Proceedings of the Korean Nuclear Society Spring Meeting
Cheju, Korea, May, 2001

Software Reliability Evaluation of Digital Plant Protection System Development Process Using V&V

Na Young Lee[1)], Il Soon Hwang[1)], Seung Hwan Seong[2)], Seung Rok Oh[3)], and Jin Young Choi[4)]

1) Seoul National University, Department of Nuclear Engineering
56-1, Shinlim-dong, Kwanak-Gu, Seoul, Korea 151-742
2) Korea Atomic Energy Research Institute
3) Dankook University, Department of Electronic Engineering
4) Korea University, Department of Computer Science

Abstract

In the nuclear power industry, digital technology has been introduced recently for the Instrumentation and Control (I&C) of reactor systems. For its application to the safety critical system such as Reactor Protection System(RPS),a reliability assessment is indispensable. Unlike traditional reliability models, software reliability is hard to evaluate, and should be evaluated throughout development lifecycle. In the development process of Digital Plant Protection System(DPPS), the concept of verification & validation (V&V) was introduced to assure the quality of the product. Also, test should be performed to assure the reliability. Verification procedure with model checking is relatively well defined, however, test is labor intensive and not well organized. In this paper, we developed the methodological process of combining the verification with validation test case generation. For this, we used PVS for the table specification and for the theorem proving. As a result, we could not only save time to design test case but also get more effective and complete verification related test case set. Add to this, we could extract some meaningful factors useful for the reliability evaluation both from the V&V and verification-combined tests.

1. Introduction

By introducing digital technology into nuclear I&C system, complicated safety algorithm, improved availability, easier maintenance and installation etc. can be obtained. Digital system possesses advantages over its analog counterpart, however, software reliability is the key issue especially when applying it to Nuclear Power Plant(NPP) safety systems. Therefore the concept of software verification and validation (V&V) was introduced to assure the reliability of safety critical systems.

At present, there is no proven, objective method to measure the reliability of a software-based product to the level of confidence required for safety-critical nuclear applications.

The Electric Power Research Institute (EPRI) classified nuclear I&C systems for digital

I&C upgrades. RPS is classified as the most safety-critical item because of the crucial role played by the RPS and short time response required of reactor trips. For the RPS, some additional recommendations are made to add confidence from the point of view of both the utility and the NRC[1].

a. Validation testing should include abnormal and faulted conditions. It should also include randomly generated test cases, to increase coverage and to avoid any manual bias in defining test cases.

b. Unambiguous formats, using tabular or mathematical representations, should be used to express required behavior in the requirements document. Reliance on natural language should be minimized.

c. More extensive structural testing should be performed to exercise each branch of each decision statement.

d. Reviewers should report to an organization separate from that of the developers.

To satisfy above recommendations, we determined to use rigorous formal method to specify and verify the requirements and design description. From the verification condition and specified tables, we could generate the validation condition i.e. test case generation. Some formal method tools provide counterexample from the safety condition, which can be used as a test case for the abnormal condition. And as a structural test, we used tool to check not only decision nodes, but also coding blocks and variable usage in the code.

In this study, we wanted to show the test coupled with V&V procedure to meet above recommendations.

2. Verification Procedure

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase, while validation is the process of evaluating a system or component at the end of the development process to evaluate whether it satisfies specified requirements.

EPRI classified and assessed various V&V methods corresponding to their characteristics[2].

Among V&V analysis methods, formal methods can best meet the requirements of safety critical system, for the language is usually based on mathematical exactness and the ability for reasoning. In V&V, of particular importance is the form of expression used for the software requirements because most of the methods and tools at later stages in the life cycle can only be used provided that the appropriate requirements tool is used from the early stages. Furthermore, the very act of expressing requirements in a rigorous structure and format is itself an important part of requirements analysis.

The major usage of using formal methods can be classified into specification and verification. Specification itself is a kind of verification and provides the function to check types and formats. Formal verification is the process showing, by means of formal deduction,

that a formal design specification satisfies its formal requirements specification[3].

