1. ANALYSIS AND RECOMMENDATIONS FOR A RELIABLE PROGRAMMING OF SOFTWARE-BASED SAFETY SYSTEMS

Juan Núñez Mc Leod, Jorge Núñez Mc Leod and Selva S. Rivera

Instituto CEDIAC
Capacitación Especial y Desarrollo de la Ingeniería Asistida por Computadora
Engineering Faculty, Cuyo National University
CC 405, 5500, Mendoza, Argentina

ABSTRACT

The present paper summarizes the results of several studies performed for the development of high reliability software on i486 microprocessors, towards its utilization for control and safety systems for nuclear power plants.

The work is based on software programmed in C language. Several recommendations oriented to high reliability software are analyzed, relating the requirements on high level language to its influence on assembler level. Several metrics are implemented, that allow for the quantification of the results achieved.. New metrics were developed and other were adapted, in order to obtain more efficient indexes for the software description. Such metrics are helpful to visualize the adaptation of the software under development to the quality rules under use.

A specific program developed to assist the reliability analyst on this quantification is also presented in the paper. It performs the analysis of an executable program written in C language, disassembling it and evaluating its inter al structures.

INTRODUCTION

In order to write a program, a high-level language is usually used, as it greatly simplifies the writing work for the programmer. However, such a high-level language is not directly interpreted by the computer, and it needs to make use of a compiler. The compiler transforms the high-level program (source code) into a low-

level program (object code), which is understandable by the computer. The result of this compilation process is a program which is functionally equivalent, but with an internal structure that may be substantially different from that of the source code. A program written in high-level language, consisting of a series of linear instructions, without any jump or bifurcation, may became with bifurcations in the equivalent low-level program.

When developing programs in high-level language the first quality factors affected (linearity, simplicity, conciseness, etc.) are the most appreciated by the reliability analysts and the hardware engineers. Therefore, one of the first steps to do is to provide both the programmers and the reliability analysts with the knowledge and the tools to fix the developing procedures and to evaluate that the programs include these factors.

It is important to point out that this is referred to a large quantity of small programs (or routines). Each one has a big impact in the reliability of the control or safety system.

PURPOSE

The present work is oriented to the development of reliable software for its use on control and safety systems. It is motivated in the control and safety systems of the Multi Purpose Reactor actually under construction in Egypt. These systems use a i486 platform programmed in C language and in assembler.

The purpose is to bring the following aids for both the programmer and the reliability analyst:

- Information in the form of recommendations, that allow for the writing of programs in a high-level language (C language). These programs, after compilation, should generate coding at machine-level, that complies with the Quality Assurance Plan.
- Metrics to assist the programmer, in order to be able to evaluate the best programming alternatives
- Metrics to assist the reliability analyst in order to be able to identify routines which are poorly developed from the reliability point of view
- A simple and adequate tool to provide the first quality control steps in a Software Quality Assurance program

PROGRAMMING STRUCTURES ANALYSIS

One of the goals of this study is to generate a series of recommendations on the programming rules in high level, in our case in C language. These recommendations are oriented to obtain machine code as linear as possible. The linearity of a code allows for a correct control procedure, in real time, during its execution. This is due to the fact that the programming lines must be executed sequentially and in a unique order. The linearity level acceptable for a certain code will depend on the code purpose. In example, if the code function is to control a certain parameter, which is sensed on a 2 out of 3 measurements, the code may be perfectly linear. The same is not true if the code has to decide among a series of different actions, as a function of the information provided by other codes, such as the preceding example.

The code linearity is directly related to the code testability (Kan, 1995). As the code is more linear, it will be easier to test and less time will be required for this activity.

Other of the objectives, is to obtain concise code (IEC-880, 1986). This is difficult to do when working in a high-level language. The optimal would be to develop the code directly in assembler language. The compromise solution will be to adopt assembler language where it becomes necessary.

On the next sections, some cases are developed, for a correct interpretation and documentation on the recommendations that follow:

Data conversion The data conversion job is realized with the mathematical coprocessor in two steps. The first step is to load the numerical value with the load instruction in original format, either floating point or integer, with their diverse sizes; next, the unload follows, according to the format required. When working in ascendant direction, that means, when converting integer to floating point, this scheme is directly used, but when converting in the reverse direction, a general conversion module to convert to 64 bits integers is used (Brey, 1991).

Independently of this module, it must be validated the possibility of this assignment. The call to this conversion module may be avoided working directly over the source code, as can be observed in Table 1. For simplicity, the validation, similar for both examples, has been omitted. In the lower part of the table, a comparison on linearity and conciseness can be observed, showing the clear advantages of the example 2.

Library functions There is a series of mathematical functions, mainly transcendent functions, already implemented in the mathematical coprocessor in the i486 (Brey, 1991). For generality reasons and in order to simplify the compilation, the compiler does not use any of these functions, implementing its own functions (library functions). A library function returns a certain value as a functions of one or more arguments, or none. However, the call to the library function implies a series or arguments adaptations. For example, the necessary argument may be double precision, and the real argument single precision, and this fact implies a preliminary conversion before the call to the calculation module (in a similar manner as that shown before). On the other side, all the library functions of a certain class have a similar structure, in order to easy the use of the compiler. An example can be observed in Table 2.

