



A Design of fault-tolerant mechanism based on DFA

Lei Tang

SHANGHAI UNIVERSITY OF ELECTRIC POWER

Volt Avengers, Software Engineering

2025.1.11 ECNU



CONTENT

- 1. Reasons for enhancing robustness
- 2. Common software faults with their classification
- 3. Experiment designing
- 4. Fault-tolerant DFA
- 5. Comparison with the former experiment
- 6. Result



Why should we enhance robustness ?

Our bodies have great availability. I have soft errors all the time: my memory fails once in a while, but I don't 'crash.' My whole body doesn't shut down when I cut a finger .

by Robert Morris





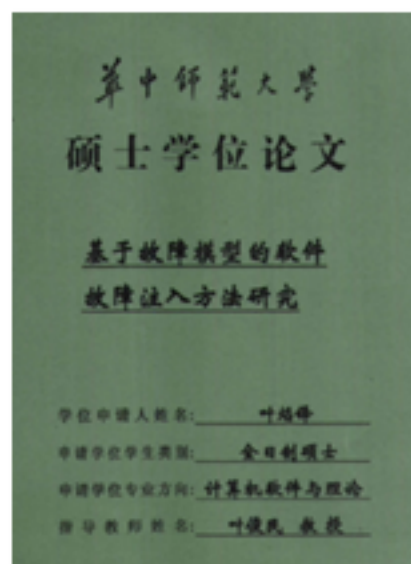
The fact is ...

1962 Mariner rocket fatal BUG:

A U.S. Navy programmer transcribed a handwritten formula into incorrect computer code, causing a 1962 Mariner 1 rocket carrying a space probe to veer off course shortly after takeoff and be destroyed by mission control 293 seconds after takeoff, causing *\$18.5 million* in economic damage.



How software disabled?



软件失效的机理可描述为^[20]：软件差错→软件缺陷→软件故障→软件失效。

- **软件错误 (Software error)** 是指软件生存期内的不希望或不可接受的人为错误。
- **软件缺陷 (Software defect)** 指存在于软件 (文档、数据、程序) 之间的一些不希望或者不能够接受的偏差, 比如, 程序中多了一个逗号, 或者多了一条无用的语句等。
- **软件故障 (Software fault)** 是指软件运行过程中出现的一种不希望或者不可接受的内部状态。
- **软件失效 (Software failure)** 是指软件运行中出现的一种不希望或不能接受的外部行为结果, 即软件故障发生以后导致的一系列后果。



Software Error



Software Defect



Software Fault



Software Disabled



Common software faults with their classification

2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)

On Error Representativeness of Function Call Interfaces for C/C++ Program

Wei Zhang, Hao Xu, Zhongjun Lu, and Jianshi Jiang*

School of Software Engineering, Tongji University, Shanghai, China

1110034@tongji.edu.cn, 21114034@tongji.edu.cn, 2111402@tongji.edu.cn, jiang@tongji.edu.cn

*corresponding author

Abstract—Software fault injection is a fundamental technique for analyzing the dependability of software systems. The code mutation based defect injections encounter efficiency (i.e., domain faults) and accuracy (when they are not done at the source code level) issues. Software error injections directly emulate the effects of software defects, which is more efficient and practical. But, whether the injected errors truly represent the impact of bugs in the real world is still an open problem. Several studies have investigated the representativeness problem of component interface (API) errors. However, the component API error injections are relatively coarse in granularity because they regard the component programs as black boxes. Thus, they cannot be used to analyze the internal error behaviors within the components. To overcome this issue, we propose a trace state machine for generating representative code mutations (in baseline) and a novel tracing approach for collecting function call interface data. This enables the analysis of the representativeness of program internal errors from the perspective of function call interface for C/C++ programs by fault-free and fault injection control experiments. The experimental results imply that the traditional error models cannot accurately emulate software defects at function call interfaces. We provide useful suggestions based on our findings for improving the representativeness of function call interface error injections.

Keywords—dependability; software fault injection; error modeling; representativeness; function call interface

1. INTRODUCTION

With the increasing complexity of software, software defects have become one of the primary threats to software systems [1]. Software fault injection serves as a fundamental technique for validating the dependability of software systems by deliberately introducing faults [1]. It can be employed to analyze whether the software meets the specified reliability level, assess the effectiveness of fault tolerance mechanisms, and find diagnosis failures.