Verification tools are largely classified into theorem proving and model checking. Theorem proving refers to the use of axioms and proof rules to prove the correctness of systems. For it is time-consuming and needs experts, it is considered better to use for reasoning about infinite state systems, which are hard to verify with model checking method. Model checking is a technique for verifying finite state concurrent systems. By using the model checking tools, verification can be performed automatically. Generally, model checking is preferable to theorem proving because of its automatic verification ability. For the critical applications, however, theorem proving is necessary for complete verification[4]. Rigorous verification procedure should be analyzed as one of the metrics for the evaluation of system reliability, which referred later.

In this paper, we chose to use PVS, formal theorem prover, as a specification and verification tool to support the work performed with model checking method by complete verification. By using PVS we could connect the V&V procedure with test case generation as well, which is described in 3.2.

2.1 Digital Plant Protection System Description

Reactor Protection System(RPS) protects the core fuel design limits and reactor coolant system boundary by tripping the reactor when the plant operation exceeds these limits.

Reactor trip is provided through an interface with the Reactor Trip Switchgear System. DPPS has two important parts for this operation, bistable and coincidence part. Bistable processors generate trips based on measurement channel digitized value such as pressure, level or temperature exceeding a digital setpoint. These trip outputs are directed to the 2 out of 4 coincidence logic processors. The outputs are then organized into initiation logics for tripping the reactor and activation of ESF trains. Fig. 1 shows the schematic diagram of DPPS.

2.2 Characteristics of PVS

PVS is a verification system for writing formal specifications and checking formal proofs. It was developed at SRI. The distinguishing feature of PVS is its synergetic integration of an expressive specification language and powerful theorem-proving capabilities. PVS provides an expressive specification language with various types. PVS typechecker generates the proof obligation automatically to cope with undecidability of typechecking and it allows PVS to enforce very strong checks on consistency and other properties (such as preservation of invariants) in an entirely uniform manner. PVS also has a powerful interactive theorem prover/proof checker[5]. PVS is designed to help in the detection of errors as well as in the confirmation of “correctness” by;

i) Rich type-system and rigorous typechecking

ii) Provide rich-mechanisms for conservative extension; definitional forms that are guaranteed to preserve consistency

iii) Provide an effective theorem prover; process of "challenging" specifications of which form "If this specification is right, then the following ought to follow"

2.3 Specification and Verification Using PVS

In this paper, we used a simplified example of DPPS. From various RPS signals, we chose to use 'High Logarithmic Power Level' signal, which is one of the analog signals. DPPS has 4 channels and 12 signals from each channel. Each signal would be compared with its trip setpoint in the bistable processor. Each 'trip' signal from 4 channels would be transmitted to coincidence processor, respectively. In the coincidence processor, voting(2 out of 4) function would be performed to determine whether the trip signal is valid or not. The function of the local coincidence logic is to generate a trip initiation signal whenever two or more like bistable signals are in a tripped condition. It can be some help to refer to Fig. 1.

Fig.2 shows the requirements specification of the simplified example of DPPS using PVS tool. In this, we specified the theorems that should be satisfied, and showed the theorem proving of it by PVS theorem prover in Fig. 3.

RPS logic is not so sophisticated, thus we can describe most of them as 'if..then..else' form. Using table can be thought of as reasonable for the representation of this type of simple conditions. Table format is easier to express and read. PVS supports table consistency conditions, which require pairwise conjunction of a set of formulas to be false(Mutual Exclusion Property or disjointness) and disjunction of a set of formulas to be true(Coverage property)[6].

Conditions in the above example can be represented as Table 1.

We translated this into PVS and verified the theorem, which is represented in Fig. 4 and 5.

From this PVS specification, we can verify consistency and correctness. The tables and theorem specified and verified could be used as a decision table in the testing, which would be described below.

3. Validation Condition Generated from Specification and Verification condition

In the test case design, ad hoc testing or error guessing is an informal testing technique that relies on inspiration, creative thinking, and brainstorming to design tests. If we use formal design technique, it would provide a higher probability of assuring test coverage and reliability[7].

3.1 Test Classification

3.1.1 Black-Box Testing(Functional)

Test conditions developed based on the program or system's functionality; input and output set is needed but tester does not know how the program or system works. Its advantage is that we can test what the program or system is supposed to do, and it is natural and understood by everyone. But exhaustive testing is not achievable, because this requires that every possible

input condition of combination be tested.