Logical operations Logical operations are based in comparisons that allow to take decision in one or other way, being true or false. These decisions constitute, structurally, jumps that break the linearity of the structure. As an example, a classic comparison type is shown in Table 3. The resulting assembler code is also shown. In this case, the use of '*' instead of '&&' (logic AND) allows the linearization of the calculation without further consequences

Conditional jumps (If-then) For this type of sentences, the argument calculation can be linearized but, in general, it is impossible to avoid at least one conditional jump, which is the essence of the IF conditional sentence. The coding of this sentence can be linearized if the objectives are simple expressions related among them. For example, it can be observed in Table 4, different linearization possibilities for the case of assigning to one variable the greater value found among other two variables.

TABLE 1
NUMERICALS DATA CONVERSION

Example 1	Example 2

...
{
double xd ;
long xl ;
short xs ;
 	
xd=10 ;
xl = (long) xd ;
xs = (short) xd ;
}
	...
	{
	double xd ;
	long xl ;
	short xs ;
	
	xd = 10 ;
	_asm { fld [xd]
	fistp [xl]
	fld [xd]
	fistp [xs] }
	}

assembler code length : 24 lines jumps : none calls : 2	assembler code length : 6 lines jumps : none calls : none

TABLE 2
LIBRARY FUNCTION

Example 1	Example 2

...
{
double a, b ;
 	
a = 20.0 ;
b = sqrt(a) ;
}
	...
	{
	double a, b ;
	
	a = 20.0 ;
	_asm { fld [a]
	fsqrt
	fstp [b] }
	}

assembler code length : 102 lines jumps : 16 calls : 4	assembler code length : 5 lines jumps : none calls : none

TABLE 3
LOGICAL OPERATIONS CALC

Example 1	Example 2

...
{
int ai, bi, ci, d ;
...
d = ai > bi && ai < ci ;
}
	...
	{
	int ai, bi, ci, d ;
	...
	d = (ai > bi) * (ai < ci) ;
	}

mov eax,dword ptr [ai] cmp dword ptr [bi],eax **jnl jp1** mov eax,dword ptr [ai] cmp dword ptr [ci],eax **jle jp1** mov dword ptr [d],00000001 **jmp jp2** jp1 mov dword ptr [d],00000000 jp2 ...	mov eax,dword ptr [ai] xor ecx,ecx cmp dword ptr [ci],eax setnle cl mov eax,dword ptr [ai] xor edx,edx cmp dword ptr [bi],eax setl dl imul ecx,edx mov dword ptr [d],ecx
assembler code length : 9 lines jumps : 3 calls : none	assembler code length : 10 lines jumps : none calls : none

TABLE 4
IF CONDITIONAL STATMENT

Example 1	Example 2	Example 3

...
{
int ai, bi, ci, d ;
...
if (ai > bi)
ci = ai ;
else
ci = bi ;
}
	...
	{
	int ai, bi, ci, d ;
	...
	ci = ai > bi ? ai : bi ;
	}
		...
		{
		int ai, bi, ci, d ;
		...
		ci = (ai > bi)* ai + (ai <= bi)* bi ;
		}

assembler code length: 8 lines jumps : 2 calls : none	assembler code length: 6 lines jumps : 1 calls : none	assembler code length : 12 lines jumps : none calls : none

Loop structures (For, While and Do-while) Something similar to what occurs for the IF sentences, also occurs for these kind of sentences. The argument calculation can be linearized, but the existence of at least two jumps, one conditional and the other one unconditional, cannot be avoided, because they constitute the essence of the loop structure.

RECOMMENDATIONS

- Avoid data conversion, and in case needed, perform this function at processor level in assembler language.
- Use the coprocessor arithmetic routines, or specific library functions for the type of data used.
- Transform the logical expressions into Boolean algebraic equations.
- When using a logical bifurcation sentence, or a loop sentence, pay special attention to the logical expression that controls it.
- Use, whenever possible, implicit IF sentences. Bear in mind that a IF-THEN sentence will add at least one conditional jump, and a IF-THEN-ELSE sentence will add at least one conditional and one unconditional jump. Each conditional jump implies additional testing effort, because the tests must cover all the variable ranges, but also all the different alternatives when executing.

- The loop sentences will add always one conditional and one unconditional jump, and bearing in mind the difficulties that these offer to the programmers, its use is not recommended.

METRICS

The quantification of the quality factors is performed with the so-called software metrics (Kan, 1995). The use of certain metrics is suggested in this chapter, trying to give the programmer the tools to the control and follow-up of the code, without over charging its work.