Software fault injection techniques have evolved into two major categories: *code mutation based software defect injection* and *defect manifestation (error) injection* [2]. The former simulates software defects by mutating program source, intermediate, or binary codes. The latter directly emulates software errors caused by software defects, including full-stop behavior (such as program crashes and hangs) and program interface data errors (such as incorrect function parameters or return data).

The drawback of code mutation is its inefficiency [3], making it less practical for modern software with larger scales and complexities. However, it offers good representativeness [2] for simulating real-world defects observed in production environments [3]. This is due to its direct emulation of software defects, with the classification and distribution of these defects continuously improving through decades of academic and industrial practice and summarization [3]. In comparison, software error injection is more efficient since it avoids a large number of domain fault experiments by directly emulating the impacts of defects. The current limitations of software error injection are mainly due to the lack of comprehensive representativeness research for this type of fault injection.

Error representativeness is commonly regarded as the similarity between injected errors and errors caused by real-world defects in terms of error patterns (what to inject), spatial distribution (where to inject), and temporal distribution (when to inject) [2]. Despite its widespread application in the dependability area [3], the validity of the simulation errors in representing realistic defect impacts is often questioned [2]. Roberts et al. [2] studied the representativeness issues regarding component interface (API) errors and found that 30%–50% software defects can be traced by errors in the component interface, helping in studying the error behaviors of systems constructed by components. However, component external interface error injection has limited capability in researching the error behaviors of component programs themselves due to its coarse granularity. For fine-grained error injections (e.g., function call interface error injections) are often used to help more comprehensively error behavior studies [1]. The function call interface error refers to the error occurring in the interface of a function (e.g., parameters, return data, global variables, files, streams, and class member variables the function uses).

Based on our knowledge, no research has comprehensively analyzed the internal program errors' representativeness at the function call interface level. However, these error injections have already been used in various domains for analyzing program error behaviors, program reliability, and the effectiveness of error handling mechanisms [1], [4]. Therefore, we intend to research the representativeness issue for internal program errors from the perspective of the function call interface.

Our study analyzes the impact of injected representative defects on the program from the perspective of the program function call interfaces and whether the existing error models [1] used so far are representative; if not, how to improve it. It should be noted that in this paper we only address the Boolean-type, consistent, valid bugs, not Modeling-type

code mutation based software defect injection and defect manifestation (error) injection

```
int main()
{
    printf(
        "hello,
        world!
        \n");
}
```

Code mutation

```
f()
{
    return 0;
}

int a = 1/f();
```

Error injection



Method we used to simulate mutation

CODE MUTATION METHOD: ODC (*Orthogonal Defect Classification*)

Orthogonal Defect Classification—A Concept for In-Process Measurements

Ran Chikarege, Senior Member, IEEE, Indrajit S. Bhattacharjee, Member, IEEE, Jurek K. Chant, Member, IEEE, Michael J. Halliday, Diane S. Moehs, Bonnie K. Ray, and Man-Yuen Wong

Abstract—This paper describes orthogonal defect classification (ODC), a concept that enables in-process feedback to developers by extracting signatures on the development process from defects. The ideas are evolved from an earlier finding that demonstrates the use of semantic information from defects to extract cause-effect relationships in the development process. This finding is leveraged to develop a systematic framework for building measurement and analysis methods. This paper

- defines ODC and discusses the necessary and sufficient conditions required to provide feedback to a developer;
- illustrates the use of the defect type distribution to measure the progress of a product through a process;
- illustrates the use of the defect trigger distribution to estimate the effectiveness and eventually the completeness of verification processes such as inspection or testing;
- provides sample results from pilot projects using ODC;
- opens the door to a wide variety of analysis techniques for providing effective and fast feedback based on the concepts of ODC.

