Research Paper
Journal of Aerospace System Engineering
Vol.16, No.2, pp.13-24 (2022)

EISSN 2508-7150
http://dx.doi.org/10.20910/JASE.2022.16.2.13

항공용 객체지향 소프트웨어에 대한 취약점 검증 방안

장정훈[1],†· 김성수[1]· 이지현[1]

[1]㈜모아소프트

Verification Methods for Vulnerabilities of Airborne Object-Oriented Software

Jeong-hoon Jang[1],†, Sung-su Kim[1] and Ji-hyun Lee[1]

[1]MOASOFT Corp.

Abstract

As the scale of airborne system software increases, the use of OOT (Object-Oriented Technology) is increasing for functional expansion, efficient development, and code reuse, but the verification method for airborne object-oriented software is conducted from the perspective of the existing procedure-oriented program. The purpose of this paper was to analyze the characteristics of OOT and the vulnerabilities derived from the functional characteristics of OOT, and present a verification method applicable to each software development process (Design, Coding and Testing) to ensure the functional safety integrity of aviation software to which OOT is applied. Additionally, we analyzed the meaning of the static analysis results among the step-by-step verification measures proposed by applying LDRA, a static analysis automation tool, to PX4, an open source used to implement flight control software.

초 록

항공용 소프트웨어의 규모가 커짐에 따라 기능적 확장, 효율적인 개발 및 코드의 재사용을 위하여 객체지향 기술의 사용이 증가하고 있으나, 그 검증방안은 기존의 절차지향 프로그램 관점으로 수행되고 있다. 본 논문에서는 객체지향 기술의 특징과 객체지향 언어의 기능적 특징에서 파생되는 취약점들을 분석하고 객체지향기술이 적용된 항공용 소프트웨어의 기능 안전 무결성을 보장하기 위한 소프트웨어 개발단계(Design, Coding, Test)별 적용 가능한 검증 방안을 제시한다. 또한, 비행제어 소프트웨어 구현에 사용되는 오픈소스인 PX4에 정적분석 자동화 도구인 LDRA를 적용하여 제시한 단계별 검증 방안 중 정적분석 결과의 의미를 분석하였다.

Key Words : OOT(객체지향기술), MISRA(자동차 산업 소프트웨어 신뢰성 협회), LDRA(리버풀 데이터 리서치 협회), Vulnerability(취약점), Safety(안전성), Verification(검증)

Ⅰ. 서 론

항공, 자동차, 철도 등 산업전반에서 소프트웨어의 요구사항이 늘어나고 기능이 복잡해짐에 따라 규모가 큰 프로젝트에서는 효율적인 개발을 위해 객체지향 기술(OOT, Object-Oriented Technology)의 수요가 증가하고 있다. OOT를 적용하게 되면, 많은 유용한 도구 사용, 코드의 재사용성, 효율적인 소프트웨어 설계 및 개발, 비용 절감, 품질 향상, 시간 단축 등의 이점을 가지고 있다.

그러나, 지금까지 항공 컴퓨터 시스템에 적용한 OOT는 그리 많지 않은데, 그 이유는 안전성이 입증된 기술을 사용하여야 하기 때문이다. OOT가 비용 효율적인 것으로 입증되었으므로 기술적으로도 안전성이 보증되면 안전이 중요한 항공 시스템에서도 많이 사용

Received: Oct. 28, 2021 Revised: Apr. 05, 2022 Accepted: Apr. 06, 2022

† Corresponding Author
Tel: *** - **** - **** E-mail: jhjang@moasoftware.co.kr
© The Society for Aerospace System Engineering

할 수 있다.

OOT를 사용할 때 신중하게 고려해야 하는 몇 가지 사항이 있다. 객체지향 언어 (Object-Oriented Language)는 한 소프트웨어 내에서 하위 클래스들이 상위 클래스의 속성 및 메서드를 재사용하고 새로운 소프트웨어를 개발 시 기존 소프트웨어의 클래스를 상속받아 재사용할 수 있어서 생산성이 높다. 기존의 기능을 수정하거나 새로운 기능을 추가할 때에도 유지보수가 용이하다. 기존 기능을 수정할 때 기존 메서드를 캡슐화하여 내부 정보를 분리해 주변의 다른 객체들에 미치는 영향을 최소화한다. 새로운 기능을 추가할 때는 상속을 통해 기존의 기능을 활용하고 새로운 속성만 추가하면 되기 때문에 효율적이다. 이외에도 요구사항 변화에 대해 안정된 소프트웨어 구조를 가지고 있어 빈번히 발생하는 요구사항 변화에 신속하게 대처할 수 있다.

객체지향 기술은 상속과 추상화 개념에서 파생되는 함수 Overloading, Overriding 특징을 포함하여 Template, Garbage Collector, Virtualization 특징을 활용하여 이미 개발되어 있는 코드를 재활용함으로써 신속한 개발을 가능하게 하지만 많은 부분들이 컴파일러나 가상머신에 의해 실행 코드가 결정되므로 충분한 검증 활동을 수행하지 않을 경우 개발자가 의도하지 않는 동작을 일으킬 수 있는 구조적인 위험성이 잠재되어 있다.

이런 객체지향 언어의 기능적 특성으로 인하여 절차지향 언어에 비해 상대적으로 소프트웨어가 불안전하고, 여러가지 측면에서 취약점(Vulnerability)을 가지고 있다. 항공용 안전 소프트웨어 개발에 대한 국제 규격인 DO-178C의 부속서인 DO-332에서는 OOT&RT (Object-Oriented Technology and Related Techniques)의 기능적 특징 및 특징에서 파생되는 7가지 취약점(Inheritance, Parametric Polymorphism, Overloading, Type Conversion, Exception Management, Dynamic Memory Management, Virtualization)에 대하여 설명하고 있다[1,2,3,4].

본 연구에서는 이러한 OOT&RT의 취약점에 대한 검증방안과 개선방안을 연구한다. 취약점으로 인해 발생하는 결함을 미리 예상한 후, 소프트웨어 구조 설계를 검토하고 테스트 자동화 도구를 활용하여 소스코드에 대한 정적분석(Static Analysis) 및 요구사항 기반 시험(Requirements-Based Test), 구조적 커버리지 분석(Structural Coverage Analysis)을 통해 예상한 결함을 확인한다. 검증 대상 소스코드는 전 세계적으로 가장 많이 사용되고 있는 PX4 오픈소스를 사용하였다. 테스트 자동화 도구로 사용할 LDRA 도구는 항공, 국방, 자동차 등 여러 산업분야에서 소프트웨어 안전성 보장을 위해 적용하고 있는 코딩 규칙 및 동적시험 기능을 지원한다[5,6]. OOT&RT의 기능적 취약점에 대한 분석 결과를 바탕으로 OOT&RT로 개발된 소프트웨어에 대한 취약점을 분석하고 개선방안을 제시한다.

