J. Korean Soc. Aeronaut. Space Sci. 48(10), 821-829(2020)
DOI:https://doi.org/10.5139/JKSAS.2020.48.10.821
ISSN 1225-1348(print), 2287-6871(online)

항공기 소프트웨어의 건전성 관리를 위해서 순서 위배 오류를 자율 수리하는 효율적인 시스템

김태형[1], 최으뜸[2], 전용기[3]

An Efficient On-the-fly Repairing System of Order Violation Errors for Health Management of Airborne Software

Tae-Hyung Kim[1], Eu-Teum Choi[2] and Yong-Kee Jun[3]
Department of Informatics, Gyeongsang National University[1,2,3]

ABSTRACT

Health management system of airborne software repairs runtime errors to provide safety and to reduce cost of maintenance. It is critical to on-the-fly repair order violation errors, because it is difficult to identify them at the development phase. Previous work, called Repairing Atomicity Violations (Repairing-AV) diagnoses order violations for each access event by comparing execution order of accesses. As a result, Repairing-AV has time overhead that is proportional to the number of access events to shared variable. This paper presents a tool called On-the-fly Repairing System (ORS) that can repair order violations of object methods containing access events. The ORS diagnoses order violations by using correct order of object methods, and treats them by stalling its thread where the error is about to occur. Experimentation with five synthetic programs shows that ORS is more efficient than Repairing-AV when the number of access events is greater than sixty.

초 록

항공기 소프트웨어의 건전성 관리시스템은 수행 중에 발생하는 오류를 수리하여 안전성을 제공하고 유지보수 비용을 절감한다. 순서 위배 오류는 개발 단계에서 모두 제거하는 것이 어렵기 때문에 운용 단계에서 자율적으로 수리할 수 있어야 한다. 순서 위배를 자율 수리하기 위한 기존 연구는 각 접근사건의 수행 직전에 접근사건 순서를 비교하여 오류를 진단하기 때문에 접근사건 수에 비례하는 시간 오버헤드를 발생시킨다. 본 논문은 접근사건을 포함하는 함수의 올바른 순서 정보로 오류를 진단하고 해당 함수를 지연시켜 조치하는 기법인 ORS를 제시한다. ORS를 평가하기 위해, 순서 위배 오류를 포함하는 5가지의 합성 프로그램에 기존 연구와 ORS를 적용하여 시간 오버헤드를 측정하였다. 그 결과, 접근 횟수가 약 60번 이상일 때 기존 연구보다 효율적임을 확인하였다.

Key Words : Airborne Software(항공기 소프트웨어), Health Management System(건전성 관리 시스템), Order Violation Errors(순서 위배 오류), On-the-fly Repairing(자율 수리)

† Received : June 5, 2020 Revised : September 15, 2020 Accepted : September 16, 2020
[1,2] Graduate Student, [3] Professor
[3] Corresponding author, E-mail : jun@gnu.ac.kr, ORCID 0000-0002-4753-3651

Ⅰ. 서 론

항공기 소프트웨어[1]는 항공기 운용에 요구되는 기능을 제공하는 임베디드 소프트웨어이다. 항공기에서 소프트웨어가 차지하는 비율은 1960년에 생산된 F-4가 8%였지만 2007년에 생산된 F-35는 90%까지 증가하였다[2]. 소프트웨어 규모가 클수록 복잡성이 증가하기 때문에 잠재적인 오류가 내재할 확률이 높아진다. 이러한 잠재적인 오류는 시스템의 안전성을 보장하기 어렵게 하므로 반드시 제거해야 한다. 항공기 소프트웨어는 개발 단계의 검증을 통해 잠재적인 오류를 탐지하여 제거하고 있다. 하지만 잠재적인 오류를 개발 단계에서 모두 제거하는 것은 어렵다[3].

항공기 소프트웨어의 잠재적인 오류 때문에 일어난 사고는 F-22 Raptor와 Boeing 737 MAX의 사례가 있다. Raptor의 사고 사례[4]는 조종사 유도진동을 막지 못한 비행 제어 소프트웨어의 오류 때문에 시험 비행 후 불시착한 사고이다. 이 사고로 불시착 시에 기체의 손상이 있었다. 737 MAX의 사고 사례[5]는 비행 제어 소프트웨어가 기수를 아래로 향하도록 조정하여 운항 중에 여객기가 추락한 사고이다. 두 번의 사고에서 탑승객 전원 총 346명이 사망했다. 이와 같은 사고를 방지하기 위해, 소프트웨어 오류가 발생하더라도 사고로 이어지지 않도록 즉시 수리하는 시스템이 필요하다.

항공기 소프트웨어의 건전성 관리시스템[3,4,6]은 소프트웨어 오류를 수행 중에 진단하고 조치하여 프로그램이 정상적으로 수행되도록 하는 것이 목적이다. 진단은 프로그램의 수행을 실시간으로 감시하여 오류를 탐지한다. 그리고 조치는 탐지된 오류를 대상으로 적절한 복구 기법을 적용하여 프로그램이 정상적으로 수행되도록 한다. 건전성 관리시스템이 적용되면 수행 중에 시스템의 건전성을 관리하여 안전성을 제공하고 유지보수 비용을 경감 할 수 있다[7].

항공기 소프트웨어에서 발생할 수 있는 오류 중, 동시성 오류[6,8-11]는 병행 프로그램에서 적절한 동기화 기법이 없이 적어도 하나의 쓰기 사건을 포함한 공유변수 접근이 있을 때 발생한다. 동시성 오류 중, 순서 위배 오류[9,10]는 개발자가 의도한 스레드 간의 공유변수 접근사건 순서를 보장할 수 없는 오류이다. 이 오류는 스레드 사이의 모든 인터리빙을 고려하여 오류를 디버깅할 수 없어 잠재적으로 프로그램에 내재될 수 있다[10]. 잠재된 순서 위배 오류가 발생하면 개발자가 의도하지 않은 비결정적인 결과를 초래하여 시스템의 실패가 발생할 수 있다.

