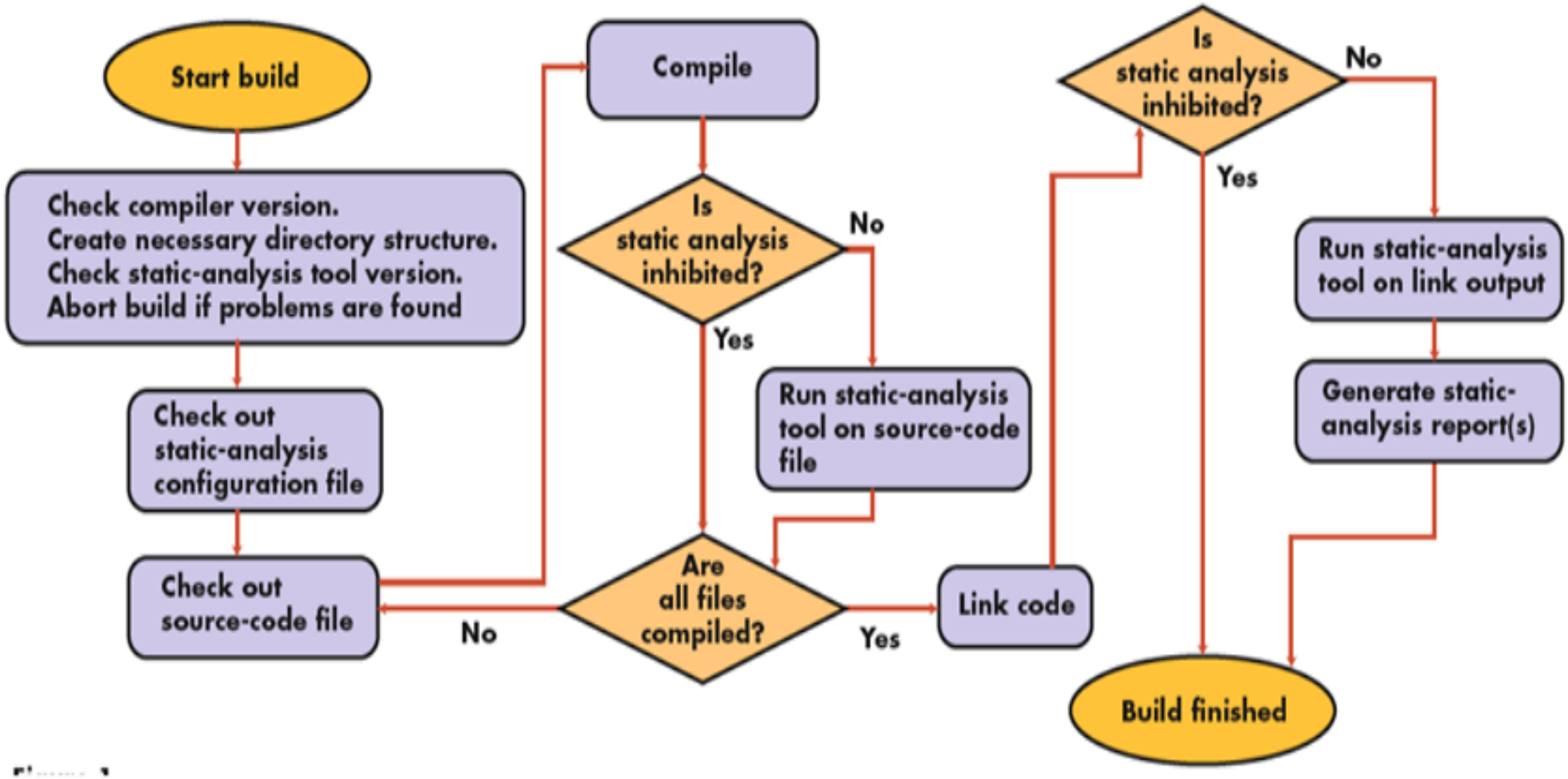
SW DEVELOPMENT PROCESS INCORPORATING STATIC ANALYSIS