II. 본 론

2.1 DO-332 개요

항공기는 한번의 중대사고에 따른 피해 손실이 매우 크고 운용 중에 수리가 불가능하다. 항공기 소프트웨어의 비중은 점점 더 커지고 있어 항공기 운용 중에 발생할 수 있는 위험요소를 사전에 식별하고 점검하여 손실을 방지하고자 FAA(Federal Aviation Administration, 미연방항공국)에서는 DO-178C 등 항공 소프트웨어 승인 규격에 따라 소프트웨어를 개발하도록 하고 있다[1].

소프트웨어 발전에 따라 다양한 개발방법 및 언어를 사용하여 개발하는 경우가 증가하고 있어 RTCA에서는 이를 고려하여 C++, Java와 같은 OOT에 적용할 수 있는 DO-332 표준을 2012년에 발표했다[2].

DO-332는 RTCA에서 발행한 DO-178C와 DO-278A에 대한 부속서로서 OOT&RT를 다루고 있다. DO-332에서는 항공기 시스템 및 장비에 탑재되는 소프트웨어 개발에 OOT&RT와 객체지향 언어를 사용할 때 항공기 시스템 및 장비의 안전 수준(Safey Level)을 보증하기 위한 소프트웨어 개발 및 검증 기준을 제공한다.

2.2 DO-332 Objective

DO-332는 DO-178C의 부속서이기에 전체적인 기본 구조와 내용은 DO-178C와 크게 다른 내용이 없으며, Table 1과 같이 OOT와 관련된 2개의 Objective가 추가되었다.

Table 1 DO-332 Table OO.A-7 Verification of Verification Process Results

	Objective	SW Level			
	Description	A	B	C	D
OO10	Verify local type consistency.	●	●	○	
OO11	Verify the use of dynamic memory management is robust.	●	●	○	
Legend: ●: The objective should be satisfied with independence. ○: The objective should be satisfied.					

2.2.1 [OO.A-7.OO10] Verify local type consistency.

이 Objective는 로컬 유형의 일관성을 검증하는 것으로 클래스 계층 구조에 대해 Local Type Consistency를 고려하여 설계되었는지 검증하여야 한다는 내용이다.

검증하는 방법에는 클래스의 구조에 따라 상위 클래스에서 생성한 테스트케이스를 대체가능한 하위 클래스에 적용하여 의도한 기능을 수행하는지 확인하는 방법(Unit Test), 테스트 가능한 모든 상황을 고려한 시나리오를 만들어 테스트하는 방법(Requirements-Based Test)이 있으며 주로 동적시험을 통해 검증한다.

SW Level에 따라 A, B인 경우에는 별도의 검증기관에서 해당 활동을 수행하고 C인 경우에는 개발 조직 자체의 활동으로 대체 가능하다. D인 경우에는 해당 활동을 생략할 수 있다.

2.2.2 [OO.A-7.OO11] Verify the use of dynamic memory management is robust.

이 Objective는 동적 메모리 관리의 사용이 올바르게 이루어지는지 검증하는 것이다. 메모리 참조의 모호성, 메모리 부족, 메모리 동시할당 등 동적 메모리 관리를 견고하게 처리하고 있는지 검증하여야 한다.

검증 방법으로는 메모리 누수가 없도록 더 이상 참조되지 않는 동적 할당된 메모리를 적절히 반환하는지, 동적 할당 요청이 성공적으로 수행될 수 있도록 적절한 문법을 사용하였는지 등의 여부를 정적분석을 통해 가능하다.

SW Level에 따라 A, B인 경우에는 별도의 검증기관에서 해당 활동을 수행하고 C인 경우에는 개발 조직 자체의 활동으로 대체 가능하다.

2.3 OOT 취약점 분석

앞서 살펴본 내용과 같이 DO-332에서는 DO-178C에서 2개의 OOT 관련된 Objective만 추가되었지만 해당 항목만을 고려할 것을 의미하지는 않는다.

Table 2는 인증과정에서 DER이 참고하는 체크리스트이다. CAST 문서에서는 DO-332에 기술된 OOT&RT 관련 취약점들을 꼼꼼히 점검할 것을 권고하고 있다.[3]

각 활동들이 수행되었는지 확인하기 위해서는 소프트웨어 각 개발단계(Design, Coding, Test)에 적용 가능한 검증 방안을 계획하고 수행해야 한다.

Table 2 SW Review Checklist for OOT

Item #	Evaluation Activity/Question
1	다음과 같은 일반적인 객체지향 개발 문제를 어떻게 해결할 것인지 계획하였는가? - Inheritance, Parametric Polymorphism, Overloading, Type Conversion, Exception Management, Dynamic Memory Management, Virtualization - Traceability - Structural coverage
2	개발 표준서(요구사항, 디자인, 코딩 표준서)가 객체 지향 구현에 대한 제한 사항(예: 대상 언어에서 지원하지 않는 규칙 처리)을 해결하는가? - OO methods and notations - Restrictions for the chosen OO language
3	요구사항, 디자인, 코드 및 테스트 케이스/절차 간의 추적성은 어떻게 설정되고 유지되는가?
4	추적성 접근 방식이 일반적인 OO 추적성 문제를 해결하는가? - Traceability of functional requirements through implementation regarding to mismatches between function-oriented requirements and an object-oriented implementation - Complexity of class hierarchies - Tracing through OO views, behavior of the classes and interaction together to provide the required function invisible in any single view
5	신청자 대체 가능성을 어떻게 해결하였는가? 예를 들어, 리스코프 치환 원칙(LSP)을 적용하여 구현하였는가?
6	다중 상속이 사용되는 경우, 하위클래스 작업의 의도가 명확하고 모호하지는 않는가?
7	오버라이딩 문제를 다루고 있는가? - State Defined Incorrectly (SDI) problem - Anomalous Construction Behavior (ACB1) problem - Anomalous Construction Behavior (ACB2) problem - State Definition Anomaly (SDA) problem - State Visibility Anomaly (SVA) problem
8	개체 또는 디자인 구성요소를 재사용하는 경우, dead/deactived code문제를 다루었는가?