3.1.2 White-Box Testing(Structural)

By examining paths of logic, test conditions can be designed, i.e. by examining the logic of the program or system, without concern for the program or system requirements, we can drives test data. Specific examples in this category include basis path analysis, statement coverage, branch coverage, condition coverage, and branch/condition coverage. By this, we can focus on the produced code and that thoroughly. But has some disadvantages as follows;

i) It does not verify that the specifications are correct. i.e., it focuses only on the internal logic and does not verify the logic to the specification

ii) There is no way to detect missing paths and data-sensitive errors

iii) We cannot execute all possible logic paths through a program because this would entail an astronomically large number of tests

3.1.3 Gray-Box Testing (Functional and Structural)

Black-box testing focuses on the program's functionality against the specification. White-box testing focuses on the paths of logic. Gray-box testing is a combination of black- and white-box testing. The tester studies the requirements specifications and communicates with the developer to understand the internal structure of the system to clear up ambiguous specifications and "read between the lines" to design implied tests[6]. In the basis of thorough understanding of the details of requirements and program structure acquired during formal specification and verification process, we can eliminate a number of useless, redundant tests.

3.2 Test Case Preparation for Functional Test

In the above RPS example, DPPS has 4 channels and 12 signals in each channel respectively. 12 signals are 10 analog signals and 2 digital signals. If we assume that all the signals have Boolean value determined at the bistable processor, 12 signals could have 2^{12} different cases, which can be expanded to $(2^{12})^4$ for all four channels. If we assume the real case that includes bypasses and some other trip conditions, however, there would be too many states to test. But after reviewing the system thoroughly, we could know that changes of all 12 signals are independent each other. So the test cases can be reduced to $2^4 \times 12=192$, which is appropriate for preliminary test. In this calculation, we excluded the operating bypass and channel bypass, which is specified in the example, for convenience.

As we specified in Fig. 4, we could generate a table specified in PVS, and use this as a decision table for test. Decision table can be used to represent and analyze logical relationships, so it is appropriate to our case. 'If..then..else' form used in our example could be directly translated to table form as shown in Table 2. In preparing this decision table, we can generate test cases effectively on the basis of understanding acquired during specification

and verification. And by specifying and verifying table with formal method, we can assure consistency and correctness of the table by theorem proving. When drew up table specification, we classified the cases as for unit test and for integration test. Functional test was performed based on these decision tables.

3.3 Structural Test

Structural coverage testing identifies program constructs that may be exercised during program execution and determines which of these constructs are in fact exercised by a set of tests. Test coverage metrics are a device to measure the extent to which a set of test cases covers a program. Here we performed the structural test to show how the functional test cases generated from the verification can cover when used as structural test input. So we chose a simple and available test tool. In this paper we used the tool named ‘ ATAC’ as a test coverage analysis tool developed at Bellcore. ATAC measures how thoroughly a program is tested by a set of tests using data flow coverage techniques, identifies areas that are not well tested, identifies overlap among tests, and finds minimal covering test sets. By using ATAC, we can analyze the coverage of blocks of consecutive statements, branch decisions, and various combinations of assignments and uses of variables.

In this study, we applied ATAC to the partial(for the bistable function) code, which is shown in Fig. 6. Block coverage counts the branch free executable code fragments that are exercised at least once. If block coverage is less than 100%, there are statements that are not exercised by any test. Decision coverage counts the number of branches that have been followed at least once. C-uses, or computational variable use coverage count the number of combinations of an assignment to a variable and a use of a variable in a computation that is not part of a conditional expression. P-uses, or predicate variable use coverage count the number of combination of an assignment to a variable, a use of the variable in a computation, and all branches based on the value of the conditional expression. In fig 6, *82(214/260) means 214 blocks were covered from all 260 blocks so the total block coverage is 82%.

This code is composed of independent parts, such as bistable part or coincidence part. Only the outputs calculated from previous part would be transferred to the next stage, so it is not so much different whether we apply ATAC to the whole code or partly.

From this structural test applied to the bistable part of code, we could get over 80% block coverage by only 10 or more functional test input sets generated from the verification. From this, we could know that test input sets used to extend test coverage are somehow overlapped with functional test cases in this example. By using some more test inputs for it, we could get much more test coverage, but that is not our objective; so have not been covered in this paper. Here, we just showed one of the usage of test cases generated from the V&V procedure.

