논문 2025-2-12 http://dx.doi.org/10.29056/jsav.2025.06.12

# 이산사건시스템 형식론 기반 시뮬레이션 모델의 요구사항 명세 및 정적 코드 검증 도구

최창범*, 양승호**, 장수영*†

# Requirements Specification and Static Code Verification of Simulation Models based on Discrete Event System Formalism

Changbeom Choi*, Seungho Yang**, Sooyoung Jang*†


## 요 약

본 논문은 이산사건시스템 형식론 기반 시뮬레이션 모델의 신뢰성 향상을 위한 요구사항 명세와 LLVM/Clang 프레임워크 기반의 정적 코드 검증 도구를 제안한다. 국방 분야 등 복잡한 시스템의 모델링 및 시뮬레이션에서는 모델 코드와 시뮬레이션 엔진 API 간의 정확한 상호작용이 필수적이며, 이를 위반하면 탐지가 어려운 런타임 오류가 발생하여 시뮬레이션 결과의 유효성을 심각하게 저해할 수 있다. 본 연구에서는 MUST-CALL, NEVER-CALL, MUST-FOLLOW와 같은 요구사항 명세를 체계적으로 정의하고, 이를 검증하기 위한 도구로 LLVM/Clang의 경로 민감 심볼릭 실행 엔진을 활용하여 소스코드 실행 전에 검증하고, DES 모델링의 특수성을 반영한 커스텀 체커를 통해 정의된 명세 위반 여부를 체계적으로 검증한다. 이러한 접근은 도메인 특화 오류를 효과적으로 탐지하여 런타임 오류를 줄이고 신뢰성 높은 시뮬레이션 모델의 효율적인 개발에 기여할 수 있다.

## Abstract

This paper proposes a requirements specification and a static code verification based on the LLVM/Clang framework for simulation models based on the Discrete Event System (DES) formalism. In the modeling and simulation of complex systems, precise interaction between the model code and the simulation engine's API is essential for the early detection and resolution of code errors. This research defines requirement specifications like MUST-CALL, NEVER-CALL, and MUST-FOLLOW to support developers in strictly adhering to simulation model development guidelines. The proposed tool utilizes LLVM/Clang's static analysis to parse C++ code and systematically check for violations of the defined requirement specifications before execution. A key contribution of this research is performing static code verification that reflects the specific characteristics of modeling based on the DES formalism. This approach aims to reduce runtime errors and improve model maintainability, ultimately contributing to the development of efficient and reliable models.


**한글키워드 :** 이산사건시스템 형식론, 요구사항 명세, 정적 코드 검증, 시뮬레이션 모델, LLVM/Clang

**keywords :** Discrete Event System Formalism, Requirements Specification, Static Code Verification, Simulation Model, LLVM/Clang

* 국립한밭대학교 컴퓨터공학과
** 국립한밭대학교 도시공학과
† 교신저자: 장수영(email: syjang@hanbat.ac.kr)

접수일자: 2025.06.03. 심사완료: 2025.06.13.
게재확정: 2025.06.20.

## 1. 서 론

현대 컴퓨팅 자원의 발전과 더불어 다양한 분야에서 체계를 개발할 때 컴퓨터 모델링 및 시뮬레이션(Modeling & Simulation, M&S)의 필요성이 증대되고 있다. 특히 국방 M&S와 같이 복잡하고 대규모인 시스템을 분석하고 평가하는 데 있어 M&S는 필수적인 분석 수단으로 자리 잡았다. 이러한 시뮬레이션 시스템은 재사용성과 모듈성을 높이기 위해, 시뮬레이션의 시간 진행과 이벤트 처리를 담당하는 범용 시뮬레이션 엔진과 특정 대상 시스템의 동작을 모사하는 시뮬레이션 모델을 분리하는 아키텍처를 채택하는 경우가 많다. 이 구조에서 모델 개발자는 C++과 같은 프로그래밍 언어를 사용하여 시뮬레이션 엔진이 제공하는 API(Application Programming Interface)를 호출함으로써 모델의 동작을 구현한다.

이러한 아키텍처에서 시뮬레이션 결과의 신뢰성은 모델 자체의 논리적 정확성뿐만 아니라, 모델이 엔진의 API를 정해진 규칙과 순서에 따라 올바르게 사용하는지에 크게 의존한다. API의 부정확한 사용, 예를 들어 필수 초기화 함수의 누락, 금지된 함수의 호출, 또는 자원 할당/해제 순서의 위반 등은 예측 불가능한 런타임 오류나 미묘한 데이터 손상을 유발할 수 있다. 이러한 오류는 모델의 논리적 결함과 구별하기 어려워 디버깅을 매우 어렵게 만들며, 모델 검증 및 확인(Verification and Validation, V&V) 과정에서 심각한 걸림돌이 된다.

