An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming

Prepared by

Lamees Alhazzaa

Introduction

- Multi-version or N-version programming has been proposed as a method of providing fault tolerance in software.
- It requires
  - Separate independent preparation of multiple versions of a piece of software for some application.
  - Versions are executed in parallel in the application environment.
  - Each receives identical input and produces its version of required output.
  - Outputs are collected by a voter, where they all should be the same.

N-Version Programming

- **Separate development can start at different points in the software development process.**
- **Each version should provide the same functional capability; use a common form of system requirements document.**
- **Coordination must exist when versions must provide data to the voter.**
- **Design specification will be redundant and independent, to highlight common design faults.**

N-Version Programming

- N-version programming benefits
  - Improves reliability.
    - Versions fault assumed to be independent.
    - Faults occur randomly.
    - Probability of failure for several versions on the same input, is very small.
  - Redundancy is an important technique for achieving fault tolerance
    - by determining and minimizing common failure modes.
- N-Version programming biggest issue
  - Cost of developing multiple versions of software

N-Version Programming

- Related work
  - Ramamoorthy: dual specification approach
  - Kelly and Avizienis: separate specification written by the same person
  - Anderson and Lee: summarized difficulties on N-Version Programming.
- This experiment
  - A Statically Rigorous test of independence was the major goal of the experiment.

Experiment Overview

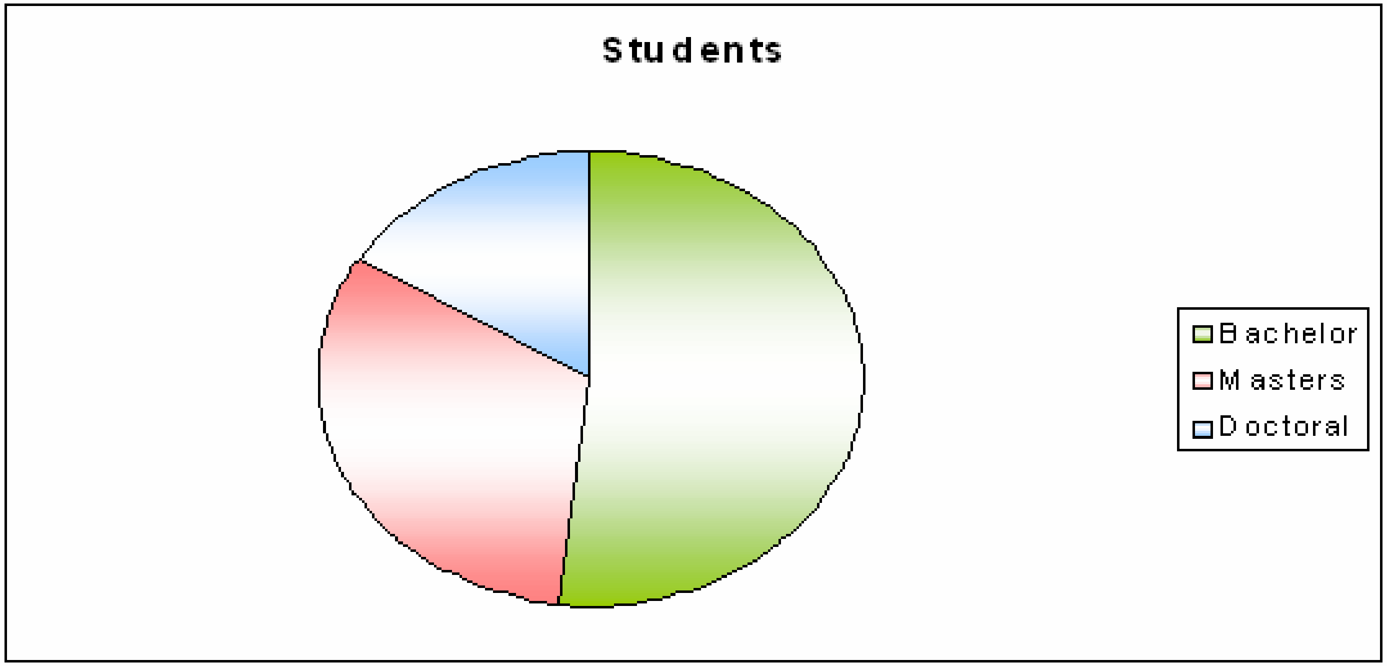
Universities	University of Virginia (UVA) University of California at Irvine (UCI)
Programmers	Graduate and Senior level students
Requirement	Write a program from a single requirement specification.
Problem	Sample Anti-missile system, which reads data that represents radar reflection of objects, then decide whether the object is a threat or not.
Result	Twenty seven programs, each of which was then subjected to one million randomly generated test cases