DO-332의 부록 OO.D에서는 OOT&RT의 7 가지 주요 특징과 OOT&RT 사용과 관련된 취약점 분석을 제공하고 있으며, Table 3과 같다.

Table 3 DO-332 Annex OO.D Vulnerability Analysis

Section	Feature
OO.D.1.1	Inheritance
OO.D.1.1.1	- substitutability - method implementation inheritance - unused code - dispatching - multiple inheritance
OO.D.1.2	Parametric Polymorphism
OO.D.1.2.1	- macro expansion
	- code sharing
OO.D.1.3	Overloading
OO.D.1.3.1	- overloading ambiguity
	- implicit type conversion
	- coincidental name collisions
OO.D.1.4	Type Conversion
OO.D.1.4.1	- narrowing type conversion
	- downcasting
OO.D.1.5	Exception Management
OO.D.1.5.1	- no action
	- inappropriate actions
OO.D.1.6	Dynamic Memory Management
OO.D.1.6.1	- ambiguous references
	- fragmentation starvation
	- deallocation starvation
	- heap memory exhaustion
	- premature deallocation
	- lost update and stale reference
	- time-bound allocation or deallocation
OO.D.1.7	Virtualization
OO.D.1.7.1	- interpreter

2.3.1 Inheritance(상속)

상속은 OOT의 핵심적인 기술이며, 이는 코드의 재사용을 위한 특화된 기능을 제공한다. 상속에 있어서 고려해야 할 5가지 취약점은 다음과 같다.

2.3.1.1 Substitutability(치환성)

LSP(리스코프 치환 원칙, Liskov Substitution Principle)를 준수하여야 한다[7].

LSP: type S가 type T의 subtype일 때, f(x)가 type T의 객체 x에 대한 검증가능한 속성이면 type S의 객체 y에 대한 f(y)도 참이어야 한다. 즉 type T의 subtype인 type S가 있는 경우, 프로그램 내에서 T type의 객체는 특성의 변경 없이 S type의 객체로 지환 가능해야 한다.

LSP를 준수하기 위해서는 다음의 3가지 사항을 만족하여야 한다.

- 하위유형에서 선행조건은 강화될 수 없다.
- 하위유형에서 후행조건은 약화될 수 없다.
- 하위유형에서 상위유형의 불변조건은 반드시 유지되어야 한다.

슈퍼클래스를 상속하여 생성된 하위클래스가 적절한 하위유형이 아닌 경우에 클래스 대체로 인해 응용 프로그램이 잘못 동작할 수 있다. 이는 슈퍼클래스에서의 정의된 메서드를 기능적으로 호환되지 않게 하위클래스에서 재정의할 때 발생할 수 있다.

2.3.1.2 Method implementation inheritance (메서드 구현 상속)

메서드 구현 상속 시, 하위클래스에 추가된 속성이 있고 그 속성을 업데이트해야 하는 메서드가 재정의되지 않은 경우에 문제가 될 수 있다. 해당 속성의 값이 상속된 메서드 실행 결과에 따라 달라지는 경우 업데이트되지 않을 수 있어서 상속된 메서드를 호출하면 객체 상태가 잘못되고 예기치 않은 동작이 발생할 수 있다.

2.3.1.3 Unused code (사용되지 않는 코드)

응용 프로그램에서 클래스의 메서드가 호출되지 않은 경우에 데드코드(Dead code) 또는 비활성화된 코드(Deactivated code)가 발생할 수 있다. 슈퍼클래스의 메서드가 하위클래스에 의해 재정의되고 슈퍼클래스가 인스턴스화되지 않는 경우에 슈퍼클래스에서 코드가 비활성화될 수 있다. 슈퍼클래스에서 메서드를 제거하면 슈퍼클래스의 무결성이 깨질 수 있다. 이러한 경우 비활성화된 메서드를 간과할런지도 모른다. 재사용 가능한 소스코드는 동일한 소스코드를 여러번 사용하도록 설계되었기 때문에 추적하기 어려우며, 재사용 가능한 소스코드에는 응용 프로그램에서 사용되지 않는 기능이 있을 수 있다.

2.3.1.4 Dispatching (디스패칭)

디스패치를 사용하면 슈퍼클래스로 유형이 선언되었지만 실제 유형이 하위클래스인 객체는 재정의된 메서드를 호출할 때 하위클래스의 메서드 대신에 슈퍼클래스의 메서드가 호출되어 예기치 않은 결과를 유발할 수 있다.

2.3.1.5 Multiple inheritance (다중 상속)

다중 상속에는 다중 인터페이스 상속과 다중 구현 상속이 있다. 다중 인터페이스 상속의 경우는 두 개 이상의 슈퍼클래스에서 메서드를 상속한 경우이며, 메서드 간의 불일치 또는 호환되지 않아서 그 결과 혼란과 예기치 않은 동작을 유발할 수 있다. 또한 다중 구현 상속에서는 메서드 상속과 다중 인터페이스 상속이 결합되어 잘못된 구현이 상속되거나 메서드의 속성이 여러 경로를 통해 상속되는 경우에 예기치 않은 동작이 발생할 수 있다.

2.3.2 Parametric polymorphism (파라메트릭 다형성)

파라메트릭 다형성(Parametric polymorphism)은 메서드나 클래스를 개별적으로 다시 작성하지 않아도 한 유형의 참조 변수로 여러 유형의 객체를 참조할 수 있도록 한 것이다. 다형성을 사용하면 유연하게 코딩할 수 있고, 코드의 재사용성이 보장되어 유지보수가 수월해진다. 하지만 소스코드와 컴파일된 오브젝트코드 간에 추적하기가 어렵다. 같은 객체를 각각 다른 메서드로 인스턴스화했을 경우 인스턴스화된 객체는 서로 다를 수 있기 때문에 주의해야 한다.