Cyclomatic Complexity One of the goals of the software programmers for high quality is the verification, as simple as possible, of the software, in order to eliminate all the errors. To allow for easy verification and maintenance of the software, it is recommended that every module in the program must have a cyclomatic complexity not exceeding 10. The McCabe cyclomatic complexity measure was designed to identify the maintainability of a program (Kan, 1995). This measure can be used to indicate the required effort to test a program. The cyclomatic complexity is equal to the number of binary decisions in a program plus 1. A 3-option decision is accounted as 2 binary decisions. A CASE sentence with n possible ways is accounted as (n-1) binary decisions. The iteration in a loop is accounted as a binary decision.

Fan-in and Fan-out The cyclomatic complexity evaluates the complexity of each module, and implicitly assumes that each module in the program is a separated entity. The structural metrics try to take into account the interactions between modules of a program or system, and quantify them.

The most common structural metrics are fan-in and fan-out (Kan, 1995). Fan-in is the number of modules that call a certain module and fan-out is the number of modules that are called by a certain module. Modules that have a large fan-in and fan-out must be relatively small and simple. Big and complex modules should have a small fan-in but may have a big fan-out. Modules with large fan-in and large fan-out may indicate a bad structural design.

Cycle-Ability Index In the present work, a cycle-ability index is proposed. This index indicates the complexity of each module taking into account the inherent difficulty in the decision taking process, the difficulty that the programmer has to understand the problem, the consequences of the proposed solution and, finally, the use and refreshment of values for the decision taking process. The following table indicates the cycle-ability index values for different sentences:

TABLE 5
STATMENTS WEIGHTED FOR CYCLE-ABILITY INDEX

Statements	Cycle-ability index
If - then -else	1
Switch or Case	2
For, While and Do-while	3

Linearity index A linearity index is proposed. It allows for the quantification of the number of calls to subroutines, and conditional or unconditional jumps present in the code. From such a way, it is straightforward to fix and control the limits to codes with different functional capabilities.

Il = Nr. subroutines called + Nr. conditionals jump + Nr. unconditionals jump

Self containing index It indicates the degree that the code has to perform all its implicit and explicit functions by itself (IAEA, 1988). The calculation formula is as follows:

$$I_S = \frac{nr.\_of\_delegated\_functions}{Nr.\_of\_delegated\_functions\_+\_nr.\_of\_not\_delegated\_functions}$$

In equation (2) are accounted those functions explicitly invoked by the program from the Operative System (delegated) and those that, being able to be used, are not used.

Maturity index The maturity index is associated to the program robustness. The robustness (IAEA, 1988) is defined as the degree until what the code can continue its execution, despite some violation of the specification hypothesis. The maturity index approaches exponentially 1 and can be calculated with equation (3)

$$I_M = \frac{MT - (Fa + Fm + Fe)}{MT}$$

Where : MT is total number of modules, Fa is added modules number, Fm is modified modules number and Fe is eliminated modules number

SPECIFIC PROGRAM TO ASSIST THE RELIABILITY ANALYST

A specific program was developed to assist the reliability analyst using MS-Visual C++ under Windows95 platform. This programs allows the code analysis in machine language and the routine separation and metrics calculation in an automatic form.

Disassembling The study of the program structure at low-level cannot be performed directly, because it is constituted as a long chain composed of ones and zeroes, grouped in octets, which meaning is decodified by the microprocessor in sequential form. Therefore, to be able to study the structure of this kind of programs, a new compilation in reverse direction is performed, in order to obtain as a result a higher level program, but keeping the machine-code structure. The most adequate language for this task is the assembler language, because it has a direct correspondence to the machine coding. The specific program presents the results in this language, as ilustrated in tables 1 to 4, which were obtained with the mentioned program.

The programmer is able to see the code in assembler language, he can observe how the modifications performed in the source code are reflected in the assembler code. This allows for the discussion and interaction with the reliability analyst.

Metrics The program calculates, automatically and for each routine, the following metrics: Cyclomatic Complexity Index, Fan-in, Fan-out and Linearity Index

CONCLUSIONS

Several studies, relating the source coding with the object coding, have been used to demonstrate how they can influence favorably in some software quality factors. Based on these, a series of recommendations for the programmers have been generated. A basic tool for the software quality analysis has been also developed, which is useful not only for the programmers but also for the reliability analysts.

Several indexes that may assist both the programmers and the reliability analysts have been implemented. These indexes allow for the establishment of principles for the development and evaluation of the programming alternatives.

CONTINUATION

The next steps are towards the possibility to perform comparative evaluations of the source and the object code, to include reliability models and to study certain hardware characteristics that may severely affect the programs reliability.

In reference to the last aspect mentioned, the analysis of failures produced by induced errors in the computer stack has started.

REFERENCES

Brey, B. (1991), **The Intel Microprocessors**, Prentice-Hall.
IAEA (1988), T.R. Nr. 282, **Manual on Quality Assurance for Computer Software Related to the Safety of Nuclear Power Plants.**
IEC Standard, Publication 880 (1986), **Software for computers in the safety systems of nuclear power stations, International Electrotechnical Commission.**
Kan, S. (1995), **Metrics and Models in Software Quality Engineering, Addison-Wesley Publishing Co.**