Journal of the Korean Nuclear Society
Volume 27, Number 1, February 1995

《Techincal Report》

A Software Engineering Process for Safety-critical Software Application

Byung Heon Kang, Hang Bae Kim, Hoon Seon Chang, Jong Sun Jeon, and Suk Joon Park
Korea Atomic Energy Research Institute
(Received April 30, 1994)

Safety-critical 소프트웨어 적용을 위한 소프트웨어 개발 절차

강병헌 · 김항배 · 장훈선 · 전종선 · 박석준
한국원자력연구소
(1994. 4. 30 접수)

Abstract

Application of computer software to safety-critical systems is on the increase. To be successful, the software must be designed and constructed to meet the functional and performance requirements of the system. For safety reason, the software must be demonstrated not only to meet these requirements, but also to operate safely as a component within the system. For longer-term cost consideration, the software must be designed and structured to ease future maintenance and modifications. This paper presents a software engineering process for the production of safety-critical software for a nuclear power plant. The presentation is expository in nature of a viable high quality safety-critical software development. It is based on the ideas of a rational design process and on the experience of the adaptation of such process in the production of the safety-critical software for the Shutdown System Number Two of Wolsong 2, 3 & 4 nuclear power generation plants. This process is significantly different from a conventional process in terms of rigorous software development phases and software design techniques. The process covers documentation, design, verification and testing using mathematically precise notations and highly reviewable tabular format to specify software requirements and software design. These specifications allow rigorous, stepwise verification of software design against software requirements, and code against software design using static analysis. The software engineering process described in this paper applies the principle of information-hiding decomposition in software design using a modular design technique so that when a change is required or an error is detected, the affected scope can be readily and confidently located. It also facilitates a sense of high degree of confidence in the 'correctness' of the software production, and provides a relatively simple and straightforward code implementation effort.

1. Introduction

Monitoring sensors and controlling actuators are very important parts of a process control system. When mal-functioning of the system can potentially lead to loss of human lives or significant loss of properties, such system is normally known as a safety-critical system. Examples of safety-critical systems are nuclear reactor shutdown system, automatic flight control system, air traffic control system, railway signal system and chemical plant control system.

In the past, process control of most safety-critical systems was implemented by hardware. It required significant human intervention. In recent years, the application of computer software to automate the process control of these systems is on the increase. Such computer software is known as safety-critical software. The increased use of safety-critical software is mainly due to the manipulative flexibility of the software over the hardware. For example, change of a software behaviour is relatively easily achieved through suitable manipulation of programming logic in the code. The software flexibility leads to the following advantages :

(i) A software system can be designed for ease of change. This has a significant cost benefit in terms of maintenance and future changes.

(ii) For identical systems, the cost of replication of a software system is much lower than that of a hardware system. For similar systems, program family of the software system can be designed so that the cost of instantiation of a family member can still be significantly lower than that of a hardware system.

(iii) Often, safety net features that are difficult or infeasible to be incorporated in a hardware system can be easily implemented in a software system to enhance the safety aspects of the system.

As safety-critical systems can affect the lives of the general public in various ways, the operation of these systems normally requires the approval of appropriate governmental regulatory agencies. Depending on the nature of each system, a set of requirements must be met in the design and construction of the system to demonstrate to the relevant regulatory agency that the safety concerns of the public can be assured before the approval is granted [1].

In a safety-critical system such as a nuclear power generation plant, the use of safety-critical software must therefore comply with its related requirements to show that it operates safely as a component within the system. When the plant is an industrial enterprise, additional commercial requirements must also be satisfied to ensure its competitiveness and survival.

To meet these requirements, this paper discusses various aspects of a software engineering process for the design and construction of safety-critical software based on the ideas of a rational design process [2] and on the experience of the adaptation of such process in the production of safety-critical software for Shutdown System Number Two of the Wolsong Nuclear Power Plants. The discussion is expository in nature of a viable high quality safety-critical software development. This process is significantly different from a conventional process commonly used in software industry in that : (i) rigorous software development phases are used to achieve precise software specification and implementation correctness, and (ii) the principle of information-hiding decomposition is systematically applied to software design to cater for future software changes and maintenance.

