KIPS Trans. Softw. and Data Eng.
Vol.8, No.7 pp.265~274 pISSN: 2287-5905

https://doi.org/10.3745/KTSDE.2019.8.7.265

Analytical Study on Software Static/Dynamic Verification Methods for Deriving Enhancement of the Software Reliability Test of Weapon System

Jihyun Park† · Byoungju Choi††

ABSTRACT

The reliability test performed when developing the weapon system software is classified into static test and dynamic test. In static test, checking the coding rules, vulnerabilities and source code metric are performed without executing the software. In dynamic test, its functions are verified by executing the actual software based on requirements and the code coverage is measured. The purpose of this static/dynamic test is to find out defects that exist in the software. However, there still exist defects that can't be detected only by the current reliability test on the weapon system software. In this paper, whether defects that may occur in the software can be detected by static test and dynamic test of the current reliability test on the weapon system is analyzed through experiments. As a result, we provide guidance on improving the reliability test of weapon system software, especially the dynamic test.

Keywords : Dynamic Software Defects, Weapon System Software, Dynamic Test

무기체계 소프트웨어 신뢰성 시험 개선점 도출을 위한 소프트웨어 정적/동적 검증 분석 사례연구

박 지 현† · 최 병 주††

요 약

무기체계 소프트웨어 개발 시 수행하는 신뢰성 시험은 크게 정적 검증과 동적 검증으로 구분된다. 정적 검증에서는 소프트웨어 코드를 수행시키지 않고 코딩 규칙 점검, 취약점 점검, 소스 코드 메트릭 점검을 수행하고, 동적 검증에서는 요구 사항을 기반으로 실제 소프트웨어를 실행시켜 기능을 검증하고 코드 실행률을 측정한다. 이러한 정적/동적 검증의 목적은 소프트웨어에 존재하는 결함을 발견하기 위한 것이다. 그러나 현재의 무기체계 소프트웨어 신뢰성 시험만으로는 여전히 탐지할 수 없는 결함들이 존재한다. 본 논문에서는 소프트웨어에서 발생할 수 있는 결함에 대해 무기체계 신뢰성 시험의 정적 검증과 동적 검증으로 탐지를 할 수 있는지를 사례실험을 통하여 분석 한다. 그 결과로 현재의 정적 검증과 동적 코드 커버리지 측정에서 더 나아가 무기체계 신뢰성 시험, 특히 동적 시험의 개선방안으로 연결하고자 한다.

키워드 : 소프트웨어 동적 결함, 무기체계 소프트웨어, 동적 시험

1. 서 론

무기체계에서 소프트웨어가 차지하는 비중이 높아지고 그 구조도 점차 복잡해지고 있어 소프트웨어 결함의 발생 빈도 역시 높아지고 있다[1]. 무기체계 소프트웨어란 무기체계에 탑재되어 무기체계를 제어하고 동작시키는 소프트웨어이다[2]. 무기체계 소프트웨어는 정해진 시간 안에 임무를 완수하여야 하고 어떠한 상황에서도 중단 없이 동작하여야 하므로 고 신뢰성이 요구되는 소프트웨어이다[2]. 무기체계 소프트웨어의 결함 발생을 줄이고 신뢰성을 확보하기 위해 현재 무기체계 소프트웨어 개발 프로세스[3]에서는 소프트웨어 신뢰성 시험을 통한 소프트웨어 검증을 반드시 수행하도록 되어있다.

신뢰성 시험에서는 자동화 도구를 이용하여 정적 시험과 동적 시험을 수행하여 소프트웨어 결함을 사전에 탐지하고 해결하도록 하고 있다. 정적 시험에서는 정적 분석 도구를 이용하여 코딩 규칙 검증, 취약점 점검 및 소스 코드 메트릭 점검을 수행하고, 동적 시험에서는 요구 사항을 기반으로 실제 하드웨어(Target)에 탑재하여 코드의 문장(statement), 분기(branch) 실행률, MC/DC (Modified Condition/Decision) 등의 실행률(coverage)을 점검한다.

※ 본 연구는 방위사업청과 방위산업기술지원센터의 지원(계약번호: UC160002D) 하에 수행되었습니다.
† 정 회 원 : 이화여자대학교 컴퓨터공학과 박사후과정
†† 정 회 원 : 이화여자대학교 컴퓨터공학과 교수
Manuscript Received : February 1, 2019
First Revision : April 29, 2019
Accepted : May 31, 2019
* Corresponding Author : Byoungju Choi(bjchoi@ewha.ac.kr)

연구[4]에 따르면 무기체계 소프트웨어가 동작하는 과정에서 발생할 수 있는 소프트웨어결함으로써 미 국방부에서 무기체계에 사용하는 오픈 소스 소프트웨어(OSS) 커뮤니티에서 등록된 이슈 분석을 통하여 메모리, 병행성, 예외처리 결함 종류를 도출하였다. 메모리 결함은 프로세스 내부의 메모리 할당, 해제 및 접근 과정에서 발생하는 결함이다. 병행성 결함은 멀티스레드 혹은 멀티프로세스 환경에서 스레드나 프로세스 간 자원 공유 및 동기화 과정에서 발생하는 결함이다. 예외처리 결함은 예외가 발생하였으나 예외처리 루틴이 정상적으로 동작하지 않는 결함이다. 이 세가지 결함은 일단 발생하게 되면 시스템 다운과 같은 치명적인 영향을 미치므로 반드시 해결되어야 하는 결함이다.

정적 검증 도구는 CodeSonar, Polyspace, QA C/C++ 등이 대표적인데, 코드를 실행하지 않고도 런타임에 발생할 수 있는 결함을 미리 탐지하기도 한다. 그러나 이 때 탐지할 수 있는 결함은 코드를 실제 실행할 때 발생하는 일명, '동적' 결함의 일부이므로 반드시 동적 검증을 수행하여야 한다. 특히 소프트웨어를 단위 컴포넌트부터 통합해가며 각 통합 단계마다 검증 과정을 거치지 않고, 모든 소프트웨어 컴포넌트를 묶어 단번에 실행하는 무기체계 소프트웨어 신뢰성 시험 방식인 일명 '빅뱅 통합 방식' 검증[3]을 수행하는 경우에는 결함이 발생하게 되면 단위 소프트웨어, 구성품 및 소프트웨어 형상 항목이 통합되는 과정 중 어디에서 결함이 발생하였는지 그 원인을 파악하기 매우 어렵다. 단순히 동적 코드 커버리지 측정만 하는 것에서 더 나아가 [4]에서 정의한 무기체계 소프트웨어 결함을 사전에 탐지할 수 있도록 하는 동적 검증은 매우 중요하다.

본 논문의 주요 기여 점은 현재의 정적 및 동적 도구로는 탐지가 되지 않는 동적 무기체계 소프트웨어결함을 사례연구 및 실험를 통하여 분석한다. 그 결과로 현재의 정적 검증과 동적 코드 커버리지 측정에서 더 나아가 동적 결함 탐지하는 과정이 필요함을 보임으로써 무기체계 신뢰성 시험, 특히 동적 시험의 개선방안으로 연결하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 정적 및 동적 검증에 대해 다룬다. 3장에서는 정적 검증 방안과 동적 결함 탐지 도구 및 런타임 오류 정적 검증 도구의 결함 탐지를 실험 분석하고, 4장에서는 결론 및 향후 연구 방향을 기술한다.