건전성 관리시스템으로 항공기 소프트웨어의 순서 위배 오류를 자율적으로 수리하는 연구는 Lee[11]가 있다. 이 연구는 항공기 소프트웨어를 사전에 시험하여 스레드 간의 공유변수 접근사건에 대한 올바른 접근순서 정보를 수집한다. 이후, 운영 단계에서 접근순서 정보를 수행 중 접근순서와 비교하여 순서 위배 오류를 진단한다. 오류가 진단되면 오류가 진단된 스레드의 접근사건을 지연시켜 올바른 순서가 되도록 조치한다.

하지만 기존 연구의 진단 방식은 프로그램 내부의 공유변수 접근사건 수에 비례하는 시간 오버헤드를 발생시킨다. 이는 수행 중에 발생하는 오류를 진단하기 위해 모든 접근사건의 수행 직전에 접근사건 순서를 비교하기 때문이다. 따라서 기존 연구의 기법을 적용하여 순서 위배 오류를 진단하는 것은 항공기 시스템의 실시간성을 만족시키기 어렵다.

본 논문은 접근사건을 포함하는 함수 간의 순서 위배 오류를 자율적으로 수리하는 On-the-fly Repairing System(ORS)을 제시한다. ORS는 사전에 정의된 함수 수행 정보와 수행 중 함수 정보를 비교하여 오류를 진단한다. 오류가 진단되면 해당 스레드의 함수를 지연하는 방식으로 조치한다. 본 논문의 평가를 위해 기존 연구와 ORS의 접근 횟수 증가에 따른 수행시간 측정 실험을 통해 ORS의 효율성을 보인다.

본 논문은 다음과 같이 구성되어있다. 2장은 항공기 소프트웨어의 건전성 관리시스템, 순서 위배 오류, 그리고 관련 연구를 설명한다. 3장은 제시하는 기법의 구조, 모듈, 그리고 작동 예시를 설명한다. 4장은 ORS의 구현 환경, 시스템 시험, 그리고 성능 실험의 분석 결과를 설명한다. 마지막으로, 5장은 본 논문의 결론을 설명한다.

Ⅱ. 연구 배경

본 장은 항공기 소프트웨어의 건전성 관리시스템과 순서 위배 오류를 설명한다. 그리고 건전성 관리시스템으로 항공기 소프트웨어의 순서 위배 오류를 자율적으로 수리하는 연구를 분석하여 해당 기법의 문제점을 제시한다.

2.1 항공기 SW의 건전성 관리시스템

항공기 소프트웨어의 건전성 관리시스템[3,4,6]은 항공기 소프트웨어에서 발생하는 오류를 진단(diagnosis) 및 조치(treatment)하여 시스템의 건전성을 유지하는 것이 목적이다. 진단은 프로그램을 실시간으로 감시(monitoring)하여 실행 중인 소프트웨어에서 오류의 발생을 탐지(detection)하는 단계이다. 조치는 발생한 오류를 수리하기 위해 적절한 복구(recovery) 기법을 적용하여 소프트웨어가 정상적으로 수행되도록 하는 단계이다.

진단 단계는 감시와 탐지를 통해 오류를 진단한다. 감시는 오류의 종류에 따라 진단에 필요한 정보들을 감시한다. 필요한 정보는 일정 시간 만료, 예상 동작과의 차이, 특정 센서/모듈의 입출력 등이 있다. 감시되는 정보들을 기반으로 특정 값의 차이를 분석하

거나 탐지 프로토콜 등을 이용하여 오류의 발생을 탐지한다. 예를 들어, 항공기 소프트웨어에서 원자성 위배를 탐지하는 도구[6]는 수행 중에 공유변수, 동기화, 스레드의 사용을 감시한다. 그리고 이 정보를 통해 락 사용규칙(locking discipline)을 검사하여 원자성 위배를 탐지한다.

조치 단계는 수행 방식에 따라 forward recovery [6,12,13]와 backward recovery[14,15]로 나뉜다. Forward recovery는 오류를 진단하면 오류가 예측되는 해당 접근사건을 지연시켜 오류를 수리하는 기법이다. Backward recovery는 오류가 발생하지 않은 특정 시점의 체크포인트를 유지하여 오류가 발생했을 때 되돌리는 기법이다.

ARINC 653[1,6] 기반의 실시간 운영체제(Real-Time Operating System, RTOS)는 통합 모듈형 항공전자 시스템에서 사용되며 Health Monitor(HM) 기능을 사용한다. HM 기능은 오류의 수준을 정의하고 각 오류에 대한 복구 방법을 명시하여 상황에 맞는 조치를 진행하는 기능이다. 또한, 프로세스 수준의 오류인 경우엔 사용자가 정의한 오류 처리기(error handler)를 호출하여 조치를 진행할 수 있다.

이런 시스템을 통해 개발 단계에서 제거하지 못한 잠재적인 오류를 수행 중에 진단 및 조치하여 시스템에 안전성을 제공할 수 있다. 그리고 오류의 전파를 막고 소프트웨어 및 하드웨어를 포함하는 하위 시스템의 고장을 막아 건전성을 관리하여 전체 시스템의 유지보수 비용을 경감할 수 있다[7].

2.2 순서 위배 오류

순서 위배 오류[9,10]는 개발자가 의도한 스레드 간의 공유변수 접근사건 순서를 보장할 수 없는 동시성 오류의 한 종류이다. 공유변수에 대해 적절한 동기화를 사용하지 않고 쓰기 사건을 하나 이상 포함할 때 발생할 수 있다. 개발 시에 순서 위배 오류의 디버깅은 스레드 사이의 모든 인터리빙을 고려해야 하므로 모든 오류를 제거하는 것은 어렵다[10]. Zhou의 연구[16]에 따르면 일반적인 소프트웨어 오류와 비교하여 순서 위배 오류를 수정하기 위해서 17% 더 많은 개발자가 필요하고 46% 더 많은 파일에 영향을 주며 72% 더 많은 패치가 요구된다.