Experiment Description

1. RTI requirements re-written and debugged.
2. Programs were written in Pascal.
3. Goals were explained to programmers.
4. Programmers questions were sent by emails and responses were broadcast to all programmers.
5. Students were supplied with fifteen input datasets and expected outputs for use in debugging.
6. Debugged versions are tested.
7. Acceptance testing were then performed using a generator written specifically for this application.
8. Versions subjected to the experimental treatment.
9. Test driver was built to generate random radar reflections.
10. A golden version was used to compare the results of the other versions with.

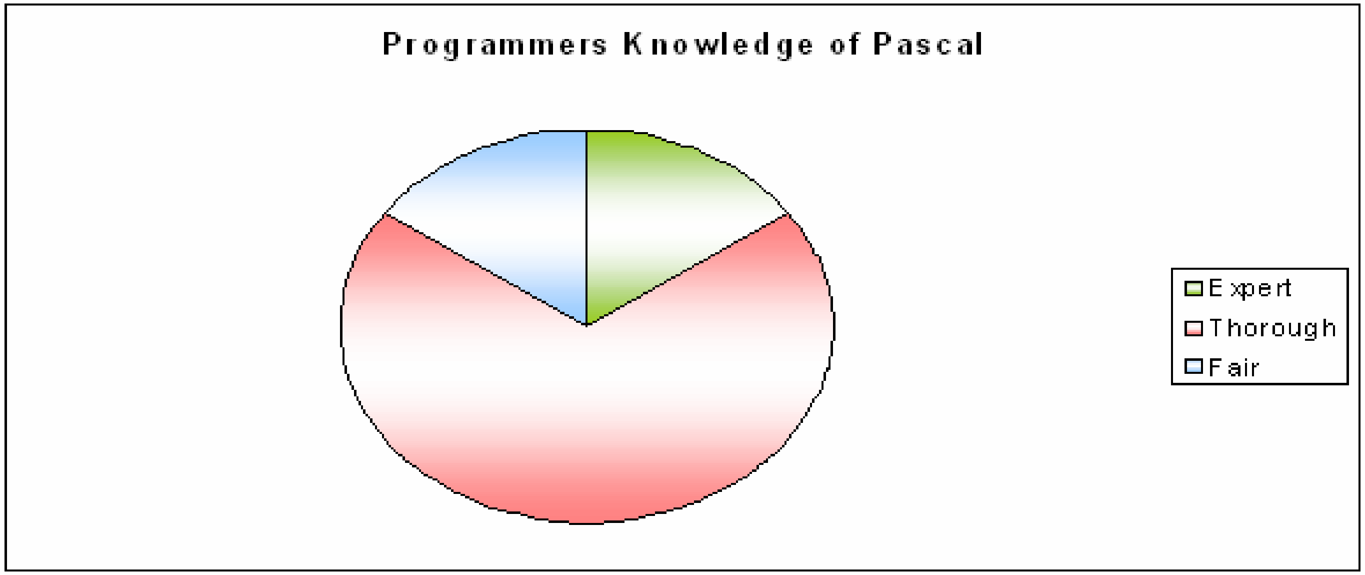
Programmer's background

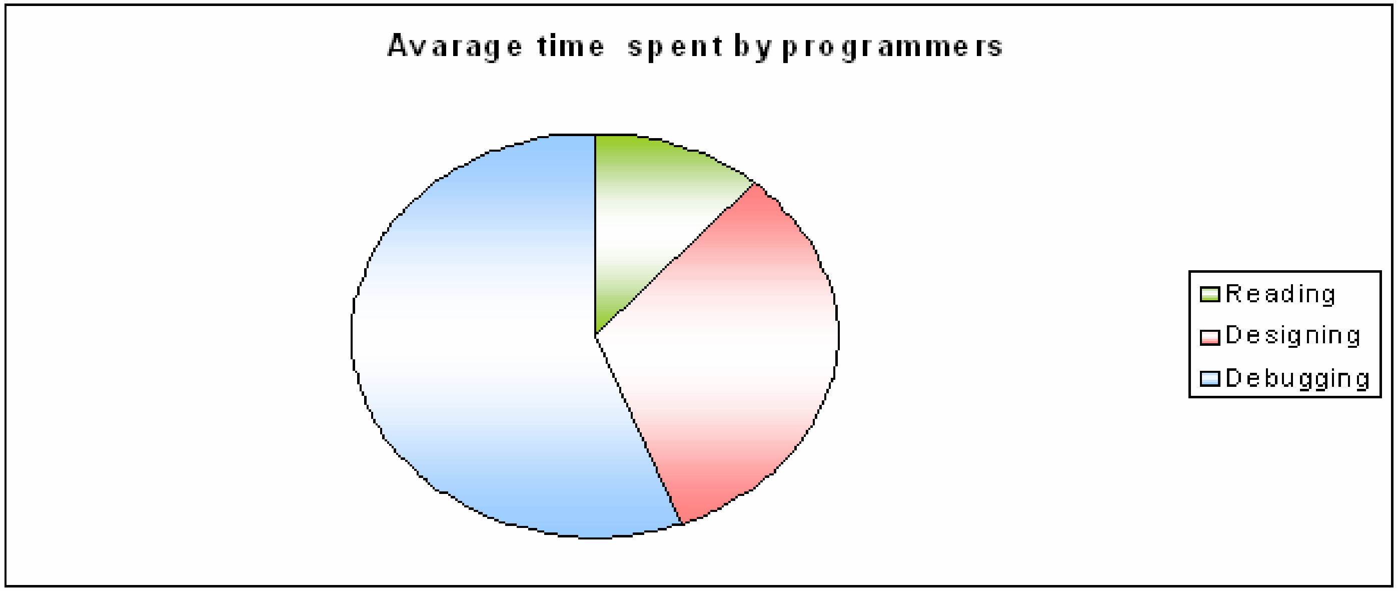
•A questionnaire of the programmers background was filled.



- mathematics, computer science
- astronomy, biology, environmental science, physics