2. 관련 연구

2.1 정적 검증

정적 검증이란 소프트웨어를 실행하지 않은 상태에서 잠재적인 결함을 검출하는 것을 말한다[4]. 무기체계 소프트웨어 신뢰성 시험의 정적 검증에서 분석하는 내용은 주로 소프트웨어가 자동차 산업 소프트웨어 안전협회(MISRA) 코딩 규칙 혹은 방위사업청 무기체계 SW 코딩 규칙과 같은 소스 코드 작성 규칙에 맞춰 작성이 되었는지 여부, 소스 코드가 보안 취약 일반 유형(CWE) 목록에 정의된 취약점을 포함하고 있는지의 여부 소스 코드 메트릭 점검이다.

정적 검증을 수행하기 위해서는 자동화 도구를 사용한다. 이 때 사용되는 자동화 도구는 코딩규칙 검증, 취약점 점검, 소스 코드 메트릭 측정과 같이 목적에 따라 달라질 수 있다. 코딩 규칙 점검을 위해서는 QAC, Code Inspector, Polyspace 등과 같은 도구가 사용되고, 취약점 점검에는 CodeSonar, Sparrow, Polyspace 등이 사용된다. 소스 코드 메트릭 측정에는 QA Framework(C/C++), Code Inspector, Polyspace 등이 사용된다.

```
typedef void* (*mallocFP)(size_t size);
typedef void (*freeFP)(void*);

void __attribute__((constructor)) init(void) {
        …
        Fp_init(malloc_address, feee_address);
        …
}

void fp_init(mallocFP allocFP, freeFP deallocFP) {
          mallocFP = allocFP;
          freeFP = deallocFP;
}
```

Fig. 1. Example of MISRA Rule Avoidance

그러나 정적 검증만으로는 모든 결함을 탐지하는데 한계가 있다. 예를 들어 MISRA 코딩 규칙에 따르면 동적 메모리 결함을 발생시키기 않도록 하기 위해서 동적으로 할당된 메모리를 사용하지 않도록 한다. 그러나 이 규칙을 회피하는 코드가 있을 수 있고, MISRA 코딩 규칙을 점검하는 정적 도구로 검사가 되지 않는 표준 라이브러리 함수에 동적 메모리를 사용하는 코드가 있을 수 있다. 또한 Fig. 1과 같이 프로그램에서 사용하는 라이브러리 함수에서 메모리 할당/해제 함수에 대한 포인터를 생성하고, 라이브러리 초기화 시 각 함수 포인터에 주소를 맵핑하는 경우에는 도구에서는 코딩 규칙 위반으로 탐지되지 않으므로 mallocFP, freeFP의 잘못된 사용으로 인한 메모리 결함 또한 탐지하기가 어렵다.

한편, 정적 검증을 통해 실제 코드를 실행하기 이전인 컴파일 타임에 런타임에 발생 할 수 있는 오류를 분석하여, 즉 '런타임 오류 정적 분석' 방법으로 메모리 결함이나 멀티 스레드에서의 data race를 탐지하기도 한다. 런타임 오류 정적 분석 도구로는 Polyspace, CodeSonar, LDRA Testbed, Coverity, Chord 등이 대표적이다. 그러나 실제 코드를 실행하지 않기 때문에 오탐율이 높고, 탐지가 되더라도 일부 결함에 한정된다.

2.2 동적 검증

동적 검증은 실 실행환경이나 그와 유사한 실행 환경에서 소프트웨어 코드를 실행시켜 테스트하고 평가하는 것이다. [4]에서 본 연구진은 무기체계 소프트웨어 결함을 Table 1과 같이 메모리, 병행성, 예외처리 결함을 정의하고, 미 국방부에서 무기체계에 사용하는 오픈 소스 소프트웨어(OSS) 커뮤니티에서 등록된 이슈를 통하여 이들 결함이 실제 무기체계 소프트웨어에서 결함으로 보고되고 있음을 확인하였다. 특히

Table 1. Dynamic Software Defects

Category	Defect			ID
Memory	Memory allocation		Memory allocation failure	M01
			Zero-size memory allocation	M02
			Memory leak	M03
	Memory deallocation		NULL pointer free	M04
			Duplicated free	M05
			Unallocated pointer free	M06
	Memory access		NULL pointer access	M07
			Released pointer access	M08
			Unallocated pointer access	M09
			Out of range memory access	M10
			Conflict with other variables	M11
Concurrency	Deadlock	Resource deadlock	Synchronized object deadlock	C01
		Message deadlock	Fail to receive synchronized message	C02
			Fail to process synchronized message	C03
			Fail to send synchronized message	C04
			Loss of synchronized message transmission	C05
			Fail to transmission asynchronous message	C06
			Fail to process asynchronous message	C07
	Atomicity violation	Variable	Variable use atomicity violation	C08
		Memory read/write	Perform a write operation on other threads while performing a read operation on shared memory	C09
			Perform write operation on other threads while performing read after write operation on shared memory	C10
			Perform write operation on other threads while performing read after write operation on shared memory	C11
			Perform read operation on other threads while performing a write operation on shared memory	C12
			Perform write operation on other threads while performing a write operation on shared memory	C13
	Order violation	Uncreate resource	Using uncreated objects	C14
			Using uncreated shared memory	C15
		Uninitialized resource	Using uninitialized objects	C16
			Using uninitialized shared memory	C17
		Released resource	Using released objects	C18
			Using released shared memory	C19
		Shared memory access	Read and write operation on shared memory from different threads	C20
			write operations on shared memory from different threads	C21
ExceptIon handlIng	Device driver exception handling	Device connection	Device connection failure	E01
			Device open failure	E02
		Device disconnection	Device disconnection failure	E03
			Device close failure	E04
		No Data	Fail to read data from device	E05
			Fail to write data to device	E06
			Fail to seek data on the device	E07
			Fail to transfer IOCTL data to/from device	E08
		Illegal Data	Reading illegal data from the device	E09
			Writing illegal data to the device	E10
			Seeking illegal data on the device	E11
			Transferring illegal IOCTL data to/from device	E12
		Power	Fail to power on device	E13
			Fail to power down device	E14
	OS structured exception handling		Interrupt handler exception handling failure	E15
	Application structured exception handling		Try-catch exception handling failure	E16

이들은 다양한 외부 센서나 UI, DB와 같은 다른 구성 요소 CSC 및 CSCI들과의 상호 작용으로 통합 이후 발생하는 결함이며, 이 가운데에는 재현이 불가하여 결함 원인 파악에 수일에서 수개월까지도 걸리는 결함이 존재하였다.