Figure 1은 순서 위배 오류의 사례를 나타낸 것으로, Mozilla에서 발생한 순서 위배 오류의 소스 코드 중 일부분이다. Thread 1의 ReadWriteProc()에서 공유변수 io_pending을 TRUE로 초기화(W1)한 후에 Thread 2의 DoneWaiting()에서 같은 공유변수를 FALSE로 변경(W2)하는 순서를 의도하였다. 하지만 PBReadAsync()가 예상보다 빠르게 진행되어 Thread 2의 DoneWaiting()에서 공유변수를 FALSE로 변경하고 Thread 1의 ReadWriteProc()에서 공유변수를 TRUE로 변경하는 순서로 실행될 수 있다. 이 실

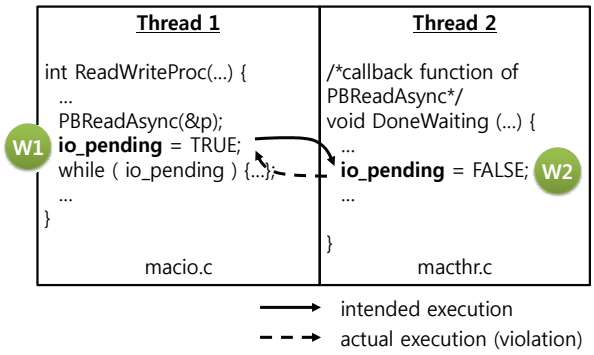

Fig. 1. An Example of Order Violation in Mozilla

행은 Thread 1이 while문을 탈출하지 못하게 하여 무한 루프를 발생시킨다. 이 예시는 PBReadAsync()가 개발자의 예상보다 빨리 진행되어 일어난 순서 위배 오류로, 개발 단계에서 코드상의 순서 위배 오류를 탐지하는 것이 어려운 것을 알 수 있다.

순서 위배 오류가 실제 사고로 이어진 사례로는 Facebook의 주식상장 과정에서 발생한 사고가 있다[17]. Facebook의 주식상장 시, 주식상장 이전 경매 프로세스인 Initial Public Offering(IPO) Cross에서 순서 위배 오류가 발생하였다. 결과적으로 주식상장 시간이 미뤄져 약 1,300만 달러에 달하는 금전적인 피해가 발생하였다. 뿐만 아니라, 콜롬비아 우주 왕복선의 소프트웨어에서도 발생되어 첫 비행 20분전에 중단되었던 사례도 있다[18].

운영 단계에서 순서 위배 오류가 발생하면 시스템의 실패로 이어질 수 있어 인적·물적 피해를 발생시킬 수 있다. 따라서 소프트웨어의 신뢰성을 위해 잠재적인 순서 위배 오류를 자율적으로 수리하여 이와 같은 피해를 사전에 방지하는 것이 필요하다.

2.3 관련 연구

항공기 소프트웨어의 건전성 관리시스템에서 순서 위배 오류를 자율적으로 수리하기 위한 연구는 Lee [11]가 있다. 이 연구는 사전 시험 정보를 통해 순서 위배 오류를 진단하는 Anticipating Invariant(AI) [12]를 이용하여 오류를 진단한다. 그리고 오류가 진단된 접근사건을 지연(stall)하는 방식으로 조치한다.

이 연구는 사전 시험 단계에서 프로그램의 수행 중에 모든 동적 명령어(I_x)에 대한 정적 명령어(S_x)인 RPre(I_x)를 수집한다. 해당 정보의 수집 조건은 다음과 같다.

(1) I_x와 같은 메모리 주소로 접근
(2) I_x가 포함되지 않은 다른 스레드의 접근
(3) I_x의 직전 접근

해당 정보가 수집되면, 올바른 수행 시에 같은 정적 명령어에 대한 RPre(I_x)의 집합인 BSet(S_x)를 식별한다. 그리고 프로그램 수행 중의 RPre()와 사전 시

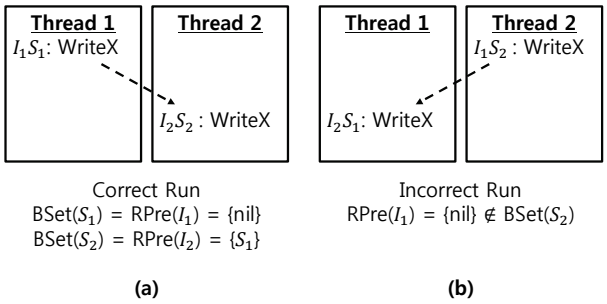

Fig. 2. An Example of Order Violation in Related Work

험에서 수집된 BSet()의 비교를 통해 오류를 진단한다. Fig. 2는 관련 연구의 순서 위배 오류에 대한 예시이다. (a)는 올바른 수행 순서로, I_1과 I_2에 대한 RPre()를 수집하고 각 정적 명령어의 BSet()을 식별하였다. (b)는 프로그램의 수행 순서로, I_1의 RPre()와 S_2의 BSet()이 일치하지 않으므로 순서 위배 오류로 진단한다.

조치는 진단 결과에 따라 세 가지의 경로로 나뉜다. 첫 번째는 순서 위배 오류가 진단되었을 때 해당 접근사건이 포함된 스레드를 멈춘다. 두 번째는 순서 위배 오류의 진단으로 멈춘 접근사건이 정상적으로 수행되어야 할 순서로 진단되면 멈춰있는 스레드를 재개시킨다. 마지막으로 오류가 진단되지 않으면 진단 단계 이후에 특정 조치 없이 해당 접근사건을 정상적으로 수행한다. Fig. 2에서는 (b)에서 오류가 진단된 Thread 2를 멈추고 Thread 1의 WriteX가 수행된 이후에 다시 Thread 2를 재개시켜 올바른 순서대로 수행되도록 한다.

기존 연구는 오류를 진단할 때 모든 접근사건의 수행 직전에 접근사건 순서를 비교한다. 따라서 접근사건 수에 비례하는 시간 오버헤드가 발생한다. 반복적으로 수행되는 항공기 소프트웨어의 특성상 공유변수 접근 횟수도 비행시간이 길어지는 만큼 증가한다. 민항기의 최대 비행시간은 19시간임을 고려해볼 때, 접근사건의 증가는 시간 오버헤드의 증가로 이어진다. 이러한 시간 오버헤드는 항공기 시스템의 실시간성을 만족시키기 어렵다는 문제를 가지고 있다. 따라서 실시간성을 보장해야 하는 항공기 소프트웨어에 기존 연구를 적용하는 것은 부적합하다.