2. Overview of Software Engineering Process

2.1. General Requirements for Safety-Critical Software

Functional and performance requirements are the most important in determining the role of safety-critical software to be included in a nuclear power generation plant. Therefore, the design and construction of the software must be such that the fulfilment of

these basic requirements is assured. Like any other comparable industrial complex, plant maintenance and modification affecting the functional and performance requirements will occur during different stages of the lifetime of the plant. Such occurrence necessitates corresponding change in software. Therefore for cost consideration, the software must be also designed to ease future maintenance and modification.

To a regulatory agency responsible for such plants, the onus to demonstrate the safety assurance of the software is on the organization for the plant [3]. One central focus, among others, of the demonstration refers to a software engineering process that is in compliance of a high-level standard document for safety-critical software development of the organization. This high-level standard document serves as a collection of mission statements for such software construction and can be used as a basis for assessing the adequacy of the software engineering process so that the safety assurance of the use of the software produced can be better judged. The success of the assessment hinges on the accuracy, consistency, completeness and reviewability of the documents for various phases of software development.

2.2. Safety-Critical Software Development

Like other quality software construction, the software engineering process described in this paper for safety-critical software involves three main activities, namely software development, software verification and software testing, as shown in Figure 1.
Software development consists of three phases of transformation. The first phase is from the system requirements description (SRD) to the software requirements specification (SRS). The SRD is normally written in a natural language by the system designers. In this phase, the unambiguous SRS written in a mathematical notation can be derived after the appropriate hardware configuration has been identified for the SRD. The second phase is from the SRS to the software design description (SDD). In this phase, software units can be specified using software design techniques such as software modularization and information-hiding to ease future maintenance and change of requirements. The third phase is from the SDD to coding. In this phase, each software module specified in the SDD is implemented using a programming language.

Software verification is a static verification to check that the transformation between two successive phase is correct. Finally, software testing is a dynamic verification to check that the software is operating as expected on a pre-defined set of input data under a controlled environment.

This software engineering process is a forward-going process in which each higher level development phase is complete before the next lower level development phase in a "topdown" fashion. A development phase is complete if the documentation for the phase has been properly reviewed and verified. To provide flexibility for the scheduling of workload, judicious overlapping of the activities between two successive development phases is allowed as long as the completion of a higher level development phase precedes the completion of the next lower level development phase. The overlapping should be properly conducted to minimize unnecessary iterations among the development phases.

3. Application of a Safety-Critical Software Engineering Process

3.1. General Documentation Requirements

Documentation is an intrinsic part of the software development described in this paper. The design activity in each software development is required to be faithfully recorded in the associated documentation. To achieve this faithful recording, the most effective way is to ensure that the documentation is performed as the design progresses such that the