4. Discussion

Until now, we went through the process of DPPS development life cycle. By using PVS,

we could get rigorous specification and ‘completeness verified’ test case by table format. Test case generation from the specification is a kind of gray-box approach. In the basis of the test cases generated from the V&V process, we could perform functional tests. Structural test was performed to see the test coverage when using test cases generated from V&V. By only using the functional test cases generated from V&V process, we could get over 80% test coverage.

As we mentioned above, it is too late to wait until the testing phase to collect and assess software quality information; also, it is inadequate only to use the results obtained from testing to assess a software safety or reliability. Littlewood suggested holistic model which includes not only product but also process and people/resource metric to assess reliability[8]. For a perfect metric table, we should check and make out all the factors needed to a reliability assessment. But ‘perfect’ cannot be achieved, so it is advisable to take the greatest possible items not overlapped, then complement the table gradually.

As a complement, we added formal specification process as a process metric, and verification of the properties as a product metric. In our approach, we used PVS as a formal method to specify and verify the requirements and design. Using formal method for specification can be one of the process metric. And verified characteristics such as completeness and consistency can support as a product metric. Also, both functional and structural test results can be treated as product metrics.

As a result, we could extract some factors from the approach described above, which can be used to evaluate the reliability of the developed system. The sources of data for reliability assessment are abbreviated in table 3; these factors can be subdivided into details. This type of table would help summarize the results from the overall approach and identify missing factors. Consequently, we can expand this table, to use as a well-organized metric for reliability assessment.

5. Conclusion

In general, V&V is a development procedure to assure there is no error in the developed system, while test is performed to show there is error. Though we cannot assure there is no error by testing, but test can be used to quantify the reliability of the developed system. V&V and test have some different characteristics, but both of them are recommended to perform to assure reliability.

In this approach, we used PVS as a formal method to perform verification, then, extracted system information from this, so we could perform test with gray-box approach. And formally specified and ‘completeness-proved’ decision tables generated in this process were used as functional test cases. We combined the tabular specification and theorem originally specified for verification to use as a functional test cases. Rigorous verification in the requirement phase makes the test set reasonable for functional test. Performing structural test with generated functional test set shows as large as over 80% test coverage. We assumed that the

thorough understanding of the system and code is the major cause of it.

In this study, we proposed V&V combined test case generation process, and as a result, enables the software development lifecycle as a whole can be treated one of software reliability metric.

Acknowledgment

This work was financially supported via the Korea Institute of S&T Evaluation and Planning (KISTEP) as a part of Strategic National R&D Program.

Reference

[1] EPRI, Handbook for verification and validation of digital systems Vol. 1: Summary, 1994

[2] EPRI TR-103331-V1 Research project 3093-01, "Guidelines for the verification and validation of expert system software and conventional software", Vol. 1, 1995

[3] John Rushby, Formal methods and their role in the certification of critical systems, Technical Report CSL-95-1, 1995

[4] E.M.Clarke, et.al. “Model Checking”, The MIT Press, 1999

[5] J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas, A tutorial introduction to PVS, SRI international, 1995

[6] S. Owre, N. Shankar, J. M. Rushby, PVS Language Reference Ver. 2.3, SRI international, 1999

[7] W.E.Lewis, "Software testing and continuous quality improvement", CRC Press LLC, 2000

[8] Debra S. Herrmann, “Sample implementation of the Littlewood holistic model for assessing Software quality, Safety and Reliability”, Proceedings annual reliability and maintainability symposium, pp.138-148, 1998