1. INTRODUCTION

IN RECENT years the emphasis on software quality has increased due to factors from several sectors of the computer industry. Software is one of the slowest processes in enabling new business opportunities, solutions, or dimensions of computing, and is a significant part of total cost. It has also been found that the dominant cause of system outages has shifted to software given that it has not kept pace, in the past few years, with improvements in hardware or maintenance [1]. Thus there is a resurgence of research in software topics such as reliability, engineering, measurement, etc. [2], [3]. However, the area of software quality measurements and quantification is beset with undue complexity and has, in some ways, advanced away from the developer. It is an area where the processes are so amorphous, the tangibles required for measurement and modeling are few. With the result academic pursuits that can't be confined to the limitations of practice evolved and become detached from the developer. In this area, the need to derive reliable measurements that are amenable to understand and intuitively plausible cannot be understated. Measurement without an underlying theory can leave the experimentalist, the theorist and the practitioner very confused.

Manuscript received October 1, 1991; revised August 1, 1992. Recommended by R. Kelly and R. Shull.

R. Chikarege, J.S. Bhattacharjee, J.K. Chant, M.J. Halliday, B.K. Ray, and M.Y. Wong are with IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

D.S. Moehs is with IBM Westborough Valley Programming Lab, Westborough, MA.

IEEE Log Number 9207044.

The goal of this paper is to develop reasonable measurements. It does this by examining the fundamental properties of such measurements and then deriving the rationale for analysis and models. To put the domain of software in-process measurements and analysis in perspective, let us examine two extremes of the spectrum: statistical defect models and qualitative causal analysis.

1.1. Statistical Defect Models

The goal of statistical defect modeling, which includes what is commonly referred to as software reliability growth, has been to predict the reliability of a software product. Typically, this may be measured in terms of the number of defects occurring in the field, the failure rate of the product, the short-term defect detection rate, etc. [4]–[6]. Although this may provide a good report card, it often comes so late in the development cycle that it is of little value to the developer. Ideally, a developer would like to get feedback during the process.

1.2. Causal Analysis

The goal of causal analysis is to identify the root cause of defects and initiate actions so that the source of defects is eliminated. To do so, defects are analyzed, one at a time, by a team that is knowledgeable in the area. The analysis is qualitative and only limited by the range of human investigative capabilities. To sustain such pursuit and provide a focus for the activity a process description has been found useful. At IBM, the Defect Prevention Process [6] and similar efforts elsewhere have found causal analysis to be very effective in reducing the number of errors committed in a software project. The qualitative analysis provides feedback to developers that eventually improves both the quality and the productivity of the software organization [7].

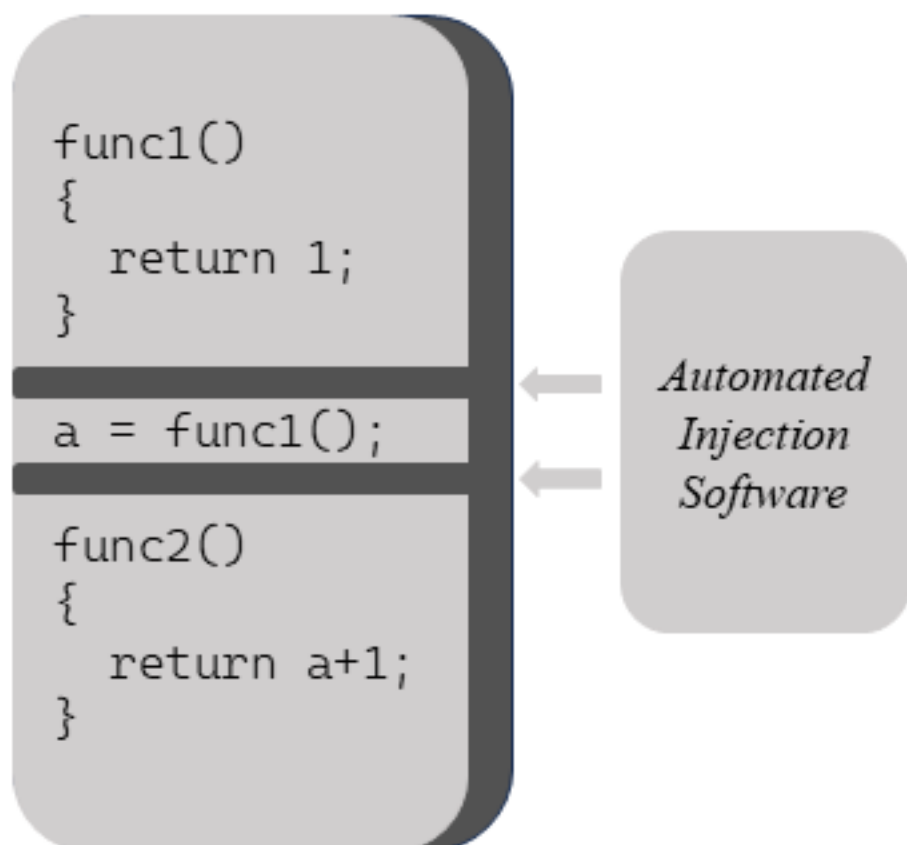
Defect Prevention can provide feedback to developers at any stage of their software development process. However, the resources required to administer this method are significant, although the rewards have proven to be well worth it. Moreover, given the qualitative nature of the analysis, the method does not lend itself well to measurement and quantitative analysis. Consequently, Defect Prevention, though not a part of the engineering process control model, could eventually work in conjunction with it.