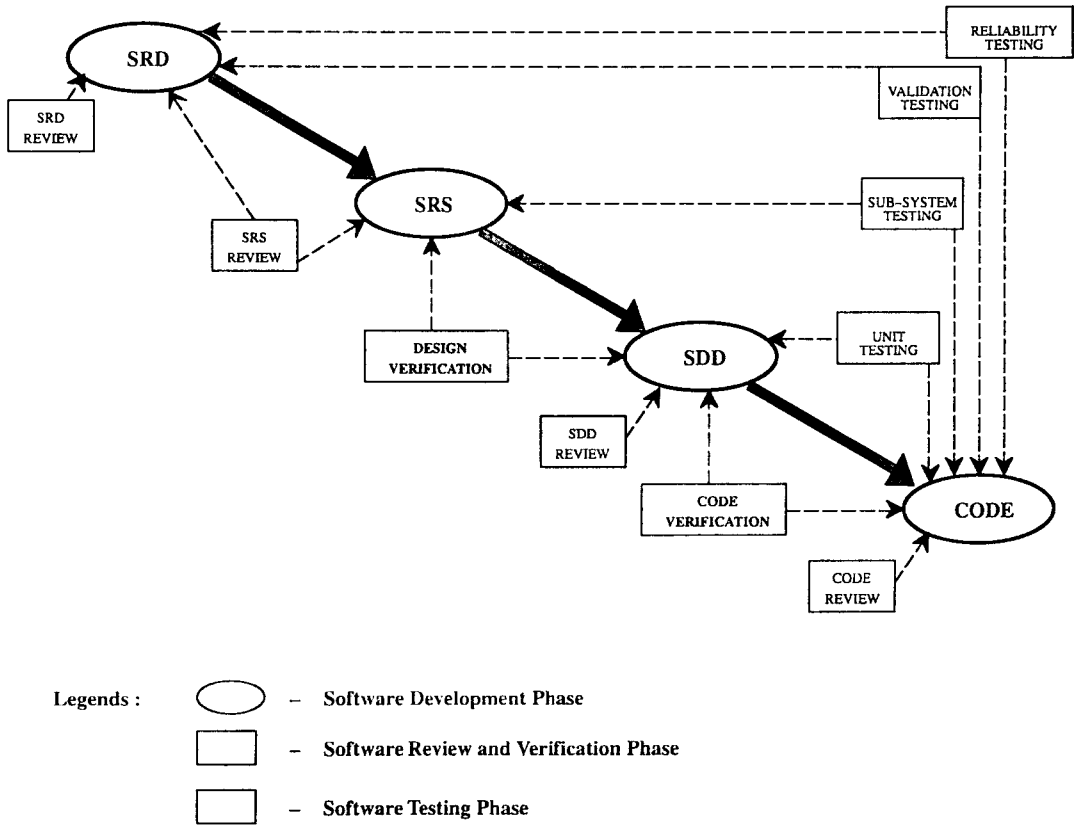

Fig. 1. A Quality Software Engineering Process

documentation for a software development phase is complete when the design activity for that phase is finished. In consequence, the documentation for a development phase is also regarded as a design medium for the phase.

The success of the documentation requires several important attributes, for example, accuracy, consistency, completeness and reviewability. It is a well-known fact that natural language is not the most appropriate language for technical specification. One of the often-quoted problems is the ambiguity in the specification using such language. Therefore, to achieve the required accuracy in technical specification, it is often necessary to choose an appropriate notation system based on mathematics.

Software specification can be difficult to read and understand, particularly when it contains decisions for different actions. This can be alleviated by using tabular formats as part of the notation used [4].

3.2. From SRD to SRS

As expounded by D.L. Parnas and his colleagues [5, 6], system behaviour specifiers are responsible for the specification of the system requirements. The specification is usually a description of the observable behaviour of the system. It should completely describe all monitored inputs, all controlled outputs and the required input/output relationship. Such relationship can be generalized by REQ(m^t, c^t), where REQ is the required system behaviour, m^t and c^t are time-functions for monitored and controlled variables.

Sometimes, natural constraints are also imposed on the required system behaviour. This is specified by a relation NAT(m^t, c^t). For a real-time system, the relationship must also include well-defined non-functional relationship to allow for concurrency and non-determinism, for example, of the occurrence of events. With the availability of the system requirements and the decision for the application of computers for the process control of the system, the system designers shall then be able to specify the required hardware for the system. The specifications for system requirements and the hardware together provide a basis from which the SRS can be derived.

From the viewpoint of safety-critical software development, the derivation of a complete and precise SRS is crucial to the success of subsequent software design. The effort of such derivation depends on the rigour in the system requirements and hardware specifications : the less the rigour, the more the effort. In current practices in industry, hardware specification is relatively more rigorous than system requirements specification for the following reason. Hardware specification often makes use of mathematics to describe well-defined behaviour of the hardware; mathematics has long been recognized for its effectiveness in making precise statements and definitions. System requirements specification is often described using a natural language ; ambiguity and inaccuracy in a natural language description are not easy to detect in such informal specification. In consequence, more reviews and discussions with the system behaviour specifiers are required for clarification during the derivation.