III. 시스템 설계

본 장은 항공기 소프트웨어의 건전성 관리를 위해서 순서 위배 오류를 자율적으로 수리하는 시스템인 On-the-fly Repairing System(ORS)을 설명한다. 그리고 해당 시스템의 내부 모듈을 설명하고 작동 예시를 보인다.

3.1 시스템 구조

Figure 3은 ORS의 전체 구조를 보여준다. ORS는 순서 위배 오류를 진단하기 위해 함수 호출 순서를 사용한다. 그리고 순서 위배 오류가 진단되면 스레드 제어를 통해 순서 위배 오류를 조치한다. 그리고 SIMA[6,11]에서 ARINC 653 Health Monitor(HM)와 Pthread Thread Management(TM)를 통해 오류 및 스레드 재개 시점을 보고 받고 상태에 해당하는 스레드 제어 명령을 수행한다.

ORS는 Diagnosis-Engine(D-Engine)과 Treatment-Engine(T-Engine)으로 구성된다. D-Engine은 함수 수행 정보를 통해 순서 위배 오류를 진단하거나 스레드를 재개하기 위한 시점을 진단한다. 오류나 재개 시점이 진단되면 HM call을 이용하여 SIMA의 HM으로 오류 및 스레드 재개 시점을 보고한다.

ORS는 오류를 진단하기 위해 미리 정의된 함수 호출 순서인 Type State Automaton(TSA)[13]을 사용한다. Fig. 4는 스레드 2개 및 함수 4개를 가지는 프로그램의 TSA의 예제이다. TSA는 특정 상태에서 각 함수가 호출되었을 때 변하는 상태 정보(ex. I → init → U)들을 가진다. 해당 정보들로 함수의 순서를 명시하여 오류 진단 시 사용한다.

T-Engine은 오류 및 재개 시점을 보고받은 HM에 의해 호출되어 스레드 제어 메시지를 SIMA의 TM으로 보낸다. 스레드 제어 메시지는 스레드 대기와 스레드 재개가 있다. 스레드 대기는 오류가 진단되면 순서 위배 오류가 발생한 스레드를 대기시킨다. 스레드 재개는 다른 스레드의 함수 호출이 정상적으로 끝나면 멈춰있는 스레드를 재개시킨다. 마지막으로

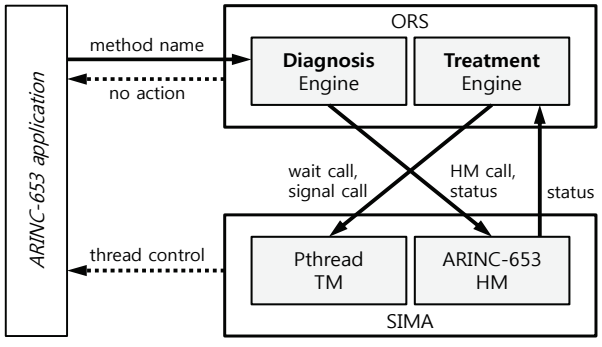

Fig. 3. Overall Architecture of ORS

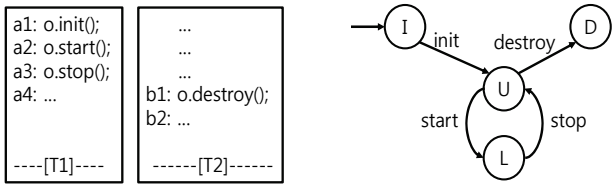

Fig. 4. Examples of Program and TSA

오류가 진단되지 않으면 스레드 제어 메시지는 동작하지 않는다.

ORS의 SIMA는 ARINC 653 HM과 TM으로 구성된다. HM은 D-Engine으로부터 오류 및 스레드 재개 시점을 보고 받으면 T-Engine을 호출한다. TM은 T-Engine의 메시지를 받아 오류가 진단된 스레드를 대기상태로 변경하거나 대기상태의 스레드를 다시 수행시킨다. ORS는 발생한 오류가 다른 소프트웨어에 전파되지 않도록 하여 건전성을 관리하기 위해서 ARINC 653 HM을 사용한다.

3.2 진단 및 조치 엔진

D-Engine은 사용자가 미리 정의한 함수 호출 순서와 수행 정보를 비교하여 순서 위배 오류를 진단한다. Fig. 5는 D-Engine의 알고리즘이다. D-Engine은 수행 중인 함수의 이름을 입력받아 getState()를 호출한다. 해당 함수는 현재 상태와 함수 이름을 매개변수로 가지며, 오류가 없는 경우에는 현재 상태에서 해당 함수가 호출되었을 때 변하는 상태를 반환하고 오류가 있는 경우에는 “bad”를 반환한다. 그리고 두 개의 if문을 통해 오류 및 스레드 재개 시점을 보고한다.

첫 번째 if문은 순서 위배 오류를 진단하는 단계이다. 반환 값이 “bad”로 나오면 HM call을 호출하여 HM에 오류를 보고한다. 오류를 보고받은 HM에 의해 호출된 T-Engine에서 보낸 메시지를 받으면 pthread_cond_wait()을 호출하여 해당 스레드를 대기시킨다. 두 번째 if문은 멈춰있는 스레드의 함수가 올바른 함수의 수행 후에 재개될 시점을 진단하는 단계이다. 반환 값이 “bad”가 아닐 경우에 HM call을 호출하여 HM에 스레드 재개 시점을 보고한다. 재개 시점을 보고받은 HM에 의해 호출된 T-Engine에서 보낸 메시지를 받으면 pthread_cond_signal()을 호출하여 해당 스레드를 재개한다.

```
D-Engine algorithm:
procedure D-Engine(method_name)
  string result_1 = getState(state, method_name)
    if result_1 == “bad” then
      RAISE_APPLICATION_ERROR(order_violation)
      RECEIVE_BUFFER(suspend_buffer)
      pthread_cond_wait()
    end if
  string result_2 = getState(state, method_name)
    if result_2 != “bad” then
      RAISE_APPLICATION_ERROR(trigger)
      RECEIVE_BUFFER(resume_buffer)
      pthread_cond_signal()
    end if
end procedure
```

