Best Practices in Code Inspection for Safety-Critical Software

Jorge Rady de Almeida Jr., João Batista Camargo Jr., Bruno Abrantes Basseto, and Sérgio Miranda Paz, *Escola Politécnica da USP*

More and more, computer systems are supervising and controlling safety-critical systems, replacing fail-safe hardware techniques. Such systems are responsible for the safety of individuals as well as the property and environment in their domain.

However, a failure in the system's hardware could cause the software to execute incorrectly, thus adversely affecting the system's outputs. The failure's source could be corruption of the stored program or microprocessor damage. But even if the software does exactly what it was specified to do, it still might not be safe if the operation scenario was unknown or not previously evaluated.[1]

The verification of all important aspects of safety-critical systems is obligatory, because it can save human lives and material resources.[2] An important part of safety analysis is software analysis, and an important part of software analysis is code inspection.

For about 20 years, our Safety Analysis Group of the University of São Paulo's Polytechnic School has been developing a code inspection methodology inserted in a safety analysis process. The key to this methodology is a code inspection checklist that covers the main aspects that developers must verify in safety-critical applications.[3,4] Our checklist is also valuable for verifying and validating non-safety-critical applications. Every care taken in verifying an application's code adds value to the system, improving its quality and reliability.

Implementing a code inspection checklist can help ensure obedience to good coding rules and improve the quality and reliability of safety-critical applications.

Safety analysis and formal inspection

Safety analysis is fundamental to safety-critical-system verification and validation. The safety analysis team should be independent, with no commitment to the system supplier. This is especially true if the team is working on systems that require Level 4 of IEC 61508 (SIL4),[5] the highest level of safety integrity.

Considering the importance of safety analysis, software design should follow a quality model—for example, the Capability Maturity Model.[6] This process includes activities involving important relationships among the system operator, the safety analysis team, and the system supplier.

Safety analysis consists primarily of qualitative analysis, whose major advantage is that it induces the analyst to think of the problem's complexity rather than abstracting it away.[7] Some of a system's quality requirements, such as reliability, safety, and maintainability, might conflict with each other, justifying a careful analysis.[8]

As part of safety analysis, formal software inspections aim to detect and eliminate errors in the products developed during the software life cycle.[9,10] Formal inspections are applicable to any product or partial product of software development, including requirements, specification, design, and code.

One tool for identifying failures during inspections is a checklist[11] that covers the general failure classes. This checklist helps inspectors by listing all the fault types or symptoms to look for.[12] (The sidebar describes two other inspection tools.)

The checklist

The checklist comprises a set of questions that guide the analysis. We limit the checklist to one page (long checklists are hard to use[13]), and the items reflect the latest project experience. Adherence to this checklist ensures that the developers have followed good rules of coding.

If the checklist consists exclusively of safety criteria, it can be called a safety checklist.[14] However, the checklist directly influences areas other than safety. For example, covering all the checklist items will likely increase the system's fault tolerance. Also, a systematic inspection technique such as a checklist will lower the possibility of overlap (that is, different parts of the inspection discovering the same faults), thus increasing the inspection's effectiveness.[15] In addition, safety is closely related to security. So, because the checklist mainly covers safety aspects, it will certainly also cover security, whose functional requirements are less restrictive.

The IEC 1508 standard greatly influences the topics that the checklist covers. The standard requires static verification methods for the source code to ensure conformance to the software module design specification, to the required coding standards, and to the requirements identified during safety planning.[5]

The standard also states that safety-critical applications must use a completely deterministic programming language. If this is impossible, developers must restrict the programming language to its nonambiguous features. Even if the language does not fulfill all requirements, developers can eventually employ a language subset that satisfies the conditions—for example, subsets of C or PL/M.

Our checklist is geared toward C, PL/M, and assembly because the programs our group has analyzed use these languages. We developed this list during several projects related to the safety analysis of critical systems, mainly those that control subway trains.

The following sections describe the 11 areas that the checklist covers. If an inspection omits any item in an area, incorrect or unexpected processing might result, thereby violating the system specifications.