To avoid potential ambiguity and inaccuracy of an informal system requirements specification, the SRS should use a suitable notation based on mathematics. In safety-critical software development, each function in the SRS should be a total function. This is to ensure that complete coverage of the input domain for the function is performed to cater for all possible cases. For example, if the input domain of the function x is completely covered by three non-overlapping partitions $P_1(I)$, $P_2(I)$, $P_3(I)$ where P_i is a predicate over the input I, the following condition table can be used to specify x :

	$P_1(I)$	$P_2(I)$	$P_3(I)$
x =	e_1	e_2	e_3

which is equivalent to the simple mathematical statement of

$x = e_1$, if $P_1(I)$ = true

$x = e_2$, if $P_2(I)$ = true

$x = e_3$, if $P_3(I)$ = true

Event tables can also be used to describe the state transition of a system state. For example, if E_1 and E_2 are conditions for potential state transition of y, the event table below can be used to specify y :

	$@E_1$	$@E_2$
y =	e_1	e_2

which may be informally described as :
for a given time t,

$@E_i$ = true, if E_i at time t is true and
E_i at the time instant immediately prior to t is false
= false, otherwise
$y = e_i$, if $@E_i$ at time t is true
y holds the same value since the last time 't ('t < t) when $@E_i$ = true,
if all $@E_i$'s at time t are false.

To avoid conflicting state transition for y, $@E_1$ and $@E_2$ must not be both true at any time. Reference [5] provides formal definitions of various event tables.

To ensure that the SRS is complete, all monitored variables, and controlled variables, their relationship and system safety measures in an informal system requirements specification must be properly identified so that the monitoring rate of each monitored variable, the performance requirement of each controlled variable, the required relationship of each controlled

variable and its associated monitored variables, and safety net features can be derived in the SRS.

Finally, likely changes to the system requirements and system design must also be identified and highlighted in the SRS (possibly through appropriate discussions with the system behaviour specifiers and system designers) as they provide vital information for subsequent software design to cater for future changes.

3.3. From SRS to SDD

The objective of the step from SRS to SDD is to derive from the SRS a software design not only to capture all the requirements in the SRS, but also to cater for future anticipated changes and modifications. The functional aspects of the software design can be described in a four-variable model using monitored variables, input variables, output variables, controlled variables and their relationship [6]. Unlike the input and output boundaries of the SRS in which the monitored variables represent the system input boundary and the controlled variables represent the system output boundary, the input and output boundaries of the SDD are represented by the computer input and output variables. The computer input and output variables can be input and output registers or some other pre-defined memory locations for holding the input and output values to the computer. Once the computer input and output variables have been decided, the relation IN(m', i') from the monitored variables to the computer input variables and the relation OUT(o', c') from the controlled variables to the computer output variables must be defined so that the software behaviour SOF (i', o') can be derived. Note that m', i', o', c' are time-functions. To be acceptable, SOF must satisfy the following relation :

$$\forall m^t \forall i^t \forall o^t \forall c^t : IN(m^t,i^t) \wedge SOF(i^t,o^t) \wedge OUT(o^t,c^t)$$
$$\wedge NAT(m^t,c^t) \rightarrow REQ(m^t,c^t).$$

There are various approaches to software design. Perhaps, the most notable ones are functional decomposition and information-hiding decomposition. Functional decomposition is a well-known technique to capture the required functional behaviour in the resultant software, but it dose not generally appear to be effective to render the software amenable to changes and modifications. One plausible explanation is the over-dominance of the use of control flow paradigm in the functional decomposition. This often results in a premature commitment to a particular scheduling strategy which can lead to entanglement of normal logic for the required behaviour and scheduling logic for the required performance. Such entanglement could be a hindrance to future changes and modifications.