Programmer’s background

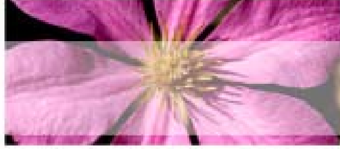


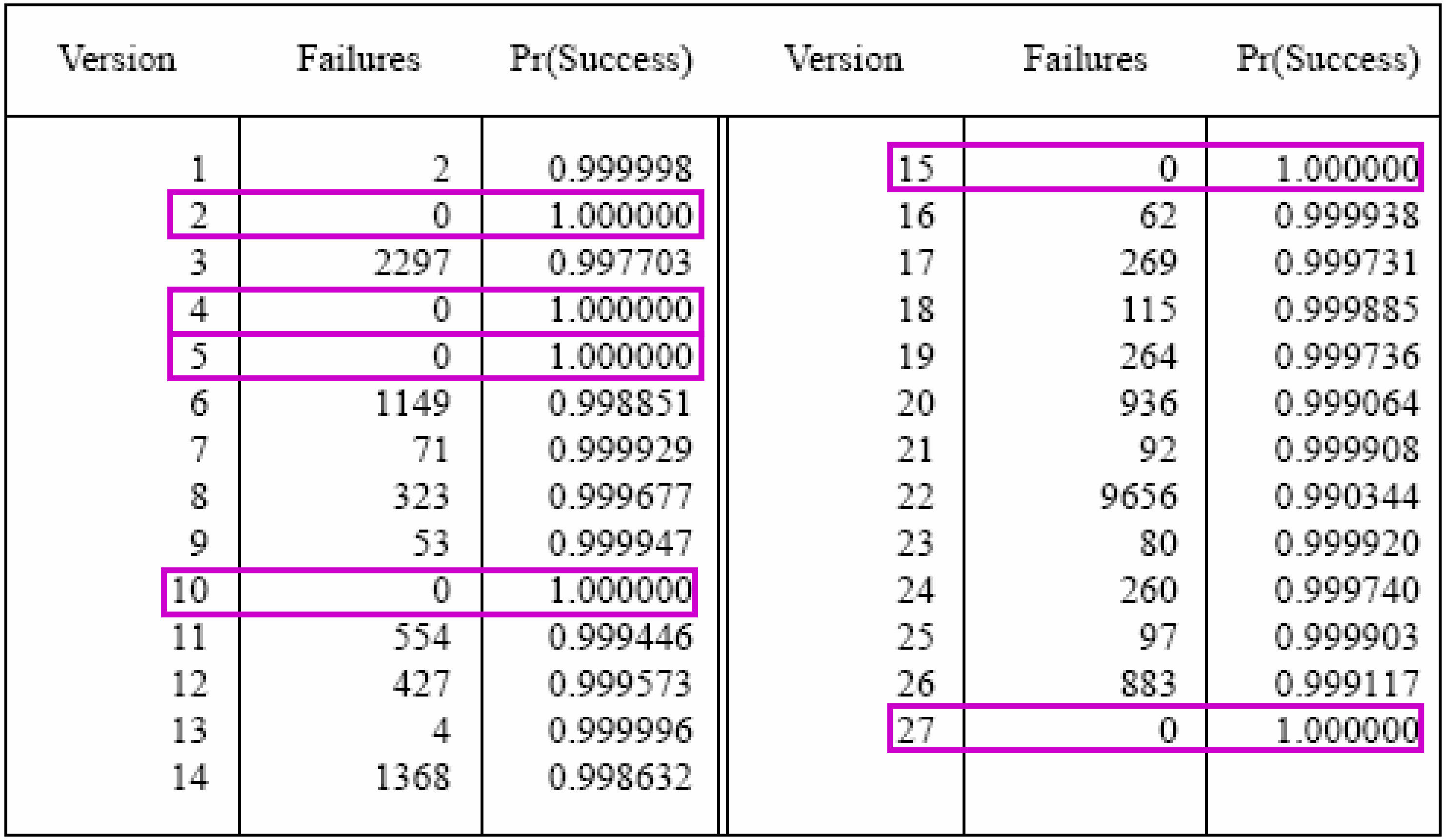


Experimental Results

- **The program calculates the results from the simulated radar tracking data and various parameters, all of which randomly generated for each test case.**
- **The launch condition is the only true output of the program, which determines the detected objects (Boolean Type).**
- **A result is record a failure for a particular version if it causes some exception *such as negative root square*.**