Routine return methods

Returning to the caller subroutine often causes mistakes that can lead to program flow interruption or discontinuity, stack corruption or overflow, or incorrect value return. These situations usually cause unpredictable behavior in the system and can affect its security.

This check verifies that all software routines have a correct return method. That is, the routines must have the necessary code to ensure that, after the processing returns to the

Software Inspection Tools

Lasse Harjumnaa and Ilkka Tervonen developed a software tool that lets a geographically distributed team make inspections.[1] Web pages store the information, which is accessible at any time.

Fevzi Belli and Radu Crisan proposed a semiautomated method that tries to reduce the cost of applying checklists.[2] Their method is interactive; that is, reviewers can actualize, extend, and check their checklists' consistency and redundancy.

References

1. L. Harjumnaa and I. Tervonen, "A WWW-Based Tool for Software Inspection," *Proc. 31st Hawaii Int'l Conf. System Sciences*, vol. 3, IEEE CS Press, 1998, pp. 379–388.
2. F. Belli and R. Crisan, "Towards Automation of Checklist-Based Code Reviews," *Proc. 7th Int'l Symp. Software Reliability Eng.* (ISSRE 96), IEEE CS Press, 1996, pp. 24–33.

If the application uses different programming languages together, verify that the calling convention is compatible between the modules.

activating routine, they continue without losses, thereby assuring continuity and preserving the system stack.

Verifying the routines' correct termination involves inspecting two items. First, if the routine returns a value, verify that all paths that drive to the routine's ending have a returned value. For example, in

```
int a (int b)
{
  if (b == 0) {
    d = 1;
    return 0;
  } else {
     if (b == 1)
       d = 3;
       // this path does
       // not return a value!
     else
       return 1;
  }
}
```

when `b` = 1, the returned value is undefined. In C source code, that kind of mistake is common.

Second, verify that parameter receiving and stack cleaning for each software routine have been coherently established by design convention. High-level programming languages usually provide standard methods for this, letting the programmer choose between different methods if necessary, usually by compilation directives such as `#pragma` in C.

This item must verify that all routines respect the convention of parameter usage or, in special cases that do not adopt the convention, that the routines coherently handle stack utilization. If the application uses different programming languages together, verify that the calling convention is compatible between the modules. When one language is low-level source code such as assembly, stacks are maintained explicitly, so verify it carefully.

Interrupt-handling routines and critical regions

This check verifies the correct interrupt servicing by the corresponding routines and by routines that depend on interrupt services in critical regions. Some rules are essential. The inspection must locate all interrupt service routines and routines that are called by interrupt services and then verify the following six items.

First, verify that the routines activated by hardware interruptions do all the correct housekeeping before returning, in all paths of return. That is, the routines must supply the correct interrupt acknowledgment commands to the respective interrupt controllers—for instance, to avoid keeping a pending interrupt that could compromise the program flow. For example, in

```
void interrupt int_rx(void)
{
  if(n_queue == MAX) return;
    // this path is dangerous
  queue[n_queue++]
    = inport(P_ADDRESS);
  outport(EOI, EOIVALUE);
}
```

when the first condition is met, the routine ends without freeing the interrupt controller with the end-of-interrupt command.

Second, for interrupt-handling routines, verify the correct return process. High-level languages supply special directives for the compiler to generate adequate object code, such as the `interrupt` keyword in C and PL/M. In low-level source code such as assembly, ending an interrupt service is explicit and must include the correct code for saving and restoring the processor state and the return-from-interrupt machine instruction.

Third, check the interrupt-handling routines, paying special attention to preventing the corruption of code that executed before the interrupt. (The inspection will need to reference the global symbols addressed by these routines when it checks the critical regions—the sixth item in this section.) In assembly source code, you must also check the restoring of all register and memory locations after their use and before the routine return. For example,

```
int_3 PROC NEAR
  PUSH R0
  PUSH R1
  MOV R0, TICK
  INC R0
  MOV R2, TIMER
    ; the value of R2
    ; will be corrupted
  ADD R2, R0
  MOV TICK, R0
  MOV TIMER, R2
```

```
    POP R1
    POP R0
    RETI
```

preserves only the registers R0 and R1, but the routine also changes R2. So, its previous value is lost, which can lead to unpredictable results. TICK and TIMER are addressed into the interrupt service, so they must always be processed in critical regions.