Fig. 5. The Algorithm of D-Engine

```
T-Engine algorithm:
procedure T-Engine()
  GET_ERROR_STATUS(status)
  string status = status.MESSAGE
  if status == “order_violation” then
    SEND_BUFFER(suspend_buffer)
  end if
  else if status == “trigger” then
    SEND_BUFFER(resume_buffer)
  end if
end procedure
```

Fig. 6. The Algorithm of T-Engine

T-Engine은 오류 및 스레드 재개 시점을 보고받은 HM에 의해 호출되며, 보고 종류에 따라 해당되는 메시지를 TM으로 보낸다. Fig. 6은 T-Engine의 알고리즘이다. T-Engine은 HM에 의해 호출되는 프로세스이며 보고 종류에 대한 정보를 획득하기 위해 HM call인 GET_ERROR_STATUS()를 호출한다. 해당 함수는 보고 정보를 반환한다. 반환 값이 “order_violation”인 경우에는 스레드를 대기시키기 위한 메시지를 전송한다. 반환 값이 “trigger”인 경우에는 스레드를 재개시키기 위한 메시지를 전송한다.

3.3 작동 예시

Figure 7은 Fig. 4의 프로그램에서 발생할 수 있는 함수 호출의 순서 위배 오류를 수리하는 예시이다. Fig. 7의 execution column은 Fig. 4의 프로그램에서 destroy()가 stop()보다 먼저 호출되는 수행 순서를 나타낸다. input column은 해당 시점의 함수 호출 시 반환되는 상태를 나타낸다. output of module column은 해당 시점의 오류 진단 및 스레드 상태를 나타낸다. 표 우측 하단의 TSA는 Fig. 4의 TSA를 각 함수에 대해 정리한 것이다.

A시점은 init()이 수행되기 직전으로 현재 상태 “I”에서 init()을 호출하면 “U”라는 정상적인 상태가 반환되므로 init()의 호출은 오류가 없는 정상적인 순서

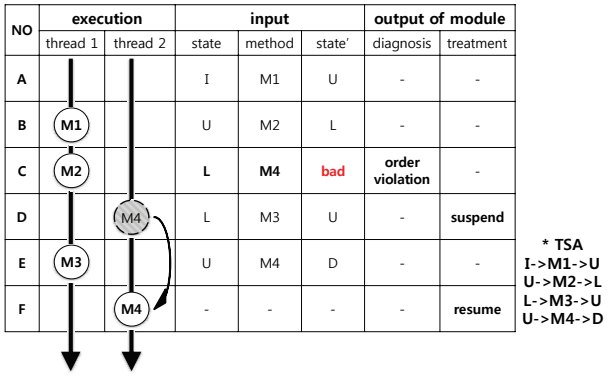

Fig. 7. Repairing Order Violation of Fig. 4

이다. B시점의 start()는 A시점과 동일하게 정상적인 순서로 판단된다. C시점은 destroy()가 호출되기 직전으로 현재 상태 “L”에서 destroy()를 호출하면 “bad”가 반환되므로 순서 위배 오류로 판단된다. 따라서 해당 스레드를 대기상태로 변경한다. D시점의 stop()은 A, B시점과 동일하게 정상적인 순서로 판단된다. E시점은 stop()이 호출된 직후의 시점으로 현재 상태 “U”에서 destroy()를 호출하면 “D”라는 정상적인 상태가 반환되므로 destroy()의 재개 시점으로 판단된다. 그러므로 두 번째 스레드를 재개시켜 destroy()가 호출한다. 결과적으로 네 개의 함수가 올바른 순서로 수행되게 하여 순서 위배 오류를 수리한다.

Ⅳ. 구현 및 실험

본 장은 ORS의 구현 및 실험에 사용된 환경을 운영 환경과 코딩/통합 환경으로 나누어 설명한다. 그리고 사용된 소스 코드와 환경설정을 설명한다. 마지막으로 시스템 작동 시험 및 성능 실험의 과정과 결과를 설명한다.

4.1 시스템 구현

ORS의 구현을 위해 사용된 코딩 및 통합 환경은 다음과 같다. Visual Studio Code 1.37.1 프로그래밍 툴 및 gcc 5.4.0 컴파일러를 사용했다. 그리고 코드 삽입에는 Low Level Virtual Machine(LLVM) 6.0.0을 이용했다. 하드웨어 운영 환경은 Intel Xeon E5-2650 2.30GHz CPU 및 64GB RAM을 사용했다. 그리고 Ubuntu 16.04 LTS-64bit OS에 linux 4.14.139-rt66 kernel 및 Simulated Integrated Modular Avionics (SIMA) 1.3.1.0을 세팅하여 ARINC 653 시뮬레이션 환경을 구성하였다.

시스템의 구현은 TSA, SIMA, Inter Process Communication(IPC) 세 가지 요소가 포함된다. TSA는 테이블 형식의 텍스트 파일로 저장된다. Table 1은 Fig. 4의 TSA를 테이블 형식으로 구현한 예시이다. State column은 현재 상태를 나타내고, init부터 destroy column은 현재 상태에서 각 함수가 호출되었을 때의 상태를 나타낸다. initial_state는 초기 상태(I)를 구별하기 위한 column이다.