무기체계 소프트웨어 신뢰성시험의 동적 검증에서는 요구 사항을 기반으로 실제 하드웨어(Target)에 탑재하여 코드의 문장(statement), 분기(branch) 실행률, MC/DC (Modified Condition/Decision) 등의 실행률(coverage)을 점검한다. 코드 커버리지 측정 도구로는 LDRA toolset, Cantata++, VectorCast, CodeSonar 등이 있다. 그러나 커버리지를 100% 만족한다 하더라도 소프트웨어 결함은 계속 존재한다. 소프트웨어 결함을 발견하는 데는 커버리지 측정률보다는 커버리지를 만족시키는데 사용한 테스

트데이터에 영향을 받기 때문이다.

일반적으로 동적 검증과정에서 결함이 발생하였을 경우, 결함을 탐지하기 위한 디버깅 도구를 사용한다. 결함 탐지를 위한 도구로는 Valgrind, Dr.Memory, ThreadChecker 등이 대표적이다.

3. 소프트웨어 결함 탐지 실험 분석

현재 무기체계 소프트웨어의 신뢰성 시험에서 사용하는 정적 검증 및 동적 검증 방안이 Table 1의 소프트웨어 결함 탐지에 얼마나 효과적인가를 분석한다. 그리고 일반적으로 결함 탐지에 사용하는 동적 결함 탐지 도구와 런타임 오류 정적 검증 도구에 대해서도 실험을 통해 결함 탐지 효과를 분석한다. 그 결과를 기반으로 무기체계 소프트웨어의 동적 시험에 대한 개선 방안을 이끌고자 한다.

3.1 정적 검증을 통한 소프트웨어 결함 탐지 분석

무기체계 소프트웨어 신뢰선 시험의 정적 검증 방안을 적용하였을 때, 소프트웨어 결함이 얼마나 사전에 탐지될 수 있는지를 분석하고자 한다.

1) 실험 설계

Table 1 소프트웨어 결함 중 무기체계 소프트웨어 신뢰성 시험의 코딩 규칙 검증, 취약점 점검 및 소스코드 메트릭 점검을 통해서 얼마나 결함을 탐지할 수 있는지 분석하기위하여 무기체계 소프트웨어의 개발에 사용되는 표준을 검토하여 Table 2와 같이 분석 대상을 선정하였다. 코딩 규칙과 관련된 산업 표준으로는 IEC61508, ISO 26262, DO-178B/C, IEC 62279, IEC 60880, EN 50128 등이 있고, 관련 코딩 규칙으로는 MISRA-C/C++, JSF++(Joing Strike Fighter Air Vehicle C++), Code convertions for the Java Programming Language, C# Coding conversion, .NET Framework Design Guideline, 방위사업청 코딩 규칙 등이 있다. Table 1의 소프트웨어 결함은 C/C++ 어플리케이션에서 발생할 수 있는 결함을 타겟으로 하였으므로[4], 그 중 MISRA-C/C++[5,6]과 방위사업청 코딩 규칙[3]을 대상으로 한다.

취약점 점검은 소프트웨어 소스 코드가 CWE 목록에 정의된 취약점을 포함하고 있는지 점검하는 것으로[3] CWE-658/659/660 등이 있다. C/C++로 작성된 소프트웨어의 경우에는 CWE-658과 CWE-659에 해당하는 항목을 점검하게 된다[7].

소스코드 메트릭은 소프트웨어의 복잡도 감소, 유지보수 용이성 증대 등 소프트웨어의 품질 향상을 위한 소스코드의 품질 측정 지표로[3], 무기체계 소프트웨어 개발 및 관리 매뉴얼[3]에 따르면 함수를 기준으로 Cyclomatic Complexity, Number of Call Levels, Number of Function Parameters, Number of Calling Functions, Number of Called Functions, Number of Executable Code Lines를 측정한다. 그러나 이러한 소스코드 메트릭은 소프트웨어의 크기, 복잡도 등과 같은 구조적 특성이 결함 발생 확률에도 영향을 줄 수 있으므로[8] 그 값을 제한하여 결함 발생 확률을 줄이는 것을 목표로 한다. 따라서 [4]의 소프트웨어 결함을 직접적으로 탐지하지는 않으므로 분석 대상에서는 제외한다.

2) 분석 결과

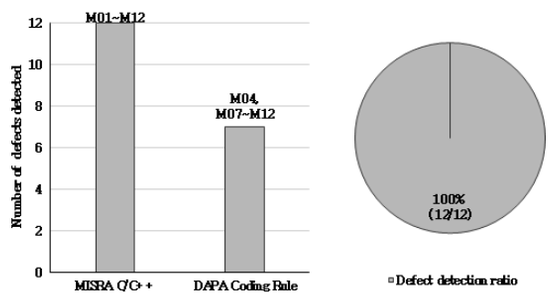
소프트웨어 결함에 대한 정적 검증의 실험 결과를 코딩 규칙 검증과 취약점 점검 각각으로 구분하여 분석한다.

Table 2. Static Analysis Target

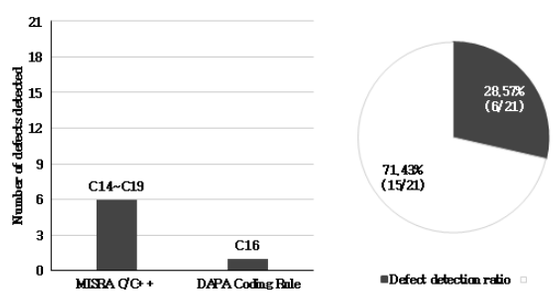
Category	Target
Coding rule	MISRA C/C++
	Defense Acquisition Program Administration(DAPA) coding rules
CWE check	CWE-658
	CWE-659

a) 코딩 규칙 검증

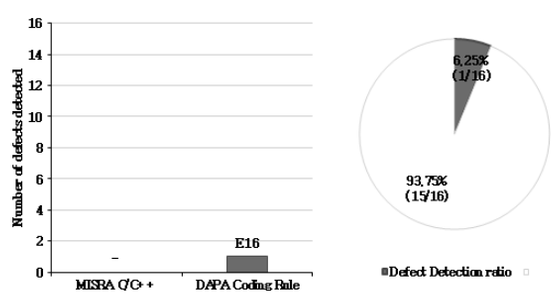
코딩 규칙이란 소프트웨어 구현에 적용하는 소스 코드 작성 규칙으로 MISRA C/C++와 방위사업청 코딩 규칙의 소프트웨어 결함 탐지 분석 결과는 Fig. 2와 같이 나타났다.



(a) Memory Defects



(b) Concurrency Defects



(c) Exception Defects

Fig. 2. Defect Detection Ratio of Coding Rule

메모리 결함 12개, 병행성 결함 21개, 예외처리 결함 16개에 대하여, MISRA C/C++은 메모리 결함 12개 모두, 그러나 병행성 결함 6개를 탐지하고, 예외처리 결함은 탐지하지 못하는 것으로 나타났다. 방위사업청 코딩 규칙을 적용할 경우에는 메모리 결함 7개, 병행성 결함 1개, 예외처리 결함 1개를 탐지하는 것으로 나타났다. 즉 코딩 규칙 검증을 통해서는 메모리 결함은 100%, 병행성 결함은 28.57%, 예외처리 결함은 6.25% 탐지 가능하였다.