Fourth, check the link process to verify that each routine's correct memory address is generated and written into the proper interrupt vector table position or equivalent. This ensures that the correct service is called.

Fifth, ensure that the interrupt service's maximum execution time is limited, because during the service execution, other interrupts cannot be acknowledged and the main loop does not execute. The inspection must check the program flow paths inside an interrupt service routine, verifying the possibility of occurrence of long loops, for example. Avoid enabling interrupts inside an interrupt service.

Sixth, verify the proper use of the global symbols that are addressed by interrupt service routines in all of the source code's critical regions. Ensure that pairs of interrupt-disabling and interrupt-enabling commands protect the symbols' use. For example, in

```
interrupt void insert (void)
{
    char new;
    new = inp(0xf3);
    queue fila[pos] = new;
    pos++;
}
char remove (void)
{
    char ret;
    if (pos == 0) return -1;
    ret = fila[pos];
    pos—;
    return ret;
}
```

the routine insert (), which is interrupt activated, adds characters that are read from a hardware port into a queue. The routine remove () removes these characters later. Because remove () does not disable interrupts, the occurrence of insert () during the execution of remove () might produce inconsistent results.

Repetitive-loop control

This check verifies loops to ensure that they can reach their final step (except loops that intentionally never end), avoiding infinite cycles in the program. It also verifies that the loop might not execute at all owing to loop control errors.

A loop that, because of programming mistakes, never finishes can prevent the activation of other routines. If a loop does not execute, its internal commands don't, either. You can check for this in two steps.

First, verify that the loop control variables are correctly tested and updated. Inadequate use or modification of values can result in the loop's nontermination or nonexecution.

Second, verify that the type of variables used in loop control is consistent with the form of their use. For example, in

```
void a (int b)
{
    char i;
    static c[256];
    for (i = b; i > 0; i—)
        c[i] = 0;
}
```

the loop never executes when the parameter (integer value b) is larger than 127. This is because the signed byte's value becomes negative, which signals skipping the loop before it executes.

In this next example,

```
void a (int b)
{
    char i;
    for (i = 0; i < b; i++)
        c[i] = 0;
}
```

the loop never finishes when it receives a value larger than 127 as the parameter (integer variable b). This is because the counter (char variable i, which is signed by default) will never reach the final limit (b).

I/O tests

This check verifies the I/O tests of important routines, especially those where reentrance must be prevented. If improper deviations occur, either from inside a routine or to a routine's body, I/O tests can detect such a failure, preventing unexpected actions. You can verify the tests in two steps.

Inadequate use or modification of values can result in the loop's nontermination or nonexecution.

The incorrect use of control structures can result in unexpected code execution, possibly causing dangerous situations.

First, verify that an input test or control variables exist that guarantee that the routine reached its end in the previous execution.

Second, verify that an output test or control variables exist that guarantee that the routine started from its beginning in the previous execution. For example, in

```
void routine(void)
{
  static int IO_Test = 0;
  if (IO_Test) fail()
  IO_Test = 1;
  /* routine body */
  if(!IO_Test) fail()
  IO_Test = 0;
}
```

`IO_Test` is a control variable that holds a nonzero value only inside the routine. An inconsistent situation will activate the routine `fail`.

Program flow control

This check verifies that the program sequence is executing properly. The incorrect use of control structures can result in unexpected code execution, possibly causing dangerous situations. You can verify routines' control flow by checking the following four items.

First, ensure the verification of control structures in selector blocks. In PL/M code, verify that a test exists before each `do case` block to verify its parameter's boundaries. If the parameter is larger than the number of block sentences, the effect of the `do case` might be undefined, depending on the compiler.[12] For example,

```
if sel > 3 then
  call fail;
else do case sel;
  do;  /* case 0 */
  end;
  do;  /* case 1 */
  end;
  do;  /* case 2 */
  end;
end;
```

needs verification because the PL/M compiler does not test the selector variable's limits.