Information-hiding decomposition is a well-established discipline using a modular design approach [7]. In this discipline, software module decomposition is primarily driven by an information-hiding scheme in such a way that not only anticipated changes in the SRS shall be localized to meet future change and modifications, but also the resultant software system shall perform all the requirements in the SRS. For example, for a simple software used for the shutdown system of a nuclear power generation station, the application software can be decomposed into three modules, namely the hardware-hiding module, the behaviour-hiding module and the software-decision-hiding module. The hidden secrets of the hardware-hiding module are the characteristics specific to a particular hardware configuration. Examples of these secrets are I/O addresses and low-level I/O driven mechanisms. The hidden secrets of the behaviour-hiding module are how the required behaviour in the SRS is achieved. Examples are trip setpoint limits and duration for time delays. Finally, the hidden secrets of the software-decision-hiding module are choices of algorithms. An example is the computational method of a formula. Generally, the hardware-hiding module is designed for the relations IN and OUT, and the behaviour-hiding and the

software-decision-hiding modules are designed for the relation SOF. As the decomposition progresses, each module is further decomposed into sub-modules in the next lower level of the module decomposition hierarchy such that each sub-module hides some secrets of the corresponding module in the higher level decomposition hierarchy. The decomposition stops when sub-modules in the respective lowest level of the decomposition are deemed by the software designers that they are cost-effective in terms of efforts in documentation, implementation, future changes and modifications. The sub-modules in the lowest level of the decomposition are also known as leaf modules, each intended to correspond directly to a software module. The documentation of the information-hiding decomposition is in the form of a module guide. The module guide is an informal description of what each module does and what secrets it hides (without telling how the secrets are accomplished). It provides a guided tour of the decomposition and the intended goal of each leaf module. For maintenance purposes, it facilitates easy identification of the affected modules. With the module guide, the software designers can then proceed to the module interface and internal design of each leaf module using the technique of data abstraction.

The purpose of module interface design of a leaf module is to provide sufficient access programs (i.e. operations) to the user of the module to achieve what the module is intended to do without the user knowing how the secrets are dealt with inside the module. The access programs are the only means to manipulate the external behaviour of the module. There are various ways to specify the module interface. The most notable ones are the algebraic specification [8] and the trace specification [9]. Briefly speaking, the former is given by the module's signature (which roughly corresponds to the declaration of the access programs in the module) and a set of axioms relating the effects of access programs for the module, and the latter is given by the declaration of the access programs, the equivalence definition for each of a set of legal (i.e. canonical) traces of access program calls to the module and the evaluation function for each legal trace. Module interface description using either one of the above specifications is still perceived to be difficult to understand and to apply by the industry. For a simple shutdown system, the external behaviour of each module is deemed fairly simple and well-understood. Therefore, in practice, the module specification is described informally and with reference to the appropriate SRS functions.

The purpose of the module internal design is to maintain the internal state of the module to be consistent with the external behaviour of the module. The module's internal state is represented by suitable data structures local to the module for recording the effects of the access programs on the data structures. The module's internal behaviour can be specified using program functions to define the effect of each access program on the corresponding data structure in a way similar to the condition table for an SRS function. For example, the access program "SSeqChk" to update the internal state of the module "SequenceChk" (which checks the correct execution sequence of a group of access programs of other modules in the main line program) can be specified formally using the program function table below:

	'SeqChk < N	'SeqChk = N	'SeqChk > N
SeqChk'	'SeqChk + 1	0	'SeqChk

Where: the integer variable SeqChk represents the internal state of the module,
'SeqChk is the value before the invocation of SSeqChk,
SeqChk' is the value after the invocation of the SSeqChk and
N is a positive integer constant.

Note that to ease later coding effort, it may be advantageous to ensure that the syntax for the condition or expression in a table cell conforms to that

of a pre-selected programming language.

Finally, in a carefully conducted information-hiding decomposition and near the end of the decomposition, the scheduling logic should be confined to the schedule module. The specification for this module is algorithmic in nature. It prescribes an execution sequence of the access programs that can satisfy the monitoring rates of the monitored variables and the performance requirements of the controlled variables after some suitable data flow analysis of the system's performance requirements. This is possible because software scheduling for a shutdown system is normally required to be deterministic for safety reason. In consequence, future changes to performance requirements only involve the re-arrangement of execution sequence in this module after suitable review of the data flow analysis.