• **메모리 결함의 경우,** Fig. 2(a)와 같이 MISRA C/C++은 12개 결함 모두, 방위사업청 코딩 규칙은 7개의 결함에 대해 탐지 가능한 것으로 나타났다. MISRA C 2012의 Rule 21.3에 따르면 동적 메모리 할당을 사용하지 않도록 하고 있으므로 해당 규칙을 따르는 경우에는 메모리 결함이 발생하지 않도록 할 수 있다. 방위사업청 코딩 규칙 중 포인터/배열과 관련된 규칙에 따르면 메모리 접근과 관련된 M04, M07~M12의 결함이 탐지될 수 있다.

• **병행성 결함의 경우,** Fig. 2(b)와 같이 MISRA C/C++은 C14에서 C19까지의 6개 결함에 대해 탐지 가능하고, 방위사업청 코딩 규칙의 경우 C16에 대해 탐지 가능한 것으로 나타났다. 즉, MISRA Rule 8.9와 Rule 18.6에 따르면 자원 사용과 관련된 사용 순서 위반 결함은 탐지가 가능하므로 이와 관련한 C14부터 C19까지의 결함은 탐지가 가능하다. 그러나 교착상태(C01~C07)나 원자성 위반(C08~C13), 공유메모리에 대한 접근 순서 위반(C20~C21)은 다른 스레드 혹은 프로세스의 실행 상태에 따라 결함의 발생 여부가 달라지는 것이므로 사전에 결함으로 탐지하기가 어렵다.

• **예외처리 결함의 경우,** Fig. 2(c)와 같이 MISRA C/C++은 탐지 가능한 결함이 존재하지 않았고, 방위사업청 코딩 규칙의 경우에는 E16에 대해 탐지가 가능하다. 방위사업청 코딩 규칙에 따르면 C++ 전용 규칙 중 'Exception specification에 기술되지 않은 모든 throw에 대하여 예외처리를 해야만 한다'는 규칙이 존재하므로 어플리케이션 예외처리 결함에 대해서는 사전 탐지가 가능하다. 그러나 디바이스 드라이버나 OS와 관련된 예외처리의 경우에는 관련 항목이 없어 탐지가 되지 않는다.

b) 취약점 점검

취약점 점검은 소프트웨어 소스 코드가 CWE 목록에 정의된 취약점을 포함하고 있는지 점검하는 활동으로, CWE-658, CWE-659를 통해 사전 탐지 가능한 결함은 Fig. 3과 같다.

CWE-658은 메모리 결함 11개, 병행성 결함 7개, 예외처리 결함 1개를 탐지 가능하고, CWE-659의 경우 메모리 결함 11개, 병행성 결함 7개, 예외처리 결함 2개를 탐지 가능한 것으로 나타났다. 즉 취약점 점검을 통해서는 메모리 결함은 91.67%, 병행성 결함은 33.33%, 예외처리 결함은 12.5% 탐지 가능하다.

• **메모리 결함의 경우,** Fig. 3(a)와 같이 CWE-658과 CWE-659 모두 M04를 제외한 11개의 결함에 대해 탐지 가능한 것으로 나타났다. M04는 NULL 포인터 해제 결함으로 CWE-658과 CWE-659에 관련된 점검 항목이 존재하지 않는다. 이는 대부분의 컴파일러에서 'free(NULL);'을 호출하였을 때 아무 동작도 하지 않기 때문이다. 그러나 gcc의 경우 'free(variable);'을 호출할 때 variable이 이미 해제된 상태라면 NULL 포인터에 대한 해제를 시도하므로 시스템 크래쉬가 발생할 수 있어 탐지가 되어야 한다.

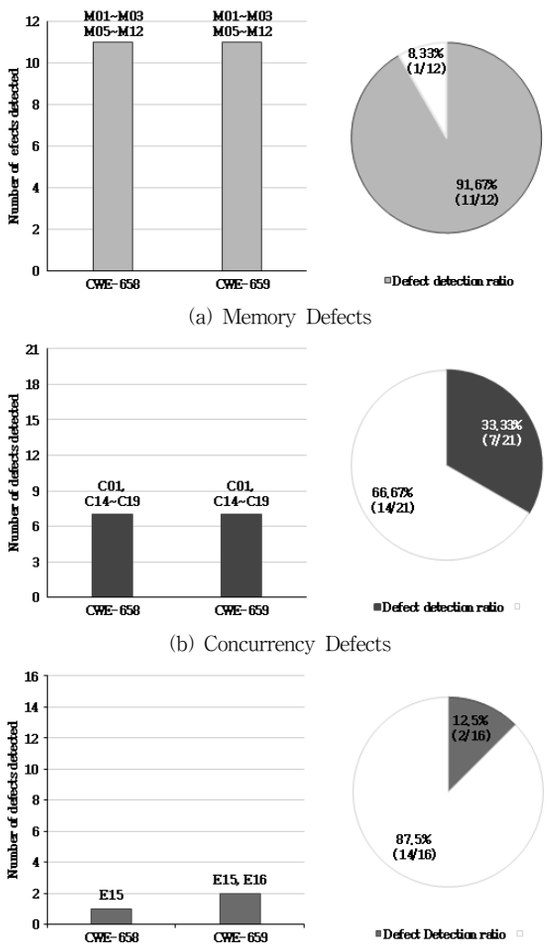


Fig. 3. Defect Detection Ratio of CWE Check

• **병행성 결함의 경우,** Fig. 3(b)와 같이 CWE-658과 CWE-659 모두 C01과 C14에서 C19까지의 7개 결함에 대해 탐지 가능한 것으로 나타났다. C01의 경우 동기화 객체나 공유 메모리에 대한 경합 조건에 대한 점검을 수행하므로 탐지가 가능하고, C14~C19의 경우 스레드 혹은 프로세스 내부에서의 호출 순서에 대한 점검을 수행하므로 결함을 사전 탐지 할 수 있다. 그러나 통신 과정에서 발생하는 데드락(C2~C07)이나 원자성 위반(C08~C13), 공유 메모리에 대한 접근 순서 위반(C20~C21)은 여전히 취약점 점검만으로는 탐지가 어렵다.

• **예외처리 결함의 경우,** Fig. 3(c)와 같이 CWE-658은 E15, CWE-659는 E15와 E16에 대해 탐지가 가능하다. 즉, CWE-658과 CWE-659에 시그널 처리와 관련된 점검 항목이

존재하므로 E15에 대해 탐지가 가능하고, CWE-659에는 C++에서 사용하는 예외처리에 대한 점검항목이 존재하므로 E16에 대해서도 탐지가 가능하다. 그러나 두 방안 모두 디바이스 드라이버와 관련된 예외 결함(E01~E14)에 대해서는 점검 항목이 존재하지 않아 탐지가 어렵다.