This software-development process segment incorporates static analysis



AUTOMATE CODING STANDARD COMPLIANCE

Sample flowchart of an automated build script incorporating static analysis



LEGACY CODE IN INTEGRATION

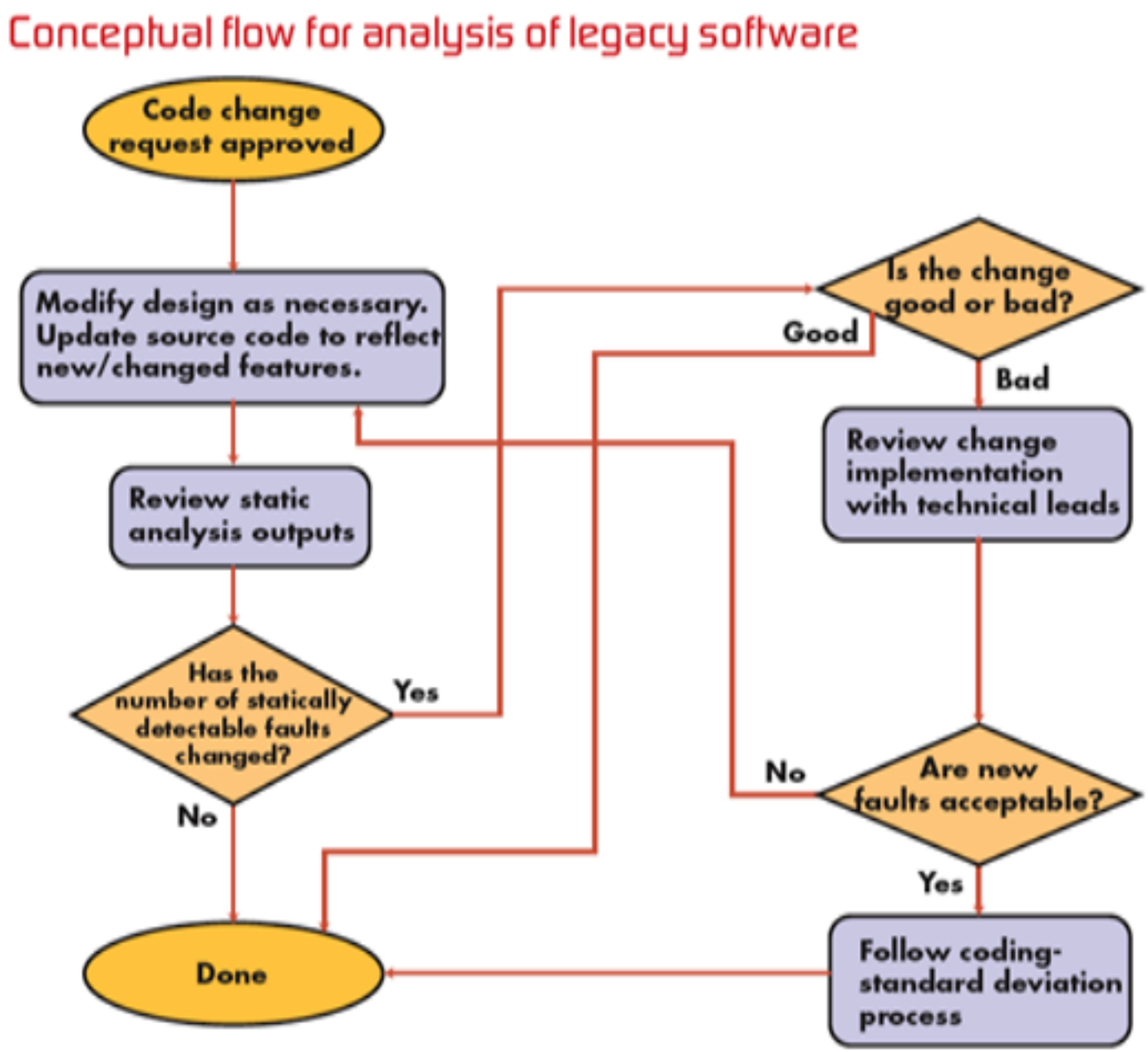
- Applying to existing code base can be challenging
- Success depends upon
  - age of the code
  - engineer's programming style
  - paradigms used
  - Diligence of engineers applying static analysis

- Many projects have abandoned SA when the first run of the tool generates 100,000 or more warnings. => ≥ 5,000 LOC
- With legacy code, it's often not practical to remove all statically detectable faults.

LEGACY SOFTWARE METHODOLOGY

- Treat each statically detectable fault as a bug fix. – Risk
  - 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 catastrophic 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.
- Coding standard development follows the same behavior as that for traditional coding standard development.

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/