Table 1. An Example of TSA

state	init	start	stop	destroy	initial_state
I	U	bad	bad	bad	Y
U	bad	L	bad	D	N
L	bad	bad	U	bad	N
D	bad	bad	bad	bad	N

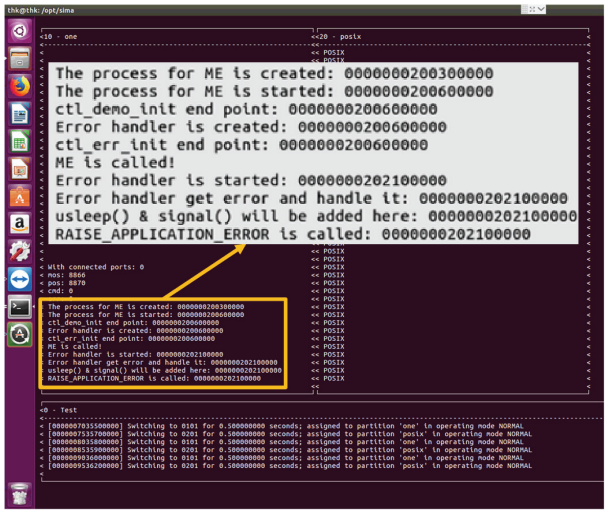

Fig. 8. An Example of SIMA output

SIMA[6,11]는 일반 OS에서 ARINC 653 규격을 만족하는 기능을 이용하기 위해 사용하는 시뮬레이터이다. Fig. 8은 SIMA에서 두 개의 파티션을 가지는 시스템의 출력 화면이다. 파티셔닝 환경을 그래픽으로 출력해주는 Simout 도구를 사용했다. 그림 상단에 두 개의 파티션이 표현되며, 좌측의 첫 번째 파티션에서 대상 프로그램 및 T-Engine을 위한 프로세스가 시작되는 것을 확인할 수 있다.

SIMA는 C언어로 개발되었기 때문에 객체지향 개념을 사용하지 못한다. 따라서 대상 프로그램을 SIMA에 포팅(porting)하지 못하고 IPC로 통신한다. IPC는 프로세스 간 통신을 위해 사용되는 기법이다. IPC는 대상 프로그램에서 오류를 진단했을 때 SIMA로 오류의 진단을 보고하고 스레드 대기/재개 메시지를 받는 역할을 한다. 본 논문에서는 message passing 방식을 사용하였다.

4.2 시스템 시험

시스템 시험은 ORS의 올바른 동작을 확인하기 위해 Fig. 4의 예시 프로그램을 사용하여 수리 실험을 진행했다. Fig. 9는 시험 프로그램의 코드로, shared_variable은 공유변수이다. init()부터 4개의 함수에 공유변수가 포함된 것을 볼 수 있다. t1_func() 및 t2_func()에 4개의 함수가 분배되어 두 개의 스레드로 수행된다.

시험 유형은 4가지 함수를 2개 및 4개의 스레드에 분배하여 호출될 수 있는 모든 순서 조합을 대상으로 하였다. 스레드가 2개인 경우에는 ORS를 적용할 수 없는 유형이 있다. 예를 들면 Fig. 4의 T1에 포함된 3개의 함수 순서가 뒤바뀌는 경우(ex. start() → init() → stop())이다. 해당 유형은 프로그램 코드의 접근사건 순서가 뒤바뀐 것으로, 동시성 오류로 볼 수 없기 때문에 수리대상이 아니다.

```
int Object::shared_variable = 0;

void Object::init()
{…
  shared_variable = 1;
}
…
void *t1_func(void *data)
{…
  pthread_mutex_lock(&sync_mutex);
  Object::init();
  pthread_mutex_unlock(&sync_mutex);
   …
void *t2_func(void *data)
{…
  pthread_mutex_lock(&sync_mutex);
  Object::destroy();
  pthread_mutex_unlock(&sync_mutex);
```

Fig. 9. Source Code for Test & Experimentation

예시 프로그램에서 2개 및 4개의 스레드로 조합할 수 있는 모든 유형은 총 192개이다. 시스템 시험을 진행한 결과, 기법을 적용할 수 없는 유형 및 오류가 없는 유형을 제외한 50개의 유형을 모두 수리할 수 있었다. Fig. 10은 50개 유형 중 하나의 시험 출력 화면이다. init()과 start()는 정상적으로 수행되고, ③의 destroy()가 호출되어 순서 위배 오류로 진단되고 해당 함수의 호출이 지연된다. 결과적으로 ①②④⑤의 순서처럼 올바른 순서로 수행된 것을 볼 수 있다.

```
root@thk:/home/thk/SWHM_implementation/TSA# ./R2
thread1 id: 2633094912
(2633094912) pre_MRC(init)
(2633094912) pre_MRC: Initial state: I
(2633094912) pre_MRC: Current state: U
(2633094912) init() ①
(2633094912) post_MRC()
(2633094912) pre_MRC(start)
(2633094912) pre_MRC: Current state: L
(2633094912) start() ②
(2633094912) post_MRC()
thread2 id: 2624702208
(2624702208) pre_MRC(destroy) ③
(2624702208) pre_MRC: Order violation
(2624702208) pre_MRC: Send a message to SIMA (report)
(2624702208) (2633094912) pre_MRC(stop)
(2633094912) pre_MRC: Current state: U
(2633094912) stop() ④
(2633094912) post_MRC()
(2633094912) post_MRC: Send a message to repairing server (trigger)
(2633094912) pre_MRC: Read Data (0 for suspend): 0
(2624702208) pre_MRC: pthread_cond_wait()
post_MRC: Read Data (2 for resume): 2
(2633094912) post_MRC: pthread_cond_signal()
(2624702208) pre_MRC: Current state: D
(2624702208) destroy() ⑤
(2624702208) post_MRC()
```

Fig. 10. The Result of System Testing

4.3 실험 설계

ORS의 효율성을 검증하기 위해 합성 프로그램에 Repairing-AV 및 ORS를 적용하여 시간 오버헤드를 측정하여 비교한다. 합성 프로그램은 8가지 함수에 대한 순서 위배 오류 및 각 함수에 포함된 공유변수 접근사건의 순서 위배 오류가 발생하는 프로그램이다. 함수의 종류는 오픈소스로 제공되는 autopilot system인 Ardupilot[19]의 함수 구조에서 차용하였다. 해당 프로그램의 전체적인 구조 파악이 힘들기 때문에 실제 함수의 내부 코드를 적용하지 않았고 종류 및 순서만을 차용했다. 합성 프로그램의 코드는 Fig. 9와 동일한 구조를 가진다. 실험에는 init(), stop(), 및 destroy() 이외에 fast_loop(), up-date_flight_mode(), input_euler_angle(), motors_output(), push() 함수가 사용되었다. 각 함수에 포함된 공유변수의 접근사건 개수는 최소 0개부터 최대 80개까지 포함되도록 하였다. 그리고 스레드의 fork/join을 반복하도록 했다.