Operator	Description
OMFC	delete the function call statement
OMVIV	delete the initialization part
OMVAV	delete the constant value assignment statement
OMVAE	delete the expression assignment statement
OMIA	delete the if construct, and left the inside statements
OMIFS	delete the if construct and the inside statements
OMIEB	delete the if construct, the inside statements, the else construct, and left the statements in else
OMLC	delete the && or , and one of the operator
OMLPA	delete the continuous statements
OWVAV	modify the value assigned to a variable by xor with 0xFF (same as [3])
OWPFV	modify the variable in the parameter of a function call (different with [3]), by xor with 0xFF
OWAEP	modify the arithmetic expression in a function parameter by deleting one of the operators (generate faulty codes for each operator)



Method we used to inject error injection

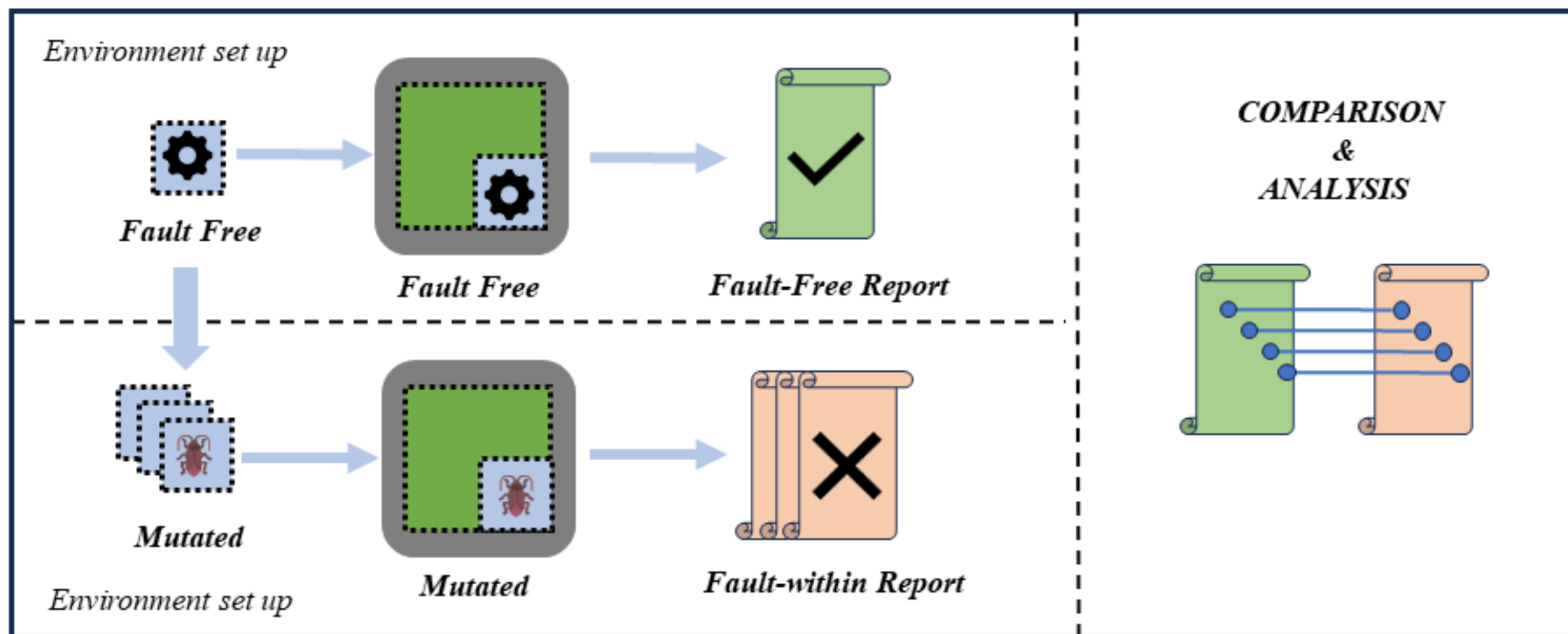


Traditionally, error injection is generally performed by dynamically injecting near the entry or exit of a function block.