C code that uses the `switch` command can also employ the `default` option. This option enables the detection of eventual failure situations when none of the options refer to the value being tested.

Also, all options of values presented in `switch` should have the keyword `break` termination. For example, in

```
switch (a)
{
  case 0:
    b = 1;
    break;
  case 2:
    b = 6;
  case 3:
    b = 7;
    break;
}
```

when `a` = 2, the keyword `break` is absent, which will also cause the processing associated with the case `a` = 3. The `switch` command ignores other values for the selection variable—which might indicate failure situations—because the keyword `default` is absent.

Second, verify that interrupt-disabling commands have corresponding enabling commands. This procedure is particularly important for periodical execution of time-critical program blocks and critical regions accessing variables that are modified by interrupt services.

If a routine does not have an enabling command corresponding to a previously executed disabling command, interrupt service routines will not execute. While the enable command is not executing, no interrupt can be serviced. So, the period when interrupts are disabled must not be long.

Interrupts can be enabled or disabled by different mechanisms, such as specific processor machine instructions or writing to status words. So, checking this item is not as intuitive as it seems. For example, in

```
PUSH PSW
CLI
  ; disable interrupts
CALL A
CALL B
POP PSW
  ; implicitly enable
  ; interrupts
RET
```

the instruction that restores the program status word (PSW) can also enable interrupts.

Third, for assembly code, verify that each applied stack command (PUSH) has a corresponding unstack command (POP). If it does not, the top-of-stack pointer might indicate an incorrect position.

Fourth, also for assembly, verify that all operands are correctly referenced to their register or memory segment. The correct register segment identification should always precede the operand references.

Unused source code

This check verifies that no source code exists between comment marks and is therefore unused (for example, routines that served for code development and are no longer used but remain as comments in the source code). These comments' presence does not directly cause interference. However, in future maintenance, such comments could induce the designer to make mistakes.

Variables and constants

To verify the appropriate use of variables and constants, check the following four items.

First, verify indexes explicitly against the limits of vectors and matrixes before addressing the corresponding vectors or matrixes. Although some high-level language compilers can generate code for that verification, languages such as C and PL/M do not perform any test at all. Even if a compiler generates such tests automatically, verify that compiler directives did not disable this feature. If a vector is in a loop and the loop control modifies its index, verify the consistency between the loop termination control and the vector dimension. For example,

```
if(i > MAX) fail();
  // lower limit not verified
queue[i] = queue[i+1];
  // i+1 may be out of bounds
```

compares the index i is to MAX, the array's upper limit. But the lower limit is not verified. So, if i is declared a signed variable, and the negative values cause fail() to not be called, the queue vector will be incorrectly addressed. The attribution is also incorrect because i + 1 might be out of the vector limit.

Second, check for the place and scope of declared symbols to verify that double declarations have not occurred. A constant or variable must be declared only once, in a single declaration file. Several different program modules should include the declaration file. Such a procedure can prevent inconsistent conditions if modifications in the constant value exist in more than one source code position. For example,

```
file #1:
  MAX EQU $2F
    ; assembly declaration
    ; in DECL.EQU
file #2:
  #define MAX 48
    // C declaration in
    // EXT.H
```

declares MAX inconsistently as 47 (2FH in hexadecimal representation) for the assembly module and as 48 for the C module.

Third, verify that, when a program module must redeclare a symbol in the same scope of a declaration or header file, the types in the external declarations are consistent. For example, in the external declaration

```
extern char *test;
  // declaration in file TEST.H
```

and the static declaration

```
int test [80];
  // declaration in file TEST.C
```

the external declaration declares the variable test as a char pointer, which is inconsistent with the static declaration (an array of integers).

Fourth, verify if all global and static symbols are initialized. If possible, verify that the initialization value is coherent. For example, in

```
int n_queue;
  // unitialized variable
int pos;
int x_lim = 0;
  // compiler initialization
void main(void)
{
    pos = 0;
      // explicit initialization
    for(;;) {
      main_loop();
  }
}
```

Several different program modules should include the declaration file.