2.3.3 Overloading (오버로딩)

오버로딩(Overloading)은 기본적으로 메서드의 명칭을 동일하게 하고, 반환형이나 인자의 유형 또는 명칭을 다르게 한다. 오버로딩은 코드 개발 시 구조 설계, 가독성, 유지보수에 대한 도움을 줄 수 있다. 그러나, 컴파일러에서 최적의 묵시적인 유형 변환 시, 전혀 의도되지 않은 형태로 메서드가 호출이 될 수 있는 취약점을 가질 수 있다. 또한, 여러 개발자가 협업을 할 때, 다른 동작을 하는 하위 프로그램에서 우연히 동일한 명칭을 사용할 경우, 의도치 않은 혼동이 발생한다. 오버라이딩(Overriding, 재정의)과 관련된 5 가지 취약점이 있다.

- SDI(State Defined Incorrectly): 재정의 메서드에 의해 수행된 계산이 변수에 대한 재정의 메서드의 계산과 의미상 동일하지 않은 경우
- ACB1(Anomalous Construction Behavior): 하위 클래스가 하위클래스의 상태 공간에 정의된 변수를 사용하는 메서드의 재정의 정의를 제공하는 경우
- ACB2(Anomalous Construction Behavior): 하위 클래스가 슈퍼클래스의 상태 공간에 정의된 변수를 사용하는 메서드의 재정의 정의를 제공하는 경우
- SDA(State Definition Anomaly): 재정의된 메서드의 정의와 일치하는 상속된 상태 변수에 대한 정의를 재정의 메서드가 제공하지 않는 경우
- SVA(State Visibility Anomaly): private state 변수가 존재할 때, 하위클래스의 모든 재정의 메서드가 슈퍼클래스의 재정의 메서드를 호출하지 않는 경우

2.3.4 Type Conversion (유형 변환)

유형 변환(Type Conversion)은 개발자에게 편의성을 제공해 줄 수 있으나, 축소 유형 변환(narrowing type conversion)으로 인한 데이터 손실, 하향 변환(downcasting)으로 객체 및 주변 데이터를 손상시킬 수 있으며, 이러한 유형 변환으로 인하여 프로그램 실행 시 런타임 에러가 발생할 수 있다.

2.3.5 Exception Management (예외 관리)

메서드 내에서 예외(Exception)가 발생하고 호출 메서드에서 예외를 처리하는 것은 객체지향 언어의 일반적인 기능이다. 객체지향 언어에는 시스템 고장(System Fault), 소프트웨어 고장(Software Fault)과 같은 예상할 수 있는 예외 또는 오류(Error)에 대해 처리(Handling) 할 수 있는 클래스가 존재하는데, 개발자가 명시적으로 예외 처리 관련 클래스를 상속받아 예외를 처리하게 되면 개발자의 의도와는 다르게 처리될 수 있는 취약점이 존재한다.

2.3.6 Dynamic Memory Management (동적 메모리 관리)

객체지향 프로그램은 가상머신이 해당 응용 프로그램이 실행되면 시스템으로부터 프로그램을 수행하는데 필요한 메모리를 할당받아 용도에 따라 static, stack, heap 등 여러 영역으로 나누어 관리한다. 하지만 복잡한 구조를 가진 프로그램은 객체를 동적으로 할당하고 해제하는 절차를 짧고 많이 반복하기 때문에 아래와 같은 취약점들이 나타날 수 있다.

1) 모호한 메모리 참조
2) 해제되지 않은 인접한 메모리 참조
3) 잘못된 메모리 영역에 메모리 할당
4) 동시 할당 시 충돌 혹은 메모리 부족 문제
5) 메모리 조각 모음 시, 할당된 메모리 영역 해제
6) 잘못된 영역의 메모리 업데이트
7) 타이밍이 맞지 않는 메모리 할당 및 해제

2.3.7 Virtualization (가상화)

가상화(Virtualization)는 실행환경을 다른 환경으로 에뮬레이션하는 기술이다. 인터프리터(Interpreter)와 같은 가상화 기술은 개발한 소프트웨어를 다른 환경에서 동작할 수 있게 한다. 이런 가상화 기술은 실행코드로 검증해야 하는데, 프로그래밍 명령을 데이터로 잘못 분류함으로써 결과적으로 요구사항이 부적절

하거나 추적성이 깨지게 되어 검증이 불충분할 수 있다.

2.4 OOT 취약점 검증방안

OOT의 7가지 취약점(OO.D.1.1~1.7)에 대하여 4가지 검증방법은 Table 4와 같다.

Table 4 Verification Methods for OOT Vulnerabilities

<table>
<tr><th rowspan="2">DO-332 Section</th><th rowspan="2" colspan="2">OOT Feature</th><th colspan="4">Verification Method</th></tr>
<tr><th>Decide Review</th><th>Code Review</th><th>R-B Test</th><th>Coverage Analysis</th></tr>
<tr><td rowspan="5">OO.D.1.1</td><td rowspan="5">Inheritance</td><td>Substitutability</td><td>O</td><td>-</td><td>O</td><td>-</td></tr>
<tr><td>Method Implementation inheritance</td><td>O</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>Unused code</td><td>-</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>Dispatching</td><td>-</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>Multiple inheritance</td><td>O</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>OO.D.1.2</td><td colspan="2">Parametric Polymorphism</td><td>-</td><td>O</td><td>O</td><td>-</td></tr>
<tr><td>OO.D.1.3</td><td colspan="2">Overloading</td><td>O</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td rowspan="2">OO.D.1.4</td><td rowspan="2">Type Conversion</td><td>Downcast</td><td>-</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>Narrow type conversion</td><td>-</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>OO.D.1.5</td><td colspan="2">Exception Management</td><td>O</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>OO.D.1.6</td><td colspan="2">Dynamic Memory Management</td><td>-</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>OO.D.1.7</td><td colspan="2">Virtualization</td><td>-</td><td>O</td><td>O</td><td>O</td></tr>
</table>

2.4.1 SW 개발 단계별 검증방법

일반적으로 Safety Critical SW 개발 프로세스는 Requirement, Design, Coding, Testing 단계로 표현되는 V모델 프로세스를 사용한다.

Requirement 단계에서는 DO-178C에서 다루는 내용과 유사하다. Design 단계에서는 설계 검토(Design Review), Coding 단계에서는 코드 검토(Code Review), Testing 단계에서는 요구사항 기반 시험(Requirements-Based Test), 구조적 커버리지 분석(Structural Coverage Analysis)을 수행하여 소프트웨어의 무결성을 확보해야 한다.