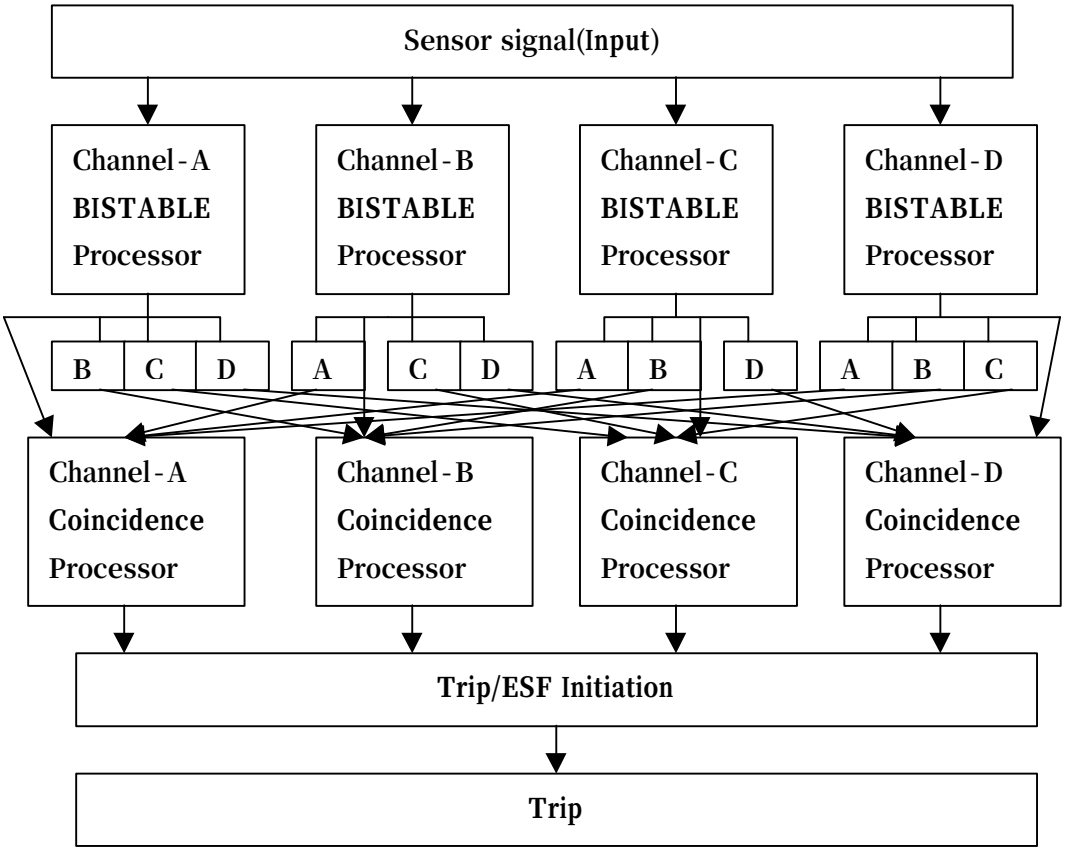

Fig. 1 DPPS system diagram

```
Logic_ChA_bi_a1_1    : int =    COND
ChA_op_by_a1    -> 0,
ELSE ->( COND ChA_bi_a1_1>a1_tripset -> 1, ChA_bi_a1_1<=a1_tripset -> 0 ENDCOND)
ENDCOND
…
Coin_ChA_a1 :bool = COND
Logic_ChA_bi_a1_1+Logic_ChB_bi_a1_1+Logic_ChC_bi_a1_1+Logic_ChD_bi_a1_1>=2 -> TRUE,
Logic_ChA_bi_a1_1+Logic_ChB_bi_a1_1+Logic_ChC_bi_a1_1+Logic_ChD_bi_a1_1<2 -> FALSE
ENDCOND
…
Val_LOGIC_ChA : THEOREM ( Logic_ChA_bi_a1_1=1) implies (ChA_op_by_a1=FALSE
  AND ChA_bi_a1_1>a1_tripset)

Trip_cond1 : THEOREM ((ChA_op_by_a1)AND(ChB_op_by_a1)) AND ((NOT ChC_op_by_a1)
AND(NOT ChD_op_by_a1)) AND ((ChA_bi_a1_1)>(a1_tripset) AND (ChB_bi_a1_1)>(a1_tripset)
AND (ChC_bi_a1_1)>(a1_tripset) AND (ChD_bi_a1_1)>(a1_tripset)) IMPLIES (Coin_ChA_a1)
```

Fig. 2 Software Requirement Specification using PVS

```
Val_LOGIC_ChA :
  |-------
1     (Logic_ChA_bi_a1_1 = 1) IMPLIES
          (ChA_op_by_a1 = FALSE AND ChA_bi_a1_1 > a1_tripset)
Rule? (grind)
Logic_ChA_bi_a1_1 rewrites Logic_ChA_bi_a1_1
   to COND ChA_op_by_a1 -> 0, ChA_bi_a1_1 > a1_tripset -> 1, ELSE -> 0
        ENDCOND
Logic_ChA_bi_a1_1 rewrites Logic_ChA_bi_a1_1
   to COND ChA_op_by_a1 -> 0, ChA_bi_a1_1 > a1_tripset -> 1, ELSE -> 0
        ENDCOND
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Fig. 3. (a) Theorem proving of Val_LOGIC_ChA