Experimental Results





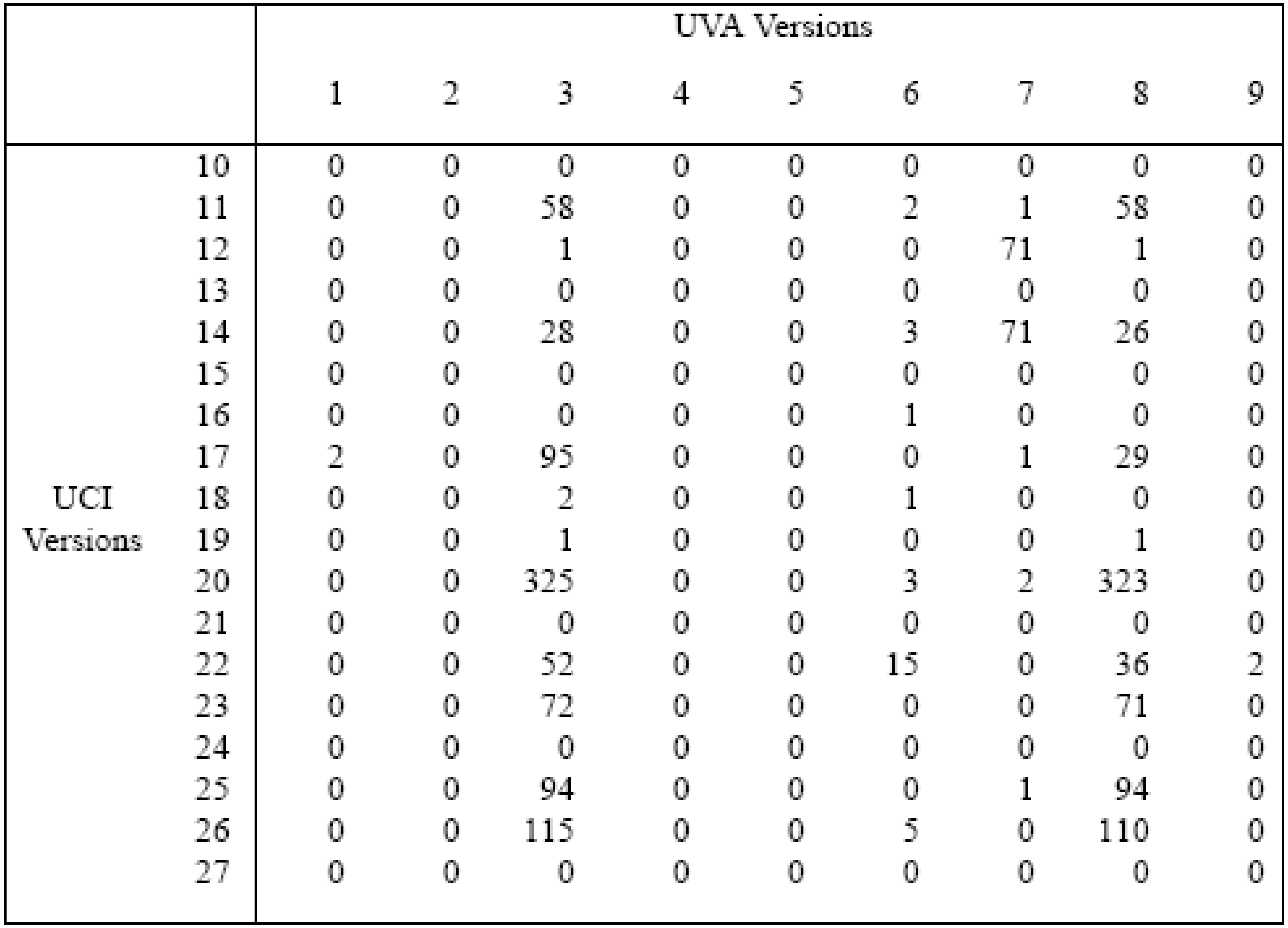
Version	Failures	Pr(Success)
1	2	0.999998
2	0	1.000000
3	2297	0.997703
4	0	1.000000
5	0	1.000000
6	1149	0.998851
7	71	0.999929
8	323	0.999677
9	53	0.999947
10	0	1.000000
11	554	0.999446
12	427	0.999573
13	4	0.999996
14	1368	0.998632
15	0	1.000000
16	62	0.999938
17	269	0.999731
18	115	0.999885
19	264	0.999736
20	936	0.999064
21	92	0.999908
22	9656	0.990344
23	80	0.999920
24	260	0.999740
25	97	0.999903
26	883	0.999117
27	0	1.000000

6 versions with no failures

Remaining versions were successful on more than 99% of the tests

23 of 27 versions were successful on more than 99.9% of the tests

Experimental Results



		UVA Versions								
		1	2	3	4	5	6	7	8	9
UCI Versions	10	0	0	0	0	0	0	0	0	0
	11	0	0	58	0	0	2	1	58	0
	12	0	0	1	0	0	0	71	1	0
	13	0	0	0	0	0	0	0	0	0
	14	0	0	28	0	0	3	71	26	0
	15	0	0	0	0	0	0	0	0	0
	16	0	0	0	0	0	1	0	0	0
	17	2	0	95	0	0	0	1	29	0
	18	0	0	2	0	0	1	0	0	0
	19	0	0	1	0	0	0	0	1	0
	20	0	0	325	0	0	3	2	323	0
	21	0	0	0	0	0	0	0	0	0
	22	0	0	52	0	0	15	0	36	2
	23	0	0	72	0	0	0	0	71	0
	24	0	0	0	0	0	0	0	0	0
	25	0	0	94	0	0	0	1	94	0
	26	0	0	115	0	0	5	0	110	0
	27	0	0	0	0	0	0	0	0	0

Model of Independence

- Separate versions of a program may fail on the same input even if they fail independently. Indeed, if they did not, their failures would be dependent. We base our probabilistic model for this experiment on the statistical definition of independence:

  Two events, A and B, are independent if the conditional probability of A occurring given that B has occurred is the same as the probability of A occurring, and vice versa. That is pr(A|B) = pr(A) and pr(B|A) = pr(B). Intuitively, A and B are independent if knowledge of the occurrence of A in no way influences the occurrence of B, and vice versa.

- The problem with this model is that it is derived from the assumption of independent failures, so this assumption is rejected.