2.4.1.1 Design Review (설계 검토)

OOT 기술 및 기법을 적용하여 SW를 개발하기 위하여는 SW요구사항을 정의하고 SW구조 및 모듈을 설계한다. 리스코프 치환 원칙(LSP)을 적용하여 서브모듈로의 변경, 적용시에도 무결하도록 설계해야 한다.

2.4.1.2 Code Review (코드 검토)

소스코드에 대한 검토에서는 구현된 소스코드에 대하여 미리 정의한 코딩 표준에 대한 준수 여부를 검사한다. 코딩 표준은 국제적으로 가장 많이 활용하고 있는 MISRA-C++:2008에서 제시하는 코딩표준을 적용하여 준수/위반 사항을 검사한다[8]. 코딩표준 검사 도구는 LDRA를 사용하며, LDRA 도구는 MISRA-C++ 코딩표준뿐만 아니라 OOT 취약점을 발견하기 위한 추가적인 룰을 지원한다[9].

OOT 특징별 검증 방안을 다루고 있는 2.4.2절에서 각각의 OOT 취약점을 검증할 수 있는 코딩표준을 LDRA Rule로 제시한다. OOT 특징별 취약점 검사를 위한 각각의 코딩표준을 취합하여 소스코드에 대한 코드검토를 수행함으로써 통합적인 OOT 취약점 검증을 수행하게 된다.

2.4.1.3 Requirements-Based Test (요구사항 기반 시험)

OOT 적용 시, 모든 기능은 메서드로 구현되므로 요구사항을 추적하여 구현된 메서드에 대한 시험을 수행하여 요구사항 충족을 확인한다. 클래스는 요구사항을 구현하기 위한 아키텍처이므로 하위클래스에 구현된 메서드를 추적하여 재정의된 하위클래스의 메서드를 추적하고 요구사항 기반 시험을 수행하여야 한다.

2.4.1.4 Structural Coverage Analysis (구조적 커버리지 분석)

구조적 커버리지 분석의 목적은 요구사항 기반 시험을 수행한 후에 코드 구조가 해당 소프트웨어 레벨에 필요한 정도로 검증되었다는 증거를 제공하고 의도하지 않은 기능이 없음을 입증하는 수단을 제공하는 것이다. 구조적 커버리지 분석에서는 다음과 같은 커버리지가 100% 달성되었는지를 확인한다.

- 문장 커버리지(Statement coverage): 구현된 소스코드의 문장을 실행한 정도
- 분기/결정 커버리지(Branch/Decision coverage): 분기/결정이 발생하는 분기를 실행한 정도
- 수정조건/결정 커버리지(Modified condition/decision coverage): 다중조건에 대하여 결정이 발생한 정도
- 소스코드와 객체코드 간의 추적성(Source code to object code traceability)
- 데이터 커플링 커버리지(Data coupling coverage): 데이터 또는 변수의 흐름을 실행한 정도
- 컨트롤 커플링 커버리지(Control coupling

coverage): 단위함수를 호출한 정도

DO-332 Table OO.A-7에서 제시하고 있는 소프트웨어 레벨별 달성하여야 하는 구조적 커버리지는 Table 5와 같다.

Table 5 Structural coverage objectives for software level

Table OO.A-7		Software Level			
Objective		A	B	C	D
5	Test coverage of software structure (modified condition/decision) is achieved.	●			
6	Test coverage of software structure (decision coverage) is achieved.	●	●		
7	Test coverage of software structure (statement coverage) is achieved.	●	●	○	
8	Test coverage of software structure (data coupling and control coupling) is achieved.	●	●	○	

2.4.2 OOT 특징별 취약점 검증방안

2.4.2.1 Inheritance 검증방안

치환성(Substitutability)를 검증하기 위해서는 가장 먼저 설계 검토를 수행한다. LDRA 도구를 활용하여 클래스계층도(Class Hierarchy Diagram) 및 메서드 구조도를 생성하여 클래스 설계를 검토한다. 요구사항 기반 시험을 통해 하위클래스와 슈퍼클래스를 대체할 수 있는 모든 클래스 타입에 대하여 동일한 테스트 케이스로 시험을 수행하여, Pass/Fail 여부로 검증한다. Fig. 1은 치환성에 대한 취약점을 검증하기 위한 테스트 케이스이다.

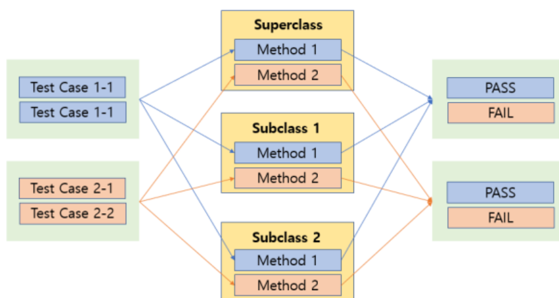

Fig. 1 Test Cases for Verifying Substitutability

메서드 구현 상속(Method implementation inheritance)은 설계 검토를 통해 하위클래스에서 추가된 속성이 있는지 식별한 후 해당 속성에 영향성이 있는 메서드 구현에 이상이 없는지 설계 검토를 수행한다. 코드 검토를 수행 상속 오류를 검토한다. 또한 요구사항 기반 시험을 수행 및 데이터 커플링 커버리지(Data coupling coverage), 컨트롤 커플링 커버리지(Control coupling coverage)를 분석하여 검증한다.

사용되지 않는 코드(Unused code)는 코드 검토에서 구조적인 데드 코드(Dead code)를 식별하고, 모든 동적시험을 수행하여 문장 커버리지(Statement coverage)를 분석하여 비활성화 메서드(Deactivated method)를 검토한다.

디스패칭(Dispatching)을 검증하기 위해서는 먼저 LDRA 도구를 활용하여 소스코드에 대한 정적분석을 수행하여 디스패치와 관련된 코딩표준을 검사한다. 또한 요구사항 기반 시험를 수행하고 컨트롤 커플링 커버리지(Control coupling coverage)를 분석하여 검증한다.

다중 상속(Multiple inheritance)은 설계 검토를 통해 클래스 간의 다중 상속 오류가 없는지 검증한다. 코드 검토를 통해 문법적 오류를 검토하고 요구사항 기반 시험 및 구조적 커버리지 분석을 수행하여 검증한다.

상속성에 대한 코드 검토 시 적용하는 LDRA Rule은 Table 6과 같다.