Table 1

Results from using the inspection code checklist

Checklist topic	Errors (%)
Routine return methods	5
Interrupt-handling routines and critical regions	5
Repetitive-loop control	5
I/O tests	20
Program flow control	10
Unused source code	10
Variables and constants	10
Source code comments	20
Source code legibility	10
Preprocessor directives	5
Code optimization	0

n_queue is not initialized, and its value at the beginning is unpredictable.

Source code comments

Verify that the comments improve the understanding of code, enhancing software maintenance. Comments must not lead to incorrect conclusions, contain ambiguities, or cause code misinterpretation.

Source code legibility

Legibility is fundamental to code maintenance; if source code is written in a complex way, understanding it will require much more effort. This might lead to errors during initial codification and during maintenance. You can achieve legibility by checking the following eight items.

First, don't abbreviate names of constants, variables, or routines, and avoid long names that hinder readability.

Second, use a mechanism for word delimitation—for example, separator characters or capital letters.

Third, ensure that variable and routine names follow coherent, consistent criteria. For example, the routines' names should always be verbal forms in the same tense and same person.

Fourth, use prefixes and suffixes related to symbol functions—for example, p for pointers and MAX and MIN for maximum and minimum:

```
BYTE Mode;
  #Define MIN_Temperature 32
BYTE * pMode;
  #Define MAX_Temperature 212
```

Fifth, represent constants and variables differently. For example, reserve uppercase letters for names of constants and lowercase letters for names of routines and variables, where the programming language allows this.

Sixth, ensure that indentation is consistent (for example, the PL/M commands DO and END should be aligned).

Seventh, use brackets whenever they contribute to expression clarity, even if they are semantically unnecessary. For example, A + (B * C) / D is preferable to A + B * C / D because it clarifies the operation.

Eighth, avoid extremely complex structures—for example, those with too many operations or nesting levels.

Preprocessor directives

Avoid indiscriminate use of preprocessor directives in the source code, which might cause errors during program maintenance. For example, the direct or indirect alteration of the control value can cause the noncompilation or the erroneous compilation of code portions.

Code optimization

Avoid optimizations during compilation, which might generate the object code in an unexpected way. High-level programming languages usually have ambiguous aspects that could have different interpretations, depending on the selected optimization level.

Comments

Our checklist has proven an excellent tool for improving analysis quality. Table 1 lists the percentage of errors we found for each checklist topic when using it on projects. These figures are related to the work's initial phase (when the safety analysis process detects the initial problems), where the number of problems is still great. Because the supplier can fix most of these problems, the total number of problems will tend to decrease, making the software more robust and consequently improving safety.

Some checklist topics are more relevant to safety than others. Such topics include routine return methods, interrupt-handling routines and critical regions, repetitive-loop control, I/O tests, program flow control, variables and constants, and code optimization. Other topics are indirectly important because they pertain only to code maintenance. These topics are unused source code, source code comments, source code legibility, and preprocessor directives.

Adherence to the checklist strongly indicates that the code meets the minimum requirements for use in safety-critical applications. However, the checklist covers only the main aspects to verify; it should be augmented to cover other programming languages and new programming techniques.

In source code with thousands of lines, verifying all the checklist items is not trivial. However, such activity contributes greatly to assuring the system's safety. Nevertheless, code inspection constitutes only one step of safety analysis of a safety-critical system. For a complete analysis, you must implement the entire methodology.