```
Trip_cond1 :
  |-------
1     ((ChA_op_by_a1) AND (ChB_op_by_a1)) AND
          ((NOT ChC_op_by_a1) AND (NOT ChD_op_by_a1)) AND
            ((ChA_bi_a1_1) > (a1_tripset) AND
               (ChB_bi_a1_1) > (a1_tripset) AND
                 (ChC_bi_a1_1) > (a1_tripset) AND (ChD_bi_a1_1) > (a1_tripset))
          IMPLIES (Coin_ChA_a1)
Rule? (grind)
Logic_ChA_bi_a1_1 rewrites Logic_ChA_bi_a1_1
   to 0
Logic_ChB_bi_a1_1 rewrites Logic_ChB_bi_a1_1
   to 0
Logic_ChC_bi_a1_1 rewrites Logic_ChC_bi_a1_1
   to 1
Logic_ChD_bi_a1_1 rewrites Logic_ChD_bi_a1_1
   to 1
Coin_ChA_a1 rewrites (Coin_ChA_a1)
   to TRUE
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Fig. 3. (b) Theorem proving of Trip_cond1

```
Logic_ChA_bi_a1_1: int=TABLE
  %-------------------------------------------------------------
                                ChA_op_by_a1
  %-------------------------------------------------------------
                                |[TRUE | FALSE]|
  %-------------------------------------------------------------
  |ChA_bi_a1_1>a1_tripset      |   0    |   1    ||
  %-------------------------------------------------------------
  |ChA_bi_a1_1<=a1_tripset     |   0    |   0    ||
  %-------------------------------------------------------------
 ENDTABLE

VAL_logic_ChA : THEOREM (Logic_ChA_bi_a1_1=1) implies
       (ChA_op_by_a1=FALSE AND ChA_bi_a1_1>a1_tripset )
```

Fig. 4. PVS specification using table format

```
VAL_logic_ChA :
|-------
1     (Logic_ChA_bi_a1_1 = 1) IMPLIES
(ChA_op_by_a1 = FALSE AND ChA_bi_a1_1 > a1_tripset)
Rule? (grind)
Logic_ChA_bi_a1_1 rewrites Logic_ChA_bi_a1_1
to 0
Logic_ChA_bi_a1_1 rewrites Logic_ChA_bi_a1_1
to 0
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Fig. 5. Theorem proving of VAL_logic_ChA

Blocks(%)	Decisions(%)	C-uses(%)	P-uses(%)	Function
100(17)	100(4)	100(11)	100(8)	main
100(30)	100(16)	92(24/26)	100(32)	initialize
100(7)	100(3)	100(2)	100(6)	Initialize_measure
100(25)	100(8)	100(28)	100(12)	input
80(48/60)	69(40/58)	47(117/249)	29(105/364)	bistable
72(68/95)	53(40/76)	61(99/162)	32(49/152)	trip_compare
73(19/26)	45(9/20)	67(22/33)	24(10/42)	manual_reset
*82(214/260)	65(120/185)	59(303/511)	36(222/616)	total

Fig. 6. Structural test coverage for unit program

Table 1. Conditions of DPPS example

Logic_ChA_bi_a1_1	ChA_op_by_a1	
	TRUE	FALSE
ChA_bi_a1_1>a1_tripset	0	1
ChA_bi_a1_1<=a1_tripset	0	0

Table 2. Decision tables

(a)For bistable test

Condition	ChA_op_by_a1			
	TRUE		FALSE	
ChA_bi_a1_1>a1_tripset	TURE	FALSE	TRUE	FALSE
Action				
Logic_ChA_bi_a1_1	0	0	1	0

(b) For coincidence test

Condition																
Logic_ChA_bi_a1_1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Logic_ChB_bi_a1_1	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
Logic_ChC_bi_a1_1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
Logic_ChD_bi_a1_1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Action																
Trip_initiation	T	T	T	T	T	T	T	F	T	T	T	F	T	F	F	F

Table 3. Factors extracted from each stage of development life cycle.

		Completeness	Consistency	Fairness	Safety	Coverage	Fault
Formal method	Requirement						
	Design						
Structural test	Code						
Functional test	Test						