However, since we have not developed automated injection software, we achieve a similar effect by manually modifying the code near the entry and exit.



Experiment Designing





ODC & ERROR INJECTION TEST REPORT

上海電力大學

SHANGHAI UNIVERSITY OF ELECTRIC POWER



FAULT TOLERANCE

題目: Volt OS 故障注入实验报告书
(ODC Method)

Under the on-campus supervision of

Dr. Man Xu, Dr. Zhen Wang

Under the off-campus supervision of

Dr. Wei Zhang

Tested by Lei Tang, Software Engineering

上海電力大學

SHANGHAI UNIVERSITY OF ELECTRIC POWER



FAULT TOLERANCE

題目: Volt OS 错误注入实验报告书
(ERROR INJECTION Method)

Under the on-campus supervision of

Dr. Man Xu, Dr. Zhen Wang

Under the off-campus supervision of

Dr. Wei Zhang

Tested by Lei Tang, Software Engineering



FAULT-TOLERANT DFA

A Deterministic Finite Automata is defined as :

$$M = \langle \Sigma, Q, \delta, Q^0, F \rangle$$

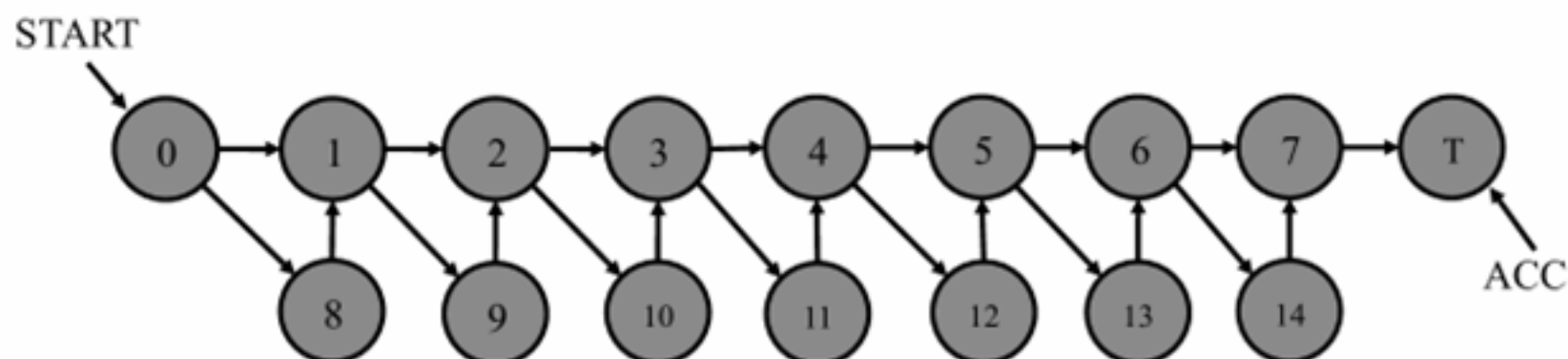
1. Σ - input alphabet
2. Q - set of states
3. δ - state-transition function
4. Q^0 - initial state
5. F - final states