실험에 사용된 유형은 총 5개로, 합성 프로그램의 스레드 수, 함수 개수, 그리고 함수의 순서 조합에 따라 나뉜다. 각 유형은 스레드 수와 순서 조합에 상관없는 시간 오버헤드를 보이기 위해 사용했다. Table 2는 5가지 유형을 나열한 것이다. 스레드 수는 각 함수가 분배된 스레드의 개수이며, 함수 개수는 각 스레드에 분배된 함수의 개수를 말한다. 마지막으로 순서는 함수 호출의 순서 위배 오류를 유발하기 위해 임의적으로 조정한 순서로, A부터 H까지의 알파벳으로 대체하여 순서를 나열하였다. 위배가 일어나는 순서의 함수를 볼드체로 표시하였다. 해당 합성 프로그램으로 공유변수 접근 횟수의 증가에 따른 시간 오버헤드를 측정하는 실험을 진행했다. 5가지 유형을 대상으로 하였으며, 프로그램 내부 스레드 fork/join의 반복은 4,000번으로 고정하였다. 공유변수 접근 횟수는 최소 5번부터 두 배씩 증가시키며 최대 640번까지 8종류의 개수를 사용하였다.

Table 2. Cases of Synthetic Program

Case	# of threads	# of methods	Order
case 1	2	7:1	ABCDEF**H**G
case 2			ABCDE**H**FG
case 3		4:4	ABC**E**DFGH
case 4	8	1:1:1:1:1:1:1:1	**B**ACDEFGH
case 5			**C**ABDEFGH

4.4 실험 결과 및 분석

Table 3은 Table 2에서 설명한 각 유형의 합성 프로그램에 Repairing-AV와 ORS 기법을 적용하여 측정한 수행 시간을 나타낸다. 공유변수 접근 횟수가 5번이면 총 공유변수 접근 횟수는 20,000회, 640번이면 총 공유변수 접근 횟수가 2,560,000회 발생한다. 그리고 총 함수 호출 횟수는 전체 수행동안 32,000번으로 일정하다.

Repairing-AV의 진단 횟수는 총 공유변수 접근 횟수와 일치한다. Table 3에서 스레드가 2개인 case 1, case 2, case 3에서 Repairing-AV의 수행 시간은 공유변수 접근 횟수에 따라 약 0.3초부터 최대 약 5초까지 증가하였다. 그리고 스레드가 8개인 case 4와 case 5에서 Repairing-AV의 수행 시간은 약 0.9초부터 최대 약 5.5초까지 증가하였다. 따라서 공유변수 접근 횟수가 증가하면 Repairing-AV의 진단 횟수도 같이 증가하기 때문에 프로그램 수행 시간도 급격히 증가함을 확인할 수 있다.

ORS의 진단 횟수는 총 함수 호출 횟수와 일치한다. Table 3에서 스레드가 2개인 case 1, case 2, case 3인 경우 약 0.8초의 일정한 수행 시간을 보였다. 그리고 스레드가 8개인 case 4와 case 5에서 ORS는 약 1.5초의 일정한 수행 시간을 보였다. ORS의 진단 횟수는 총 함수 호출 횟수와 일치하기 때문에 공유변수 접근 횟수가 증가하더라도 총 함수 호출 횟수는 32,000번으로 고정적이다. 따라서 ORS가 적용된 경우 공유변수 접근 횟수에 영향을 받지 않고 일정한 오버헤드로 진단 및 조치가 가능하다.

Figure 11은 Table 3의 데이터 중 case 1에 대한 그래프를 나타낸다. x축은 공유변수 접근 횟수이고 y축은 프로그램 수행 시간이다. Fig. 11에서 Repairing-AV는 총 공유변수 접근 횟수에 비례하기 때문에 수행 시간이 지수적으로 증가하며, ORS는 총 함수 호출 횟수에 비례하기 때문에 수행 시간이 일정한 것을 볼 수 있다. 공유변수 접근 횟수가 5번에서 20번까지인 구간에서는 총 공유변수 접근 횟수의 증가량이 적기 때문에 Repairing-AV의 오버헤드가 낮다. 이

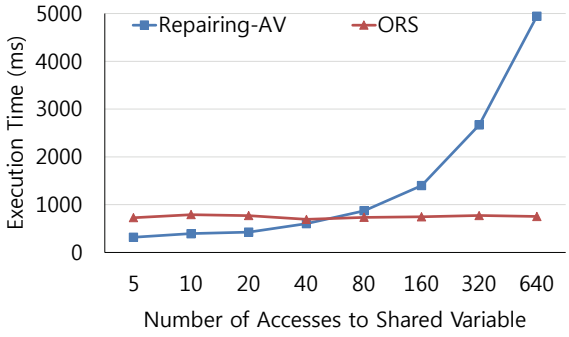

Fig. 11. Execution Time of Case 1

구간에서는 ORS가 약 두 배의 오버헤드를 가지는 것을 볼 수 있다. 그리고 20번일 때부터 Repairing-AV의 수행 시간이 증가하기 시작하여 약 60번일 때 ORS와 수행 시간의 교차가 일어난다. 마지막으로 약 60번에서 640번까지의 구간에서는 Repairing-AV의 수행 시간이 급격하게 증가함을 확인할 수 있다.

실험 결과를 분석해 볼 때, 두 기법의 수행 시간이 일치하는 약 60번의 시점에 Repairing-AV의 진단 횟수는 약 240,000회, ORS의 진단 횟수는 32,000회이다. 따라서 ORS로 진단하는 시간은 Repairing-AV보다 약 8배 높음을 알 수 있다. 따라서 총 공유변수 접근 횟수가 총 함수 호출 횟수의 약 8배보다 낮은 경우에는 Repairing-AV를 사용하는 것이 효율적이며, 높은 경우에는 ORS를 사용하는 것이 순서 위배 오류를 수리하는 것이 더 효율적이다.