코딩 규칙 검증이나 취약점 점검만으로도 메모리 결함이나 일부 병행성, 예외처리 결함 중 탐지가 가능한 결함이 존재한다. 그러나 여전히 탐지가 되지 않는 병행성, 예외처리 결함이 존재한다. 정적 검증으로 탐지가 가능한 결함 중에서도 소프트웨어의 실행에 따라 발생 여부가 달라지는 결함이 존재할 수 있고, 코딩 규칙을 회피하는 소스 코드나 코딩 규칙 검증이 적용되지 않는 소프트웨어(기개발 소프트웨어, 상용 소프트웨어)[3]가 존재할 수도 있다. 또한 취약점 점검이 되었다 하더라도 시스템의 제한된 자원으로 인한 메모리 할당 실패와 이로 인한 잘못된 메모리 접근 등의 결함은 여전히 발생할 수 있다. 따라서 정적 검증을 완벽하게 통과한 이후에도 동적 검증을 통해 소프트웨어 결함에 대한 탐지가 필요함을 알 수 있다.

3.2 동적 검증 도구의 소프트웨어 결함 탐지 분석

Table 1 소프트웨어 결함에 대해 무기체계 소프트웨어 신뢰성 시험을 위한 동적 검증 도구를 적용하였을 때 이들 결함이 얼마나 탐지가 될 수 있는지 분석하고자 한다. 동적 검증에 사용되는 도구는 크게 코드 커버리지를 측정하기 위한 도구와 디버깅을 위한 결함 탐지를 위한 도구로 구분할 수 있다. 그러나 코드 커버리지 측정 도구를 이용해 커버리지를 100% 만족한다 하더라도 여전히 소프트웨어 결함은 존재할 수 있다. 기존의 문장, 분기 실행률과 MC/DC 실행률만으로는 소프트웨어가 통합되어 상호 작용하는 과정에서 발생하는 결함을 탐지하기에는 한계가 존재하기 때문이다. 또한 소프트웨어 결함을 발견하는 데는 커버리지 측정률보다는 커버리지를 만족시키는데 사용한 테스트데이터에 영향을 받기 때문이다.

본 장에서는 디버깅을 위한 결함 탐지 도구를 적용하였을 때의 Table 1의 소프트웨어 결함 탐지 여부를 실험을 통해 분석하고자 한다. 한편 추가적으로 동적 결함 탐지 방식은 아니나 관련 연구의 2.1에서 설명한 런타임 오류 정적 분석으로 소프트웨어 결함을 얼마나 탐지할 수 있는지도 본 실험을 통해 분석하고자 한다.

1) 실험 설계

첫째, 실험 대상 소프트웨어는 국내 무기체계 소프트웨어 개발 과정에서 사용하는 오픈 소스 소프트웨어 가운데 선택하였다. 무기체계 소프트웨어에서 사용하는 오픈 소스 6933개를 분석하였고, 그 중에서 1) Linux 기반에서 동작 가능한 C/C++ 어플리케이션이며 2) 메모리/병행성/예외처리 결함이 발생할 수 있는 함수를 사용하는 어플리케이션으로 SQLite[9]를 선정하였다. SQLite는 임베디드 시스템에 맞춰 경량화된 데이터베이스 관리 시스템이다. 실험에 사용한 버전은 3.24.0이고 소스 코드 파일의 크기는 8.5MB이다. 그러나 SQLite로는 모든 결함을 적용할 수 없어 적용이 불가한 결함 유형에 대해서는 어플리케이션(Server_app, Client_app, DeviceControl, CharDevDriver, IHChangeLib)을 추가로 작성하였다. Table 3은 본 실험에 사용된 어플리케이션이다.

Table 3. Target Applications

Application	LOC	Language	Description
SQLite	240,206	C	Database management system
Server_app	112	C/C++	A program that waits for a message to be sent from the client. When the message is received, displays the message contents and sends the message back to the client.
Client_app	81	C/C++	A program that sends a string as a message when a connection is made to the server, and outputs a message received from the server.
Device Control	168	C/C++	Test application for accessing and controlling 'CharDeviceDriver'
CharDev Driver	313	C/C++	Device drivers for char devices on Linux systems
IHChange Lib	78	C/C++	A library that replaces the application's interrupt handler routines to test OS exception handling

Table 4. Dynamic and Static Tools

Category	Tool	Defect to be detected
Dynamic Defect Detection	AMOS v1.0[10]	Memory defects
	AMOS v3.0[11]	Exception handling defects
	Valgrind Toolset[12]	Memory defects, Concurrency defects
	Sanitizer[13]	Memory defects
Runtime Error Static Analysis	Clang	Runtime error
	Splint	Runtime error
	CppCheck	Runtime error
	Lint	Runtime error
	Relay	Runtime error
	gcc	Runtime error

둘째, 분석 대상 도구는 Table 4의 동적 결함 탐지 도구 4개와 런타임 오류 정적 분석 도구 6개를 대상으로 하였다. 적용 도구는 1) C/C++ 어플리케이션을 대상으로 하는 도구이며, 2) 실험을 위해 사용될 환경인 ARM 기반의 리눅스에 적용이 가능한 도구이다.

셋째 동적 결함을 발생시키기 위하여 뮤테이션 기법을 이용해 결함을 인위적으로 주입하였다. 실험대상 어플리케이

션에는 모든 동적 결함을 포함하고 있지 않기 때문이다. 메모리 결함의 경우에는 F.Wu 등 방안[14]과 H.Shahriar 등의 방안[15]을 적용하여 18개의 뮤턴트를 생성하였고, 병행성 결함의 경우에는 Markus Kusano와 Chao Wang이 제안한 CCmutator[16]와 Alper Sen이 제안한 뮤테이션 연산자[17]를 적용하여 97개의 뮤턴트, 예외처리 결함의 경우에는 Kim 등이 제안한 뮤테이션 연산자[18]를 적용하여 25개의 뮤턴트를 생성하였다. 각각의 결함 유형별 뮤턴트를 실험 대상 소프트웨어에 1개씩 적용한 결과 동일한 뮤턴트와 컴파일이 되지 않는 뮤턴트, 결함으로 동작하지 않는 뮤턴트를 제외하고 메모리 11개, 병행성 21개, 예외처리 16개의 결함을 발생시켰다.

2) 분석 결과

동적 결함 탐지 도구 및 런터임 오류 정적 분석 도구에서 Table 1 소프트웨어 결함을 얼마나 탐지할 수 있는지를 분석하도록 한다.

a) 동적 결함 탐지 도구

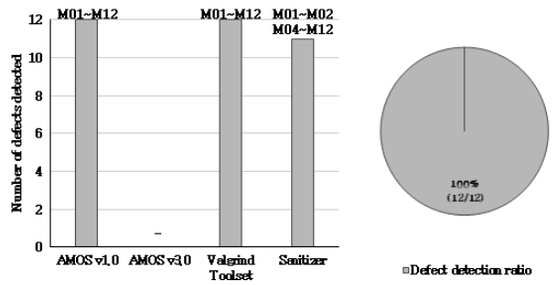
동적 결함 탐지 도구의 경우, Fig. 4와 같이 메모리와 예외처리 결함에 대해서는 모두 탐지를 하였다. 병행성 결함의 경우에는 7개의 결함이 탐지되었다. 탐지된 결함은 데드락 결함 중 동기화 객체 데드락(C01)과 순서 위반 결함의 일부(C14~C19)이다.