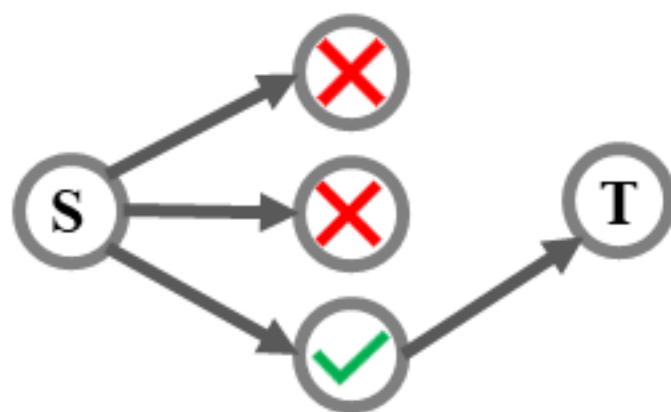
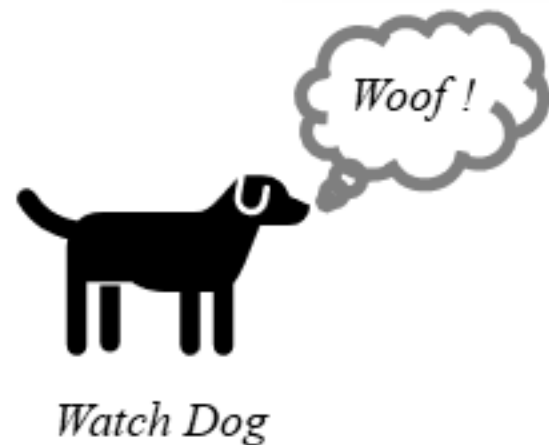
$$\delta (q^a , \Sigma^a) = q^b$$



Construction of *FAULT-TOLERANT DFA*



$$\delta(q^a, \Sigma^a) = q^b$$





Construction of Watch Dog

We can implement a *decision maker* using a triple modular redundancy system. The state is judged to be correct *if and only if the outputs of all three decision makers are correct.*



Decision maker 1
Output : 1



Decision maker 2
Output: 1



Decision maker 3
Output: 1



Watch Dog
GOOD DOG!



Construction of Watch Dog

We can implement a *decision maker* using a triple modular redundancy system. The state is judged to be correct *if and only if the outputs of all three decision makers are correct.*



Decision maker 1
Output : 1



Decision maker 2
Output: 0



Decision maker 3
Output: 1

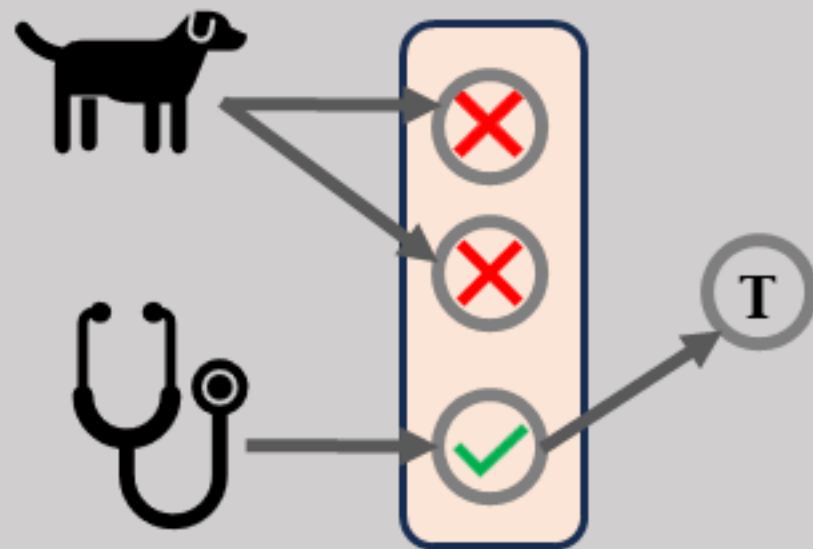


Woof!

Watch Dog
BARKING DOG!



DOCTOR MODULE



When the doctor program detects the barking of a dog, the backup program will be activated at this time, and the problematic module will be dynamically replaced while the program is running.