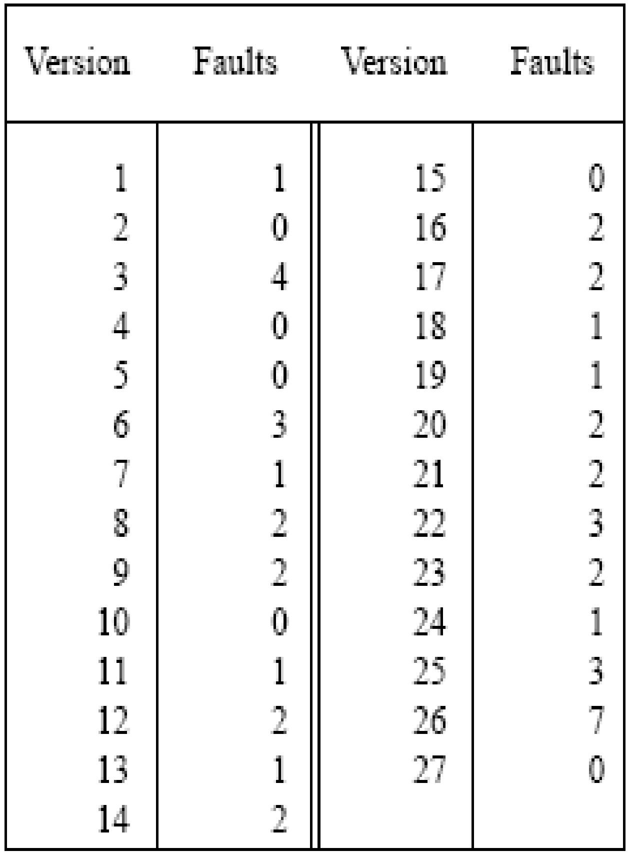
Analysis of Faults

•A *fault* is an instance of program text in any particular version that causes that version to fail when that program text is executed on some test case.

•If a programmer made the same mistake in implementing two different but similar launch conditions, they were recorded as two different faults.

•45 faults were detected in the program versions. (non-correlated/ correlated)

•The faults described in this section all survived that acceptance procedure.



Version	Faults	Version	Faults
1	1	15	0
2	0	16	2
3	4	17	2
4	0	18	1
5	0	19	1
6	3	20	2
7	1	21	2
8	2	22	3
9	2	23	2
10	0	24	1
11	1	25	3
12	2	26	7
13	1	27	0
14	2		

Analysis of Faults

- Non-correlated faults:
  - Omission of assignment of a value to a function➔ use garbage values from memory. (fail 607 times)
  - Use of wrong expression to index an array ➔A specific example is the following function call:

```
sam3pts(x[i], y[i], x[j], y[j], x[k], y[k]);
```

Analysis of Faults

- Correlated Faults:
  - Comparison of angles. (*4 versions fault*).
  - Omission of assumption about the angle subtended by three points.

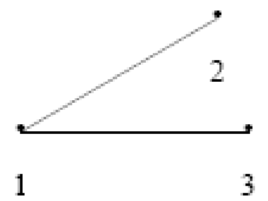


Figure 1A

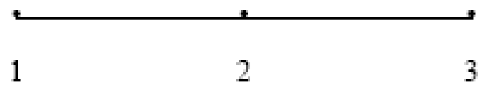


Figure 1B

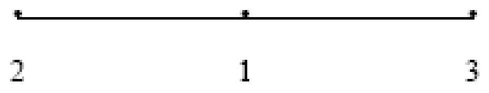


Figure 1C

Discussion

- An experiment of this size would be extremely expensive to undertake if professional programmers were used as the experimental subjects.
- It is strongly considered to pick students with different backgrounds./
- The twenty seven versions ranged in length from 327 to 1004 lines of code. This is much smaller than most real-time systems which may include millions of lines of code.
- The experiment is realistic.

Conclusion

- For the particular problem that was programmed for this experiment, it was concluded that the assumption of independence of errors that is fundamental to the analysis of N-version programming *does not hold*.

- Using a probabilistic model based on independence, the results indicate that the model has to be rejected at the 99% confidence level. It is important to notice that:
  - First, it is conditional on the application used.
  - Second, reliability of an N-version system may not be as high as theory predicts under the assumption of independence.
  - Third, common faults of programmers were surprisingly high, which complicates the experimental analysis.

Thank you