• **메모리 결함의 경우,** Fig. 4(a)와 같이 메모리 결함을 탐지 대상으로 하는 AMOS v1.0과 Valgrind toolset, Sanitizer에서 각각 12개, 12개, 11개의 결함을 탐지하였다. Sanitizer에서 탐지하지 못한 결함인 M03은 메모리 누수 결함으로 프로그램이 종료된 후 해제되지 않고 남아있는 메모리를 결함으로 보지만, Sanitizer의 경우 프로그램이 종료된 이후 남아있는 메모리에 대해서는 확인하지 않아 결함으로 탐지하지 못하였다.

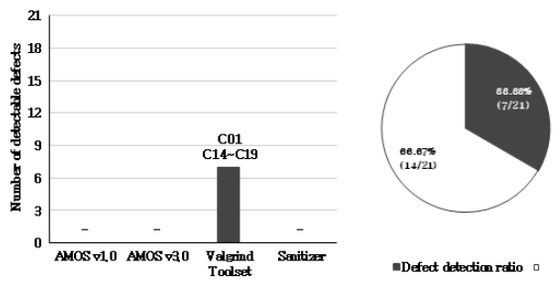
• **병행성 결함의 경우,** Fig. 4(b)와 같이 Valgrind toolset에서 C01과 C14에서 C19까지의 7개 결함에 대해 탐지 가능한 것으로 나타났다. C01의 경우 동기화 객체나 공유 메모리에 대한 경합과 관련된 함수에 대한 모니터링을 수행하므로 탐지가 가능하고, C14~C19의 경우 스레드 혹은 프로세스 내부에서의 호출 순서에 대한 모니터링을 수행하므로 결함을 탐지하였다.

그러나 Fig. 5의 예처럼 동기화 메시지를 서버에서 처리하지 못하여 발생하는 데드락(C04)과 같은 통신 데드락(C2~C07)의 경우에는 send() / recv()와 같은 통신 관련 함수를 모니터링 하지 않아 결함으로 탐지하지 못하였다. 또한 Fig. 6과 같이 동기화로 보호되지 않는 변수나 메모리에 대한 원자성 위반(C08~C13)이나 동기화로 보호되지 않는 공유 메모리에 대한 순서 위반(C20~C21)에 대해서는 모니터링이 되지 않나 결함으로 탐지되지 않았다.

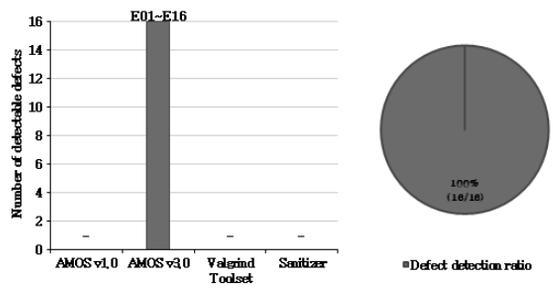
• **예외처리 결함의 경우,** Fig. 4(c)와 같이 예외처리 결함 탐지 도구인 AMOS v3.0에서 모든 결함을 탐지하였다.



(a) Memory Defects



(b) Concurrency Defects



(c) Exception Defects

Fig. 4. Defect Detection Ratio of Dynamic Defect Detection Tools

```
...
while ( (cnt = recv(in, buf, 256,0)) > 0)
{
    fprintf(stdout,
      "Server:: Received - %s", buf);
    sleep(TIMELIMIT);
    send(in, buf, strlen(buf), 0);
    memset(buf, 0, 256);
}
...
```

(a) Server_app

```
...
in = send(id, buf, strlen(buf), 0);
if(in < 0){
    fprintf(stderr, "Error!₩n");
    return 0;
}
fprintf(stdout,
  "Client: Send - %s₩n", buf);

memset(buf, 0, 256);
in = recv(id, buf, 256, 0);
if(in < 0){
    fprintf(stderr, "Error!!₩n");
    return 0;
}
fprintf(stdout,
  "Client: ACK - %s₩n", buf);
```

(b) Client_app

Fig. 5. Example of a Message Deadlock

b) 런타임 오류 정적 분석 도구

런타임 오류 정적 분석 도구를 적용한 결과, Fig. 7과 같이 메모리 결함에 대해서는 Clang, Splint, Cppcheck가 각각 5개, 6개, 7개의 결함을 탐지하였고, 병행성 결함은 Cppcheck가 3개, Lint와 Relay가 각각 1개의 결함을 탐지하였다. 예외처리 결함의 경우에는 gcc에서 1개의 결함을 탐지하였다. 메모리 결함의 경우 각 도구들이 탐지한 결함이 달라 세 도구를 모

```
...
int total;
//pthread_mutex_lock(…);
total = get_total(); //read from server
total ++;
save_total(total); //write to server
//pthread_mutex_unlock(…);
...
```

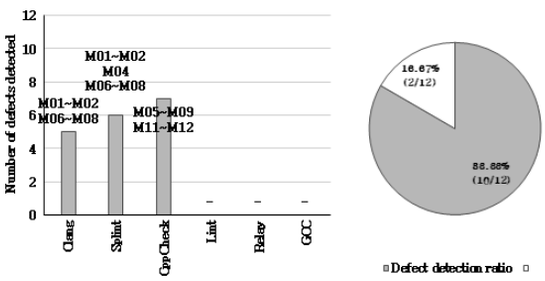
(a) Thread 1 of Client_app

```
...
int total;
//pthread_mutex_lock(…);
total = get_total(); //read from server
total --;
save_total(total); //write to server
//pthread_mutex_unlock(…);
...
```

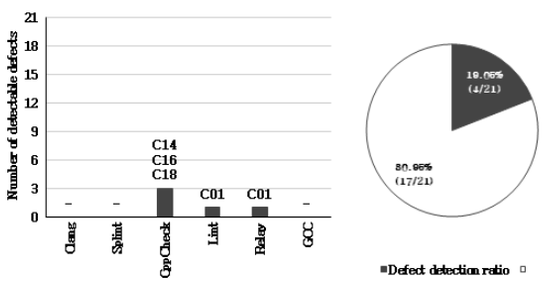
(b) Thread 2 of Client_app

Fig. 6. Example of an Atomicity Violation due to using Variable

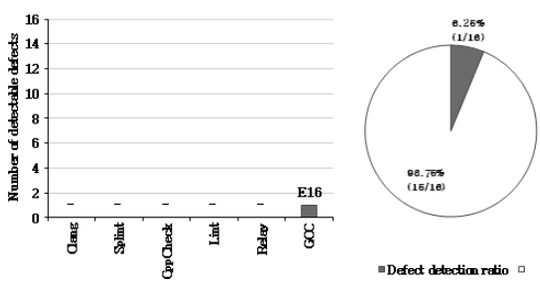
두 결함 탐지에 적용할 경우 10개의 결함에 대해 탐지가 가능하고, 병행성 결함의 경우에는 Cppcheck과 Lint/Relay에서 탐지한 결함이 달라 4개의 결함에 대해 탐지가 가능하다. 즉 런타임 오류 정적 분석 도구를 통해서는 메모리 결함 83.33%, 병행성 결함 19.05%, 예외처리 결함 6.25%에 대해 탐지 가능하였다.