V. 결 론

항공기 소프트웨어에서 순서 위배 오류가 발생하기 전에 자율적으로 수리하는 건전성 관리시스템은 개발 단계에서 제거되지 않은 오류를 수행 중에 진단하고 조치하여 소프트웨어에 안전성을 제공하고 유지보수 비용을 경감할 수 있기 때문에 중요하다. 하지만 Repairing-AV는 접근사건 수에 비례하는 시간 오버헤드를 발생시켜 실시간 시스템에 적용하기

Table 3. Execution Time (ms) of Each Case

# of accesses	# of functions	# of total accesses	Case 1		Case 2		Case 3		Case 4		Case 5	
			Repairing-AV	ORS	Repairing-AV	ORS	Repairing-AV	ORS	Repairing-AV	ORS	Repairing-AV	ORS
5	32,000	20,000	315.29	725.31	280.97	648.20	335.69	837.32	944.39	1578.01	952.89	1524.63
10		40,000	391.45	790.26	363.24	825.43	323.37	828.96	1090.82	1544.44	1066.17	1533.89
20		80,000	423.76	770.29	331.15	772.10	442.24	847.63	1023.49	1518.74	1132.85	1507.09
40		160,000	599.11	690.73	667.94	863.94	598.58	962.47	1355.28	1718.09	1406.14	1598.91
80		320,000	871.07	734.91	870.74	791.37	925.02	849.16	1532.08	1496.20	1654.49	1512.87
160		640,000	1397.07	745.71	1457.51	805.47	1449.60	844.60	2220.98	1527.47	2274.79	1536.38
320		1,280,000	2669.01	771.29	2672.45	748.10	2696.24	857.77	3511.05	1526.69	3531.41	1540.23
640		2,560,000	4943.94	750.66	4982.22	776.81	4873.74	812.85	5527.96	1605.81	5637.81	1634.12

어렵게 만든다. 이는 사전 시험 단계에서 수집한 올바른 접근사건의 순서를 기반으로 모든 접근사건에 대한 오류 발생을 진단하기 때문이다.

본 연구는 접근사건이 포함된 함수 간의 순서 위배 오류를 진단하고 해당 함수가 포함된 스레드를 지연시켜 올바른 순서로 수행되도록 조치하는 기법인 ORS를 제시하였다. ORS는 접근사건이 증가하더라도 함수 호출 수에만 비례하는 시간 오버헤드를 가지는 것을 보였다. 특히 접근 횟수가 약 60번 이상일 때 효율적인 것을 확인했다. 향후 다중 공유변수에 대한 연구를 진행하여 다중 공유변수인 경우의 효율성을 분석해야 한다.

후 기

이 연구는 2018년도 경상대학교 연구년제연구교수 연구지원비에 의하여 수행되었음.

References

1) Airlines electronic engineering committee (AEEC), *avionics application software standard interface – ARINC Specification 653 – Part 1. (supplement 2 – required services)*, ARINC Inc, 2015.

2) Merendino, T., Latimer IV, D. T., Hammons, C. B., Falkenthal, D., Capell, P. and Firesmith, D. G., *The Method Framework for Engineering System Architectures*, CRC Press, 2008.

3) Mahadevan, N., Dubey, A. and Karsai, G., "Application of software health management techniques," *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2011, pp. 1~10.

4) Srivastava, A. N. and Schumann, J., "The case for software health management," *2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology*, 2011, pp. 3~9.

5) Koenig, D., "A new software glitch has been found in Boeing's troubled 737 Max jet," Associated Press, June 27, 2019.

6) Ha, O. K., Tchamgoue, G. M., Suh, J. B. and Jun, Y. K., "On-the-fly healing of race conditions in ARINC-653 flight software," *29th Digital Avionics Systems Conference*, 2010, pp. 5.A.6-1~5.A.6.11.

7) Scandura, P. A., Jr., "7. Vehicle health management systems," *Digital avionics handbook*, CRC Press, 2015.

8) Netzer, R. H. and Miller, B. P., "What Are Race Conditions?," *ACM Letters on Programming Languages and Systems (LOPLAS)*, Vol. 1, No. 1, 1992, pp. 74~88.

9) Lucia, B. and Ceze, L., "Cooperative empirical failure avoidance for multithreaded programs," *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, Vol. 48, No. 4, 2013, pp. 39~50.

10) Lu, S., Park, S., Seo, E. and Zhou, Y., "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329~339.

11) Choi, E. T., Lee, D. S., Jun, Y. K., and Lee, S. J., "On-the-fly Atomicity Violation Repairing Technique for Airborne Health Management Systems," *Journal of The Korean Society for Aeronautical and Space Sciences*, Vol. 48, No. 7, 2020, pp. 547~554.

12) Zhang, M., Wu, Y., Lu, S., Qi, S., Ren, J. and Zheng, W., "A lightweight system for detecting and tolerating concurrency bugs," *in IEEE Transactions on Software Engineering*, Vol. 42, No. 10, 2016, pp. 899~917.

13) Zhang, L. and Wang, C., "Runtime prevention of concurrency related type-state violations in multi-threaded applications," *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014. pp. 1~12.

14) Sidiroglou, S., Laadan, O., Perez, C. R., Viennot, N., Nieh, J. and Keromytis, A. D., "Assure: automatic software self-healing using rescue points," *ACM SIGARCH Computer Architecture News*, Vol. 37, No. 1, 2009, pp. 37~48.

15) Zhang, W., Kruijf, M. D., Li, A., Lu, S. and Sankaralingam, K., "ConAir: featherweight concurrency bug recovery via single-threaded idempotent execution," *In Proceedings of the eighteenth international conference on Architectural support for programming languages an operating systems*, 2013, pp. 113~126.

16) Zhou, B., Neamtiu, I. and Gupta, R., "Predicting concurrency bugs: how many, what kind and where are they?," *In Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, 2015, pp. 1~10.

17) Jackson, J., "Nasdaq's Facebook glitch came from 'race conditions'," Computerworld, May 21, 2013.

18) United State Department of Defense, "Appendix E. Generic Software Safety Requirements and Guidelines," *Joint Software Systems Safety Engineering Handbook*, August 2010, pp. E-15~E-18.

19) Luo, Z., Xiang, X. and Zhang, Q., "Autopilot system of remotely operated vehicle based on Ardupilot," *Intelligent Robotics and Applications*, 2019. pp. 206~217.