3.4. From SDD to Code

The translation of leaf modules to software modules can start after the design and specification of the leaf modules have been completed. For discussion purpose, we shall refer to a leaf module in the SDD as an SDD module and the code for an SDD module as a software module.

The translation is performed on a module-by-module basis and is mostly mechanical in nature. In this phase, the input domain analysis for an access program in the software module should no longer be a concern to the programmer because it has already been performed during the previous software design phase in specifying the program function for the access program in the SDD module. To the programmer, the translation of data structures in the SDD module is trivial because the corresponding data structures in the software module are, at the most, only different in syntax. The translation of the specification of the access program in an SDD module is also simple. When the specification is in the form of an algorithm, the translation is either a duplicate or a slightly more detailed code version of the algorithm for the access program in the software module. When the specification is in the form of a program function table, the function program table corresponds to a guarded command of the form :

```
if guard 1 action 1
| ....
| guard n action n
fi.
```

which requires that when the command is executed, *one and only one guard i (i.e. condition)* has a "true" value and the action i is performed [10]. Thus each condition cell of a program function table corresponds to a guard and each assignment to an action. Guarded command can be implemented by an *IF-THEN-ELSE* statement of a programming language.

The choice of a suitable programming language can also have a significant effect on the translation process. For a programming language with a well-defined semantics, quality compilers will conform to the language definition in their implementation. A programmer who is knowledgeable about one compiler can confidently apply the same knowledge in writing correct programs for another compiler. With a well-defined language, the chance of mis-use of a language construct is also reduced. For safety-critical software, the programming language should also be strongly typed to allow systematic checking on type-compatibility and valid range during program compilation and program execution. This enhances the safety aspect of the software without placing extra burden on programming effort. Even though no programming language is entirely safe for safety-critical software application, proper subset of suitable and well-defined part of some programming languages can often be identified for use. Finally, to support software design using modular approach, the programming language should also provide facilities for modular programming to ease programming effort.

3.5. Verification

In a rigorous development of safety-critical software, verification activity is performed to ensure that each development phase is correct relative to its previous phase by validating the documentation between the two phases.

The top-level verification starts after the SRS has been derived from the SRD containing the system requirements and the system design specifications. The verification is in the form of discussion and review sessions among the SRS specifiers, the system requirements specifiers, the system design specifiers, and the reviewers to ensure that the intended functional, performance and safety aspects of the system requirements are correctly translated into the SRS. Note that from time to time during the preparation of the SRS, the SRS specifiers may seek clarification from the system specifiers on some interpretation of an informal description of the system.

The next level verification between the SDD and the SRS starts after the SDD has been reviewed and completed. The verification is to demonstrate, mathematically or through rigorous argument, that the behaviour of each control variable in the SDD is an acceptable behaviour as required by the SRS for the controlled variable. This normally involves the identification of functions or groups of functions for the behaviour of the controlled variable in the SDD such that each function or group commutes with (i.e. is equivalent to) the corresponding function or group in the SRS through suitable logical manipulation. The verification also includes the checking of the execution schedule in the schedule module to ensure that the performance requirement of each controlled variable can be met assuming that the actual time to execute each program loop in the schedule is within the specified program loop time in the SDD.

After all the leaf modules in the SDD have been translated into software modules for a programming language, the last level verification between the SDD and the code can take place. The verification is performed on a module-by-module basis, i.e. each software module is compared against the leaf module in the SDD to show that the software module is a correct implementation of the leaf module. In general, the comparison involves two steps. The first step is to abstract a program function for each access program in the software module. This step is omitted if the access program is specified as an algorithm. The second step is to show that the data structures in the software module are equivalent to those in the leaf module and that each abstracted program function of the software module is logically equivalent to the corresponding program function of the leaf module. When the access program of the software module is a duplicate of the algorithm specified for the access program of the leaf module, inspection is adequate to confirm that they are the same. The verification is relatively easy and it can further be simplified by following some coding disciplines to ease verification effort. For example, the structure of the decision statements in the program should follow the structure of the condition cells in the program function table.