(a) Memory Defects



(b) Concurrency Defects



(c) Exception Defects

Fig. 7. Defect Detection Ratio of Runtime Error Static Analysis Tools

• **메모리 결함의 경우,** Fig. 7(a)와 같이 Clang에서 5개 결함(M01~M02, M06~M08)을 탐지하였고, Splint에서 6개 결함(M01~M02, M04, M06~M08), Cppcheck에서 7개 결함(M05~M09, M11~M12)을 탐지하였다. 세 가지 도구를 적용하였을 때 탐지하지 못하는 결함은 메모리 누수(M03)와 할당되지 않은 포인터 접근(M10)이다. 이 두 결함의 경우 실제 실행 에 따라 결함 발생이 달라지므로 런타임 오류 정적 분석 도구로는 탐지가 어려운 결함이다.

• **병행성 결함의 경우,** Fig. 7(b)와 같이 Cppcheck에서 3개 결함(C14, C16, C18)과 Lint와 Relay에서 C01 결함을 탐지하였다. 런타임 오류 정적 검증 도구가 비 실행 기반이므로 소스 코드 자체의 병행성과 관련된 함수 문법 오류 혹은 변수에 대한 접근은 분석이 가능하여 결함으로 탐지하였으나, 병행성 결함의 경우 실제 실행 결과가 결함 분석에 필요한 경우가 많아 결함 탐지에 한계가 존재한다.

• **예외처리 결함의 경우,** Fig. 7(c)와 같이 gcc에서 try-catch 예외처리와 관련된 결함(E16)에 대해 탐지하였다. 그러나 디바이스 드라이버와 관련된 결함(E01~E14)과 OS 예외처리 결함(E15)에 대해서는 탐지하지 못하였다.

동적 결함 탐지 도구의 경우 메모리와 예외처리 결함을 모두 탐지하였으나, 병행성 결함의 경우 도구에서 모니터링 대상이 아닌 함수의 사용으로 인한 결함이나, 변수에 대한 추적이 필요한 결함에 대해서는 탐지하지 못하는 한계가 있었다. 런타임 오류 정적 검증 도구의 경우, 런타임 오류를 탐지하기 위한 목적으로 개발이 되었다 하더라도 비실행 기반이므로 소프트웨어 결함 탐지에 한계가 있다. 특히 Fig. 4와 같은 통신 데드락이 발생하는 경우, 소스 코드 자체의 문법에는 오류가 없으며, send/recv 함수의 사용 순서도 올바른 순서로 되어 있으므로 결함으로 탐지되지 않는다. 그러나 서버에서 클라이언트에 응답을 보내기 위한 send() 함수를 호출하기 전 sleep() 함수를 호출함으로써 데드락이 발생하게 된다. 서버에서 send() 함수가 호출되지 않는다면 클라이언트에서는 응답을 기다리고 있는 상태가 되고, 서버 역시 다른 클라이언트로부터의 요청을 받지 못하는 상태가 되기 때문이다. 이와 같이 결함 탐지에 실제 실행 결과가 필요한 결함에 대해서는 런타임 오류 정적 검증 도구의 결함 탐지 효과가 미미하게 나타났다.

3.3 무기체계 소프트웨어 신뢰성 시험 개선 방안

서론에서 언급하였듯이 무기체계 소프트웨어 개발 프로세스[3]에서의 동적 시험은 전체 개발 소프트웨어를 단번에 묶어 검증한다[3]. 그러나 이러한 방식은 엄밀히 말하면, 개발 과정에서 단위 소프트웨어(CSU), 구성품(CSC) 및 소프트웨어 형상항목(CSCI)이 통합되면서 상호 작용하는 과정에 발생할 수 있는 결함에 대한 신뢰성 검증 방식으로는 한계가 있다. 각 소프트웨어 구성 요소들이 단계적으로 결합하며 상호 작용하는 과정에서 발생할 수 있는 동적 결함을 찾아내기 어렵고, 결함을 찾아내더라도 그 원인을 찾기 어렵기 때문이다.

또한, 현재의 무기체계 소프트웨어 개발 프로세스에서는 신뢰성 시험으로써 정적 시험과 동적 시험으로 요구사항 기반의 코드 실행률 측정만을 수행하는데, 이러한 소프트웨어

검증 방식으로는 소프트웨어 결함을 모두 걸러낼 수 있도록 하는 데에는 한계가 있다. 즉, 3.1절의 분석 결과에 따르면 정적 검증만으로는 무기체계 소프트웨어 결함을 모두 탐지하지 못하였다. 이것은 결함이 실행 과정에 따라 발생 여부가 달라지기도 하고 실행 환경에 밀접하게 관련되어 있는 경우가 많기 때문이다. 동적 시험에서는 LDRA tool, Cantata++, VectorCast, CodeSonar 등의 도구를 사용한다. 그러나 이 도구들은 코드 실행률 측정을 위한 도구이므로 결함 발견을 위한 도구와는 사용 목적이 다르다. 따라서 이러한 도구들로 커버리지를 만족시키더라도 여전히 결함이 발생할 가능성이 존재한다. 동적 시험 평가 수행 중에 결함이 발생하여 동적 결함 탐지 도구를 이용하여 디버깅을 하게 되면 해당 부분에 대해 신뢰성 시험을 재실시하여야 하므로 개발 기간이 길어질 뿐만 아니라, 동적 결함 탐지 도구로도 탐지가 되지 않는 복잡한 결함의 경우에는 디버깅에도 매우 오랜 시간이 걸리게 된다. 3.2절의 정적 검증에서 활용할 수 있는 런타임 오류 정적 검증 도구의 경우에서도 소프트웨어 결함 탐지에는 한계가 있었다.

본 실험 분석을 통하여 무기체계 소프트웨어 검증 개선 방안을 다음과 같이 도출할 수 있다.

> “무기체계 소프트위어 신뢰성 시험을 위한 동적 검증 수행 시, 커버리지 측정과 아울러 Table 1의 소프트웨어 결함이 없음을 확인하는 과정을 추가.”

소프트웨어 동적 결함이 없음을 확인하기 위해서는 동적 결함 탐지 도구를 활용하면 된다. 좀 더 구체적으로 말하면, 동적 검증을 수행하며 코드 실행률을 측정하는 동안에 메모리, 병행성, 예외처리 각 결함 군을 탐지할 수 있는 동적 결함 탐지 도구를 함께 탑재하여 동작시키도록 하는 것이다. 이렇게 함으로써 코드 실행률 측정은 물론이고 동적 시험 동안에 발생할 수 있는 결함들에 대해 사전에 탐지하도록 함으로써 무기체계 소프트웨어의 신뢰성을 향상시킬 수 있다.