References

1. I.B. Pirie, "Software—How Do We Know It Is Safe?" *Proc. 1999 IEEE/ASME Joint Railroad Conf.*, IEEE Press, 1999, pp. 122–129.
2. N.G. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
3. M. Henricson and E. Nyquist, *Programming in C++, Rules and Recommendations*, Document M 90 0118 Uen, Ellemtel Telecommunication Systems Laboratories, 1992.
4. W.D. Yu, "A Software Fault Prevention Approach in Coding and Root Cause Analysis," *Bell Labs Tech. J.*, vol. 3, no. 2, Apr.–June 1998, pp. 3–21.
5. *CEI IEC Std. 61508-3—Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, Int'l Electrotechnical Commission, 1997.
6. M.S. Krishnan and M.I. Kellner, "Measuring Process Consistency: Implications for Reducing Software Defects," *IEEE Trans. Software Eng.*, vol. 25, no. 6, Nov./Dec. 1999, pp. 800–815.
7. C.B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Trans. Software Eng.*, vol. 25, no. 4, July/Aug. 1999, pp. 557–572.
8. R. Singh, "A Systematic Approach to Software Safety," *Proc. 6th Asia Pacific Software Eng. Conf.*, IEEE CS Press, 1999, pp. 420–423.
9. B. Cheng and J. Ross, "Comparing Inspection Strategies for Software Requirement Specifications," *Proc. Australian Software Eng. Conf.*, IEEE CS Press, 1996, pp. 203–211.
10. M. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.*, vol. 15, no. 3, 1976, pp. 182–211.
11. T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley, 1994.
12. S. Biffl and M. Halling, "Software Product Improvement with Inspection," *Proc. 26th Euromicro Conf.*, vol. 2, IEEE CS Press, 2000, pp. 262–269.
13. Y. Chernak, "A Statistical Approach to the Inspection Checklist Formal Synthesis and Improvement," *IEEE Trans. Software Eng.*, vol. 22, no. 12, Dec. 1996, pp. 866–874.
14. R.R. Lutz et al., "Safety Analysis of Requirements for a Product Family," *Proc. 3rd Int'l Conf. Requirements Eng.* (ICRE 98), IEEE CS Press, 1998, pp. 24–33.
15. A. Porter and L. Votta, "Comparing Detection Methods for Software Requirements Inspections: A Replication Using Professional Subjects," *Empirical Software Eng.*, vol. 3, no. 4, Dec. 1998, pp. 355–379.

About the Authors

Jorge Rady de Almeida Jr. is a professor in the Computing and Digital Systems Engineering Department at the University of São Paulo's Polytechnic School. He is also a coordinator of the school's Safety Analysis Group, where he has performed research in safety-critical systems. He has substantial experience in safety analysis of urban-railway systems and database systems. He received his PhD in computer engineering from the Polytechnic School. Contact him at the Computing and Digital Systems Eng. Dept., PCS, Escola Politécnica da USP, Av. Prof. Luciano Gualberto, trav. 3, n. 158, CEP 05508-900, São Paulo, Brazil; jorge.almeida@poli.usp.br.

João Batista Camargo Jr. is a professor in the Computing and Digital Systems Engineering Department at the University of São Paulo's Polytechnic School. He also coordinates the school's Safety Analysis Group, where he has performed research in safety-critical systems and risk analysis. He has substantial experience in safety analysis of urban-railway systems. He received his PhD in computer engineering from the Polytechnic School. Contact him at the Computing and Digital Systems Eng. Dept., PCS, Escola Politécnica da USP, Av. Prof. Luciano Gualberto, trav. 3, n. 158, CEP 05508-900, São Paulo, Brazil; joao.camargo@poli.usp.br.

Bruno Abrantes Basseto is a researcher at the University of São Paulo's Polytechnic School, where he participates in projects with the Safety Analysis Group. He has substantial experience in safety analysis of urban-railway systems. He received his master's in computer engineering from the Polytechnic School. Contact him at the Computing and Digital Systems Eng. Dept., PCS, Escola Politécnica da USP, Av. Prof. Luciano Gualberto, trav. 3, n. 158, CEP 05508-900, São Paulo, Brazil; bruno.basseto@poli.usp.br.

Sérgio Miranda Paz is a researcher at the University of São Paulo's Polytechnic School, where he participates in projects with the Safety Analysis Group. He has substantial experience in safety analysis of urban-railway systems. He received his PhD in computer engineering from the Polytechnic School. Contact him at the Computing and Digital Systems Eng. Dept., PCS, Escola Politécnica da USP, Av. Prof. Luciano Gualberto, trav. 3, n. 158, CEP 05508-900, São Paulo, Brazil; smpaz@laa.pcs.usp.br.