3.6. Testing

After completion of all software modules, various operational aspects of the application software can be tested to confirm the expected behaviour of the software. The objectives of the testing activities in the software engineering process are to expose latent defects and to provide assurance that the software satisfies all of the functional, performance and safety-related requirements, as specified in the SRD, SRS and SDD. The software verifiers are responsible for leading and conducting testing, and for producing the required documents. Software designers who participate in the implementation of the software should not involve in the following testing activities: preparation of test plan and analysis of test results. Unit testing, sub-system testing, validation testing and reliability testing should be performed for the testing

of a safety-critical software.

During unit testing, each module is individually tested to verify the consistency with the SDD. All important processing paths through the module are tested for expected results. Unit testing starts with test plan which describes specific classes of tests and test data. Once a test plan is prepared, test cases are generated. Test data should be carefully designed to exercise in-bounds and out-bounds conditions. Expected results for all test cases must be defined in advance based on the SDD. In many cases, programs known as "drivers" are used to exercise the module and record and report the results.

Sub-system testing is intended to check the interaction among the application software, the pre-developed software and target hardware. During the sub-system testing, the software system is assembled and tested in a systematic manner. Sub-system testing is performed to test that the code meets the requirements specified in the SRS using "bottom-up" techniques. The bottom-up approach incorporates modules into a build ; tests the build and then integrates it into the software structure. The build normally comprises a set of modules that perform a major function of the software system. Initially, the function may be represented a stub that is replaced when the build is integrated. To facilitate the production of test cases for sub-system testing, the test cases can be categorized to various groups. In general, test cases for safety-critical software system can be categorized into the following groups: functional behaviour compliance, timing and performance compliance, environmental interfaces mapping and operational constraints compliance, and fail safe compliance.

Validation testing is to test that the code with target hardware and pre-developed software meets the SRD. Validation testing relies on "black box" technique. Test values are supplied through the analog and digital inputs of the computer system, and the software is judged to have passed or failed on the basis of the output of the computer system. The values of the test inputs and of the expected outputs are based on the SRD.

Reliability testing is to demonstrate using statistically valid random testing that the code with target hardware and pre-developed software satisfies the software reliability requirements specified in the SRD. Like validation testing, reliability testing uses black box technique. It means that the tests are external to computer system. Reliability testing can be performed using the same test rig as validation testing. However, while validation testing typically verifies one function by evaluating the effect of changes to one input while other inputs are held constant, reliability testing simulates actual conditions, in which simultaneous changes occur on all inputs. The reliability under such conditions can be statistically demonstrated with some level of confidence by this testing. It is essential that the degree of uncertainty be quantified so that it can be shown to be consistent with the reliability requirements of the overall system.

4. Conclusions

This paper has discussed various aspects of a software engineering process for safety-critical software application. Such process is adapted in the production of safety-critical software for Shutdown System Number Two of the Wolsong 2, 3 & 4 nuclear power generation plants.

This process is significantly different from a conventional process in terms of rigorous development phases. In a conventional process, the SRS is usually non-existent and the SDD is predominantly described in a natural language or an informal pseudo-code. Without a precise SRS, the software designers have to rely on the informal SRD and relevant hardware manuals to design the software. In consequence, the software designers must accomplish two tasks: first, they have to understand the real and exact interpretation of the system requirements ; second, they have to capture this understanding (probably supplemented by their informal notes) of the SRD in

the design of the software. Without the help of precise notations while carrying out the two tasks, ambiguity and mis-understanding of the system requirements can be a significant contributing factor to the problems in the progress or quality of the software design. Such problems will be inherited in the coding phase. As often quoted, late design errors detected in the coding phase of a software development are often costly to correct. One of the important objectives in the process described in this paper is to avoid such costly errors by investing more effort in the front-end to obtain a precise SRS from which the software designers can focus their attention to design and to specify software modules for subsequent implementation.