그러나 3.2절에서 분석한 바와 같이 현재의 동적 결함 탐지 도구로는 표4의 소프트웨어 결함 중 병행성 결함에 대해서는 일부 결함을 탐지하지는 못한다. [4]에 따르면 무기체계 소프트웨어에는 병행성 결함이 많이 내포할 수 있으며, 단위 소프트웨어(CSU), 구성품(CSC) 및 소프트웨어 형상항목(CSCI)의 상호 작용에 따라 발생하는 결함 해결하기도 어려운 결함이다. 이러한 병행성 결함들은 반드시 해결이 되어야 한다. 이를 위해 병행성 결함 탐지 방안에 대한 최신의 연구들을 활용하여 무기체계 소프트웨어의 결함 탐지 도구로 구현하도록 하거나, 현재 소프트웨어 결함 탐지 도구의 성능을 개선하도록 하여야 한다.

4. 결론 및 향후 연구

소프트웨어의 신뢰성을 향상시키기 위한 연구는 꾸준히 이루어져왔다. 그 결과로 정적 및 동적 검증과 관련된 다양한 기법들이 연구되었다. 정적 검증과 동적 검증은 그 목적이 다르므로 각각의 검증 단계에서 모두 수행하여야 한다. 그러나 무기체계 소프트웨어 개발을 포함한 대부분의 소프트웨어 검증 평가 방법(무기체계 소프트웨어 개발에서는 신뢰성 시험에 해당)으로 정적 검증을 수행한 이후 동적 검증으로 코드 실행률 측정 만을 수행하고 있다. 이 경우 신뢰성 시험이 통과되었다 하여도 탐지되지 않은 결함이 존재할 수 있다.

본 논문에서는 소프트웨어에서 발생할 수 있는 결함에 대하여 무기체계 소프트웨어의 정적 검증 방안 및 동적 결함 탐지 도구, 런타임 오류 정적 분석 도구가 얼마나 이들 결함을 탐지할 수 있는지 분석하였다. 그 결과 정적 검증을 통해서도 소프트웨어 결함의 일부를 탐지한다는 점을 보였다. 또한 정적 분석을 통한 런타임 오류를 탐지하는 도구를 통해서도 일부 결함을 탐지할 수 있음을 보였다. 그러나 이러한 정적 검증 방안이나 런타임 오류 정적 분석 도구를 적용한다 하더라도 여전히 탐지되지 않는 결함이 존재한다. 동적 결함 탐지 도구로 디버깅하여도 모든 소프트웨어 결함을 탐지하기는 못하였다. 따라서 정적 검증 이후 소프트웨어 코드 실행률 측정으로 이루어지는 현재의 소프트웨어 신뢰성 시험 검증 평가 방법에서 Table 1의 소프트웨어 결함이 없음을 확인하는 평가를 추가하는 개선점을 도출할 수 있었다.

향후 현재의 동적 결함 탐지 도구로는 탐지가 어려운 결함을 탐지할 수 있는 도구를 개발하고 있으며, 그 효과를 분석하는 연구를 추진할 계획이다.

References

[1] J. Kim, S. Jeong, I. Hwang, H. Cho, D. Kim, and Y. J. Jang, “M&S Verification, Validation and Accreditation Research Direction Considering the Characteristics of Defense M&S,” *Journal of Korean Institute of Industrial Engineers,* Vol.39, No.6, pp.486-497, 2013.

[2] Kyeongyoun Kwon, Joonseok Joo, Taesik Kim, Jinwoo Oh, and Jihyun Baek, “A Study on Quality Assurance of Embedded Software Source Codes for Weapon Systems by Improving the Reliability Test Process,” *Journal of KIISE*, Vol.42, No.7, pp.860-867, 2015.

[3] Weapon System Software Development and Management Manual, “Defense Acquisition Program Administration Manual No.2017-8,“ 2017.

[4] Jihyun Park and Byoungju Choi, “Analysis on Dynamic Software Defects for Increasing Weapon System Reliability,“ *Journal of KIPS Tr. Software and Data Eng.*, Vol.7, No.7, pp. 249-258, 2018.

[5] Motor Industry Software Reliability Association. MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems: March 2013. Motor Industry Research Association, 2013.

[8] S. N. Cant, D. R. Jeffery, and B. Henderson-Sellers, “A Conceptual Model of Cognitive Complexity of Elements of the

Programming Process," *Information and Software Technology,* Vol.37, No.7, pp.351-362, 1995.

[10] Jooyoung Seo, Byoungju Choi, and Suengwan Yang, "A Profiling Merhod by PCB Hooking and its Application for Memory Fault Detection in Embedded System Operational Test," *Journal of Information and Software Technology*, Vol. 53, No.1, pp.106-117, Jan. 2011.

[11] Jooyoung Seo, Byoungju Choi, and Sihyun Lee, "Software Generated Device Exception for more Intensive Device-related Software Testing: An Industrial Field Study," *Journal of Systems and Software,* Vol.86, No.12, pp. 3193-3212, Dec. 2013.

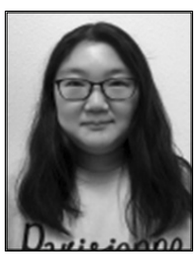
[14] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke, "Memory Mutation Testing," *Information and Software Technology*, Vol.81, pp.97-111, 2017.

[15] Hossain Shahriar and Mohammad Zulkernine, "Mutation-based Testing of Buffer Overflow Vulnerabilities, Computer Software and Applications," *(COMSAC'08) 32nd Annual IEEE International*, pp.979-984. 2008.

[16] Markus Kusano and Chao Wang, "CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications," in *Proceeding of the 28th IEEE/ACM International Conference on Automated Software Engineering,* pp.722-725, 2013.

[17] Alper Sen, "Mutation Operators for Concurrent SystemC Designs," in *Microprocessor Test and Verification (MTV),* 10th International Workshop on, pp.27-31, Dec. 2009.

[18] Sunwoo Kim, John A. Clark, and John A. Mcdermid, "Class Mutation: Mutation Testing for Object-oriented Programs," *In Proc. Net. ObjectDays. Erfurt,* Germany: Net. Objects, pp.9-12, 2000.



박 지 현

https://orcid.org/0000-0003-4478-7565
e-mail : pola0527@ewhain.net
2006년 이화여자대학교 컴퓨터학과 (학사)
2019년 이화여자대학교 컴퓨터공학과 (석・박사)
2019년~현 재 이화여자대학교 컴퓨터공학과 박사후과정
관심분야 : 임베디드 소프트웨어 테스트, 결함 주입 테스트

최 병 주

http://orcid.org/0000-0003-3985-7645
e-mail : bjchoi@ewha.ac.kr
1983년 이화여자대학교 수학과 (학사)
1990년 퍼듀대학교 컴퓨터학과 (석・박사)
1995년~현 재 이화여자대학교 컴퓨터공학과 교수
관심분야 : 소프트웨어 검증 평가, 소프트웨어 테스트