Table 6 LDRA Rules for Inheritance

LDRA Rule	Rule Description	MISRA-C++:2008
28 S	Duplicated base classes in a derived class.	-
205 S	Use of multiple inheritance.	-
262 S	Non virtual function redefined.	-
327 S	Reuse of struct or class member name.	-
595 S	Base member with change of access.	-
70 D	DU(Defined, Unused) anomaly, variable value is not used.	0-1-6 0-1-9
76 D	Procedure is not called or referenced in code analysed.	0-1-10

2.4.2.2 Parametric Polymorphism 검증방안

LDRA 도구를 활용하여 소스코드에 대한 정적분석을 수행하여 파라메트릭 다형성과 관련된 코딩표준을 검사하고, 적용 가능한 모든 유형에 대한 테스트케이스를 작성하여 요구사항 기반 시험을 수행하여 확인

한다. 또한 실행 가능한 모든 type의 Data를 적용하여 기능이 올바르게 동작하는지 여부를 검토하여 검증한다.

파라메트릭 다형성(Parametric polymorphism)에 대한 코드 검토 시 적용하는 LDRA Rule은 Table 7과 같다.

Table 7 LDRA Rules for Parametric polymorphism

LDRA Rule	Rule Description	MISRA-C++:2008
551 S	Cast from base to derived for polymorphic type (MR).	5-2-3

Fig. 2에서는 파라메트릭 다형성 및 LSP를 위반한 소스코드 예시를 보여준다.

```
class morph_base
{
public:
  morph_base (int v);
  virtual int get_val ();
  int val;
};

morph_base::morph_base (int v)
{
  val = v;
}

int morph_base::get_val ()
{
  return val;
}
```

```
class morph_derived : public morph_base
{
public:
  morph_derived (int v);
  virtual int get_val ();
};

morph_derived::morph_derived (int v) : morph_base (v)
{
}

int morph_derived::get_val ()
{
  return 2 * val;
}
/*
This has violated the "Liskovsubstitution principle"
*/
```

Fig. 2 Example Code for LSP Violation

2.4.2.3 Overloading 검증방안

설계 검토를 통해 클래스 상속 관계에서 파생되는 Overloading 오류가 없는지 검증한다.

관련된 코딩 규칙에 대하여 정적분석을 수행하여 위반사항을 확인하고, 요구사항 기반 시험을 수행한 후에 Data coupling과 Control coupling coverage를 분석하여 개발자의 의도에 맞게 적합한 메서드가 호출되는지 여부를 확인하여 검증한다.

오버로딩(Overloading)에 대한 코드 검토 시 적용하는 LDRA Rule은 Table 8과 같다.

Table 8 LDRA Rules for Overloading

LDRA Rule	Rule Description	MISRA-C++:2008
392 S	Class data accessible thru non const member	9-3-1 9-3-2
396 S	Possible ambiguous numeric/pointer overloads.	-
433 S	Type conversion without cast.	5-0-11 5-0-12
555 S	Base class member name not unique.	10-2-1
556 S	Wrong order of catches for derived class.	15-3-6
595 S	Base member with change of access.	-

2.4.2.4 Type Conversion 검증방안

유형 변환(Type Conversion)의 취약점을 검증하기 위해서 Upcasting과 Type widening이 안전하게 되는지, 예기치 않은 문제가 발생하지 않았는지 확인한다. Narrowing type conversion 사용 시 명시적 변환과 관련된 코딩 규칙을 준수하였는지 확인하고, 소스코드 내에 Range check를 할 수 있도록 추가해서 취약점을 검증한다. 또한 요구사항 기반 시험 및 Structural Coverage 분석을 통해 유형 변환에 따른 오류가 없는지 검증한다.

유형 변환(Type Conversion)에 대한 코드 검토 시 적용하는 LDRA Rule은 Table 9와 같다.

Table 9 LDRA Rules for Type Conversion

LDRA Rule	Rule Description	MISRA-C++:2008
448 S	Base class pointer cast to derived class.	5-2-2
458 S	Implicit conversion: actual to formal param (MR).	-

2.4.2.5 Exception Management 검증방안

설계 검토 단계에서 예기치 않은 예외(Exception)에 의해 소프트웨어가 오동작하지 않도록 사전에 미리 확인한다. 정적분석(Static Analysis)을 수행하여 관련된 코딩규칙을 검사하고, 요구사항 기반 시험을 수행 후에 개발자가 의도한 방어 코드에 대한 구조적 커버리지를 분석하여 검증한다.

예외 관리(Exception Management)에 대한 코드 검토 시 적용하는 LDRA Rule은 Table 10과 같다.

Table 10 LDRA Rules for Exception Management

LDRA Rule	Rule Description	MISRA-C++:2008
44 S	Use of banned function, type or variable.	18-4-1
47 S	Array bound exceeded.	5-0-16
629 S	Divide by zero found.	0-3-1
671 S	Class data accessible thru non const handle.	9-3-2

2.4.2.6 Dynamic Memory Management 검증방안

Dynamic Memory Management의 검증방안은 메모리 할당 관련 문법 검토, 메모리 누수 검토 등을 포함한 정적분석을 수행하여 검증하고, 또한 요구사항 기반 시험 및 Structural Coverage 분석을 통해 메모리 관리에 취약점이 없는지 검증한다.

이 외에도 동적 메모리 관리의 문제는 개발하는 환경에 따라 고려해야할 요소가 많고 각각의 메모리 할당 기법에 따른 활동이 다양하다.

메모리 참조 모호성, 조각화 부족, 할당 해제 부족, 메모리 고갈, 조기 할당 해제, 업데이트 손실 및 부실 참조, 바인딩 해제된 할당 또는 할당 해제 시간과 관련하여 동적 메모리 관리 사용에 대한 강건성(Robustness)를 확인한다. 동적 메모리 관리에서의 취약점을 확인하기 위한 활동은 다음과 같다.

a. 할당자가 다른 참조가 존재하지 않는 (배타성) 메모리 참조를 반환하는지 확인한다.

b. 사용 가능한 충분한 메모리가 있을 때 할당 요청이 성공하도록 메모리가 구성되어 있는지 확인한다.

c. 더 이상 참조되지 않는 할당된 메모리가 재사용을 위해 필요하기 전에 재 확보되었는지 확인한다.

d. 프로그램에서 언제든지 필요한 최대 스토리지를 수용할 수 있는 충분한 메모리가 있는지 확인한다.

e. 참조 일관성이 유지되는지, 즉 각 객체가 고유하고 해당 객체로만 표시되는지 확인한다.

f. 객체 이동이 해당 객체에 대한 모든 참조와 관련하여 고유한지 확인한다.

g. 메모리 관리 작업(예: 할당, 할당 해제 및 사용 가능한 메모리 합체)이 지정된 시간 내에 완료되었는지 확인한다.