This process is also significantly different from a conventional process in terms of software design techniques. In a conventional process, the decomposition of software design is primarily functional decomposition. Functional decomposition does not cater well for future maintenance and modifications. This may have a significant effect on the effort of future review, verification and testing of the software after the occurrence of maintenance or modifications. In the worst case, failure to localize the effect of software change may require that the resultant software must be completely reviewed again, re-verified and re-tested. Another important objective in the process described in this paper is to plan early in the software design phase to isolate the effects of expected changes in system requirements or hardware devices using the principle of information-hiding. Therefore when maintenance or modifications are required, only the affected software modules need to be reviewed, verified and tested.

During the course of the software development for the Wolsong project, some system requirements and hardware I/O devices do change. Experience shows that the software design using information-hiding principle does localize the effect of these changes. On the use of mathematical notations, experience in the project shows that they are very valuable to facilitate review, verification and testing once team members are familiar with the notation. This is due to the fact that time wasted in resolving ambiguity and mis-understanding of a specification is greatly minimized.

A common concern on the use of this process in the industry is perceived to be the significantly high cost of specification and verification involved. This need not be the case when the higher confidence required of the safety-critical software and the longer-term cost are taken into consideration.

Proper software tools to support the development phases play an important role in the design and the construction of safety-critical software by providing design assistance in organizing and maintaining the design database, editing the diagrams and the tables, facilitating the design documentation, coding the program and maintaining the configuration control. The availability of these tools can greatly reduce the software production cost. Identification of these tools and their effects on the software development phases shall be discussed in the near future.

Acknowledgements

The authors wish to thank all the staff of Shutdown Systems Computer Hardware & Software Branch of AECL (Atomic Energy of Canada Limited) CANDU for the Wolsong 2, 3 & 4 Project. In particular, they appreciate the following individuals for their encouragement and comments while writing this paper: N. Ichiyen, D. Chan, C. Choo, and T. Yip. The authors have benefited from the experience of software engineering process in the design and construction of the safety-critical software for the nuclear power plant while they participated in the Wolsong project.

References

1. D.L. Parnas, G.J.K. Amis, and J. Madey, "Assessment of Safety-critical Software", Technical Report 90-295, ISSN 0836-0227, TRIO, Queen's

University, Kingston, Ontario, Canada, December 1990.

2. D.L. Parnas and P.C. Clements, "A Rational Design Process : How and Why to Fake It", IEEE Transactions on Software Engineering, Vol. SE-12, NO. 2, February 1986.
3. G.J.K. Asmis and P. Wigfull, "The Process for Regulatory Approval of Safety-critical Software", COG CANDU Computer Conference, Markham, Ontario, November 1990.
4. K.L. Heninger, J.W. Kallander, D.L. Parnas, and J.E. Shore, "Software Requirements for the A-7E Aircraft", NRL Report 3876, Naval Research Laboratory, Washington, DC, November 1978.
5. A. John van Schouwen, "The A-7 Requirements Model : Re-examination for Real-Time Systems and an Application to Monitoring Systems", Queen's University, Kingston, Ontario, Canada, April 1990.
6. D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering (Version 2)", CRL Report No. 237, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, Sept. 1991.
7. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, 15, 12, December 1972, pp. 1053−1058.
8. J. Goguen, J. Thatcher, E. Wangner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types", Current Trends in Programming Methodology, iv, Raymond Yeh, Editor, Prentice-Hall, 1978, pp. 80−149.
9. D.L. Parnas, Y. Wang, "The Trace Assertion Method of Module Interface Specification", Technical Report 89-261, Queen's University, Kingston, Ontario, Canada, 1989.
10. E.W. Dijkstra, "A Discipline of Programming", Prentice Hall, Englewood Cliffs, New Jersey, 1976.