Comparison with the former experiment

The former version

```
1 Hello, VoltOS!  
2 developed by Lei Tang, SUEP.  
3 Mutated Mode:  
4 -----welcome to Volt OS-----  
5 page allocator initializing...  
6 HEAP_START = 0x80008338(aligned to 0x80009000),  
7 HEAP_SIZE = 0x07ff7cc8,  
8 num of reserved pages = 8,  
9 num of pages to be allocated for heap = 32751  
10 TEXT: 0x80000000 -> 0x80004238  
11 RODATA: 0x80004238 -> 0x800047ff  
12 DATA: 0x80005000 -> 0x8000502c  
13 BSS: 0x80005030 -> 0x80008338  
14 HEAP: 0x80011000 -> 0x80000000  
15 page initialized.  
16 [VoltCommand]$ go  
17 Sync exceptions! Code = 7  
18 panic: DOPE! What can I do!  
19  
20  
21  
22 系统崩溃了。
```

```
1 Hello, VoltOS!  
2 developed by Lei Tang, SUEP.  
3 Mutated Mode:  
4 -----welcome to Volt OS-----  
5 page allocator initializing...  
6 HEAP_START = 0x80008338(aligned to 0x80009000),  
7 HEAP_SIZE = 0x07ff7cc8,  
8 num of reserved pages = 8,  
9 num of pages to be allocated for heap = 32751  
10 TEXT: 0x80000000 -> 0x800041d0  
11 RODATA: 0x800041d0 -> 0x80004797  
12 DATA: 0x80005000 -> 0x8000502c  
13 BSS: 0x80005030 -> 0x80008338  
14 HEAP: 0x80011000 -> 0x80000000  
15 page initialized.  
16 [VoltCommand]$ gopanic: Num of task should be  
17 greater than zero!  
18  
19 QEMU: Terminated  
20  
21 程序崩溃了。
```

CRASH!

The robust version

```
status of uart initialization is 1  
status of uart initialization is 1  
status of uart initialization is 1  
final decision of uart initialization is 1  
woof! UART initialized success!  
0, 1, -1, -1, -1, -1, -1, -1,  
page allocator initializing...  
HEAP_START = 0x0000718(aligned to 0x00000000), HEAP_SIZE = 0x07ff7cc8,  
num of reserved pages = 8, num of pages to be allocated for heap = 32747  
TEXT: 0x00000000 -> 0x00007228  
RODATA: 0x00007228 -> 0x00008134  
DATA: 0x00009000 -> 0x00009000  
BSS: 0x00009000 -> 0x0000c718  
HEAP: 0x00015000 -> 0x00000000  
page initialized.  
page allocator initializing...  
HEAP_START = 0x0000c718(aligned to 0x00000000), HEAP_SIZE = 0x07ff7cc8,  
num of reserved pages = 8, num of pages to be allocated for heap = 32747  
TEXT: 0x00000000 -> 0x00007228  
RODATA: 0x00007228 -> 0x00008134  
DATA: 0x00009000 -> 0x00009000  
BSS: 0x00009000 -> 0x0000c718  
status of scheduler initialization01 is :1  
status of scheduler initialization02 is :1  
0, 1, 2, 3, 4, 5, 6, -1,  
the status of uart_and_command is 0  
the status of uart_and_command01 is 1  
the status of uart_and_command02 is 1  
woof! COMMAND initialized failed!  
0, 1, 2, 3, 4, 5, 6, 14,  
woof! RESTARTING...  
uart display and interruption handler initialized!  
0, 1, 2, 3, 4, 5, 6, 7, ACCEPTED  
Hello, VoltOS!  
developed by Lei Tang, SUEP.  
ROBUST Mode:  
-----welcome to Volt OS-----  
[VoltCommand]$
```

ACCEPTED!



Result

1. Through the construction of the Deterministic Finite Automaton (DFA), we have solved the problem of failures after loading caused by the incorrect initialization of the Kernel.
2. Through **backup, redundancy**, as well as the detection and state pushing by the *watchdog program*, along with the diagnosis and state rollback by the *doctor program*, the design of the fault-tolerant computing mechanism has finally been achieved.

(you can check the comparison analysis video)



Thanks to Dr. Man Xu and Dr. Zhen Wang, SUEP, for their help and guidance on the direction of fault-tolerant computing in this experiment. 🌹 🌹

Thanks to Dr. Wei Zhang, an off-campus technical expert, for his great assistance as well as his provision and explanation of the thesis. 🌹 🌹

THANK YOU!