이러한 취약점을 확인하기 위한 활동에 대한 상세한 검증 기법은 Table 11과 같다.

Table 11 Verification Technique for DMM in OOT

Technique	Activity						
	a	b	c	d	e	f	g
Object pooling	AC	AC	AC	AC	AC	N/A	MMI
Stack allocation	AC	MMI	MMI	AC	AC	N/A	MMI
Scope allocation	MMI	MMI	MMI	AC	AC	MMI	MMI
Manual heap allocation	AC	AC	AC	AC	AC	N/A	MMI
Automatic heap allocation	MMI	MMI	MMI	AC	MMI	MMI	MMI

AC = application, MMI = memory management infrastructure, N/A = not applicable

객체 풀링(Object pooling) 방법은 개발자가 특정 유형의 객체를 풀(Pool)에 넣고 사용할 때마다 꺼내고 더 이상 필요하지 않은 각 객체가 실제로 해당 풀로 반환되고 풀로 반환되는 모든 객체에 더 이상 참조가 없는지 확인한다. 또한 풀은 객체에 따라 다르므로 개발자는 메모리 부족을 방지하기 위해 풀 크기를 조정한다.

스택 할당(Stack allocation)은 주어진 메서드와 호출된 메서드에서만 스택과 같은 할당 메서드를 사용하는 것이다. 개발자는 객체 할당에 필요한 각 스택의 크기를 고려하고 스택에 저장된 객체에 대한 참조가 만들어지지 않도록 한다.

범위 할당(Scope allocation)은 범위 메모리의 할당 규칙을 정하여 객체에 범위를 지정하고 활성화 객체를 관리하는 것이다. 범위가 지정된 객체를 저장하기 위해 백업 저장소를 관리하는 데 더 많은 주의를 기울여야 하며 메모리 조각화 위험을 최소화한다.

수동 힙 할당(Manual heap allocation)을 사용하려면 객체가 더 이상 참조되지 않는 경우에만 응용 프로그램이 각 객체의 할당을 취소한다. 일반적으로 객체는 이동되지 않지만 조각화로 인한 할당이 실패하지 않도록 보장해야 한다.

자동 힙 할당(Automatic heap allocation)은 가비지(사용하지 않는 객체)를 자동으로 수집하며, 응용 프로그램에서 더 이상 참조하지 않는 모든 객체가 사용 가능한 목록으로 반환되고 사용 가능한 목록이 통합되어 후속 메모리 할당이 성공하며 아직 남아 있는 객체가 수집되지 않았는지 확인한다. 개발자는 가비지 수집

활동이 응용 프로그램의 적시성을 방해하지 않으며 실행에 사용할 수 있는 메모리가 충분함을 보여야 한다. 많은 가비지 수집기에서 사용 중인 메모리가 사용 가능한 메모리에 가까워짐에 따라 수집 시간이 증가하므로 힙 크기에는 이 효과를 완화하기 위한 안전 여유가 포함되어야 한다.

동적메모리 관리(Dynamic Memory Management)에 대한 코드 검토 시 적용하는 LDRA Rule은 Table 12와 같다.

Table 12 LDRA Rules for Dynamic Memory Management

LDRA Rule	Rule Description	MISRA-C++:2008
44 S	Use of banned function, type or variable.	18-4-1
51 D	Attempt to read from freed memory.	0-3-1

2.4.2.7 Virtualization 검증방안

가상화(Virtualization)는 실행환경을 다른 환경으로 에뮬레이션하는 기술이다. 인터프리터(Interpreter)와 같은 가상화 기술은 개발한 소프트웨어를 다른 환경에서 동작할 수 있게 한다. 이런 가상화 기술은 실행코드로 검증해야 하는데, 정적분석(Static Analysis)을 수행하여 관련된 코딩규칙을 검사하고, 변환된 소스코드의 일치성을 확인하기 위하여 요구사항 기반 시험을 수행 후에 개발자가 의도한 방어 코드에 대한 구조적 커버리지를 분석하여 검증한다.

가상화(Virtualization)에 대한 코드 검토 시 적용하는 LDRA Rule은 Table 13과 같다.

Table 13 LDRA Rules for Virtualization

LDRA Rule	Rule Description	MISRA-C++:2008
202 S	Class data is not explicitly private.	11-0-1
357 S	Base class is mixed virtual and non virtual.	10-1-3
521 S	Class derived from virtual base class.	10-1-1
543 S	Virtual base not in diamond hierarchy.	10-1-2
559 S	Virtual member defined more than once.	10-3-1

2.5 OOT 취약점 검증방안의 적용

PX4 오픈소스에 정적분석 자동화 도구인 LDRA를 적용하여 제시한 단계별 검증 방안 중 코드 검토 결과의 의미를 분석하였다.

2.5.1 PX4 코드 검토 결과

구현된 소스코드에 코드 검토를 위한 LDRA Rule은 Table 14와 같다.

Table 14 LDRA Rules for OOT

LDRA Rule	Rule Description	MISRA-C++:2008
28 S	Duplicated base classes in a derived class.	-
44 S	Use of banned function, type or variable.	18-4-1
47 S	Array bound exceeded.	5-0-16
202 S	Class data is not explicitly private.	11-0-1
205 S	Use of multiple inheritance.	-
262 S	Non virtual function redefined.	-
327 S	Reuse of struct or class member name.	-
357 S	Base class is mixed virtual and non virtual.	10-1-3
392 S	Class data accessible thru non const member	9-3-1 9-3-2
396 S	Possible ambiguous numeric/pointer overloads.	-
433 S	Type conversion without cast.	5-0-11 5-0-12
448 S	Base class pointer cast to derived class.	5-2-2
458 S	Implicit conversion: actual to formal param (MR).	-
521 S	Class derived from virtual base class.	10-1-1
543 S	Virtual base not in diamond hierarchy.	10-1-2
551 S	Cast from base to derived for polymorphic type (MR).	5-2-3
555 S	Base class member name not unique.	10-2-1
556 S	Wrong order of catches for derived class.	15-3-6
559 S	Virtual member defined more than once.	10-3-1
595 S	Base member with change of access.	-
629 S	Divide by zero found.	0-3-1
671 S	Class data accessible thru non const	9-3-2

	handle.	
51 D	Attempt to read from freed memory.	0-3-1
70 D	DU anomaly, variable value is not used.	0-1-6 0-1-9
76 D	Procedure is not called or referenced in code analysed.	0-1-10

비행제어 소프트웨어 구현에 사용되는 오픈소스인 PX4 소스코드에 LDRA tool을 이용하여 25개 Coding Rule을 적용한 결과는 다음과 같다.

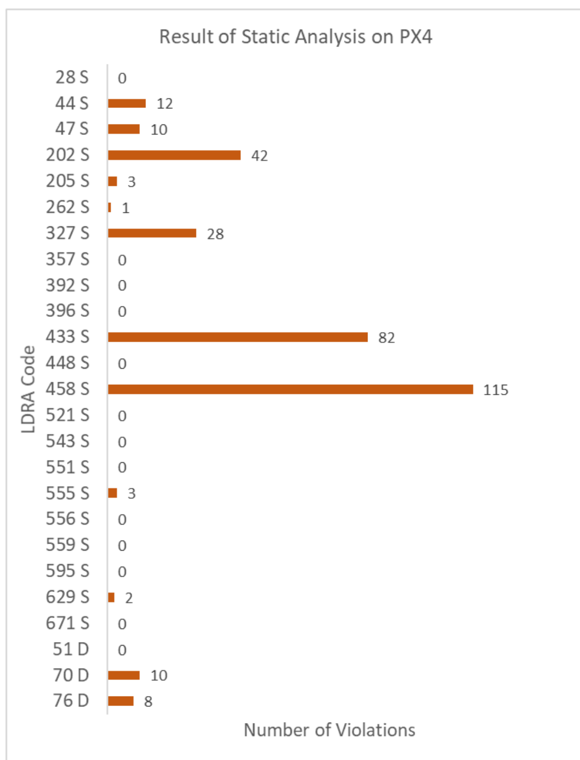

Fig. 3 Result of Static Analysis on PX4

LDRA를 통해 PX4 코드를 정적 시험 한 결과 Fig. 3과 같이 취약점이 검출되었으며 OOT&RT기반 시스템의 잠재적인 위험성이 존재한다는 것을 확인하였다.

가장 많이 검출된 항목은 '[458 S] Implicit conversion : actual to formal param' 으로 PX4 오픈소스의 검증대상 소스코드에는 함수 호출시에 암묵적인 형변환의 취약점이 다수 포함되어 있다고 해석할 수 있다.

또한 두 번째로 많이 검출된 항목은 '[433 S] Type conversion without case'으로 함수의 Overloading과 관련된 구현이 충분하지 않은 경우에 검출될 수 있는 규칙이라고 볼 수 있다. 따라서 개발자가 의도하지 않은 함수가 수행될 수 있는 가능성을 내재하고 있다고 해석할 수 있다.

III. 결 론

본 연구에서는 객체지향 언어로 개발한 소프트웨어에 대해 DO-332에서 제시한 OOT&RT의 취약점을 분석하고, 소프트웨어가 DO-332의 가이드라인을 따라 적합하게 개발되었는지 확인하기 위한 검증방법을 4가지로 제시하였다.

- 설계 검토 (Design Review)
- 코드 검토 (Code Review)
- 요구사항 기반 시험 (Requirements-Based Test)
- 구조적 커버리지 분석 (Structural Coverage Analysis)

OOT 취약점 중에서 일부 취약점은 소스코드를 구현한 이후에만 검증이 가능하지만, 소프트웨어 설계 단계에서부터 시험 단계까지 전체적으로 검증을 수행하여야 한다. 소스코드 구현 이후에는 테스트 자동화 도구(LDRA Tool Suite)를 활용하여 검증이 가능하다.

앞의 두 가지의 Coding Rule 이외에도 객체지향 기술의 취약점과 관련된 위반사항들이 검출되었으므로 PX4 오픈소스를 활용하여 기능을 구현할 경우에는 관련 취약점에서 파생되는 결함이 발생하지 않도록 주의를 기울여야 한다.

이처럼 사고에 따른 손실이 매우 큰 항공 소프트웨어 개발자 및 관련 업무를 하는 사람들은 객체지향 언어의 취약점에 대해 명확히 이해하며 개발 과정에 참여해야 하고, 설계 검토, 코드 검토 이외에도 요구사항 기반 시험 및 구조적 커버리지 분석과 같은 검증활동을 통해 기능 안전 무결성을 보증해야 한다.

향후 PX4 코드에 요구사항 기반 동적 시험 및 code coverage 시험을 통해 추가적인 위험요소를 분석할 필요가 있다. 또한 OOT 취약점 중에서 동적메모리관리(Dynamic Memory Management)와 가상화(Virtualization)는 개발 언어, 개발 방법, 개발 환경, 개발자 등 수많은 요인에 따라 달라질 수 있는 방법이기에 조금 더 많은 연구가 필요하다.

후 기

본 연구는 국토교통부/국토교통과학기술진흥원의 지원으로 수행되었음(과제번호: 21DPIW-C153651-03, 과제명: 공공혁신조달 무인이동체 통합기술관리 및 시험평가체계 개발).

References

[1] FAA, AC20-115D, “Advisory Circular in Airborne Software Development Assurance Using EUROCAE ED-12 and RTCA DO-178”, 2017

[2] RTCA Inc, DO-332, Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, 2011

[3] FAA, CAST-4, Object-Oriented Technology (OOT) In Civil Aviation Projects: Certification Concerns, 2000

[4] FAA, CAST-8, Using of the C++ Programming Language, 2002

[5] LDRA Ltd., User Guide for LDRA tool suite, 2016

[6] SGS-TÜV Saar GmbH, Certificate of LDRA Tool Suite, Certification Report No. K1C20003, 2015

[7] Liskov, B. H. and Wing, J. M., A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16(6). 1811–1841, 1994

[8] MISRA, MISRA-C++:2008, Guidelines for the use of the C++ language in critical systems, 2008

[9] LDRA Ltd., MISRA-C++:2008 Standards Model Compliance for C++, 2020.