이 문제의 핵심은 모델의 구현 검증, 즉 "모델을 올바르게 구축하고 있는가?"의 문제에 있다. 전통적인 V&V 활동은 주로 "올바른 모델을 구축하고 있는가?"를 확인하는 개념 모델 유효성 확인에 초점을 맞추는 경향이 있다. 하지만 개념 모델이 아무리 정확하더라도, 이를 구현한 소스코드가 엔진과의 약속(API 프로토콜)을 위반한다면 시뮬레이션의 신뢰성은 보장될 수 없다.

이러한 배경하에, 본 연구는 시뮬레이션 실행 전 단계에서 소스코드 자체를 분석하여 API 사용 규칙을 요구사항 명세로 정의하고, 위반 여부를 자동으로 탐지하는 정적 분석 기법의 중요성에 주목한다. 정적 분석은 코드를 실행하지 않고 잠재적 오류를 식별함으로써, 복잡한 시뮬레이션 환경에서 디버깅에 걸리는 시간과 노력을 크게 줄이고 소프트웨어의 신뢰성을 근본적으로 향상시킬 수 있다.

따라서 본 논문은 이산사건시스템(Discrete Event System, DES) 형식론 기반 시뮬레이션 모델을 위한 핵심 API 사용 규칙을 체계적으로 정의하는 요구사항 명세 방법을 제안하고, LLVM/Clang 프레임워크를 활용하여 이러한 요구사항 명세를 C++ 소스코드 수준에서 자동으로 검증하는 정적 검증 도구를 제안한다.

본 연구의 주요 기여는 다음과 같다.

- DES 기반 시뮬레이션의 의미론적 제약을 반영하는 API 사용 규칙(예: MUST-CALL, NEVER-CALL, MUST-FOLLOW)의 체계적인 요구사항 명세 방법을 제안한다.
- LLVM/Clang 정적 분석기의 경로 민감 분석 및 확장성을 활용하여, 제안된 규칙을 C++ 소스코드에서 직접 검증하는 도구의 아키텍처를 제안한다.
- 범용 정적 분석 도구와 차별화되는 도메인 특화 검증 접근법을 통해, DES 시뮬레이션 모델 개발의 신뢰성과 생산성을 향상시키는 방안을 제안한다.

## 2. 관련 연구

### 2.1 이산사건시스템 형식론

이산사건시스템 형식론은 1976년 B. P. Zeigler

에 의해 제안된 집합론에 근거한 형식론으로 모의 대상 시스템을 해당 시스템보다 작은 요소로 세분화하여 각각 모델로 만들고 이를 조합하여 시스템을 구성하는 형태로 시스템을 모의한다[3]. DES 형식론은 세분화된 시스템의 구성 요소를 원자모델(Atomic Model)과 여러 모델을 조합하여 더 큰 모델로 구성하는 결합모델(Coupled Model)로 시스템을 구성한다. 특히 결합모델은 내부 구성 요소로 원자모델과 결합모델로 구성될 수 있어 시스템을 계층적이고 모듈러하게 표현할 수 있다.

원자모델은 시간 명세를 상태 천이 수준에서 표현할 수 있으며 이를 활용하여 시스템의 동작을 표현한다. 원자모델은 특정 시간에 대상 시스템의 상태를 상태 집합의 원소인 State로 표현하고 해당 State에서 외부 입력이 들어오거나 특정 시간 동안 외부로부터 입력 사건이 발생하지 않으면 출력 사건을 발생시키고 상태를 변경하는 모델이다. 이때 모델은 모사 대상 시스템의 행위에 해당하는 코드를 실행한다. 다음 그림1은 원자모델의 명세다.

$$AM = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta >$$

$X$ : Input Events Set

$Y$ : Output Events Set

$S$ : States Set

$\delta_{ext} : Q \times X \rightarrow S$; External Transition Function

$Q = \{(s,e) \mid s \in S,\ 0 \le e \le ta(s)\}$; Total state of AM,

$e$ : elapsed time

$\delta_{int} : S \rightarrow S$ ; Internal Transition Function

$\lambda : S \rightarrow Y$ ; Output Function

$ta : S \rightarrow R_{\infty}^{+}$ ; Time Advance Function

그림 1. 원자모델의 명세
Fig. 1. Specification of Atomic Model

원자모델은 한 번에 한 가지 상태(s∈S)에 머물고 있으며 한 상태(s∈S)에 최대로 머물 수 있는 시간은 ta(s)이다. 한 상태(s∈S)에 머물고 있을 때 (0≤e≤ta(s)) 외부 입력 이벤트가 발생하면 외부 상태 천이 함수($\delta_{ext}$)에 의해서 다른 상태로 상태를 변경하고, 한 상태에 머물 수 있는 시간이 지나면(e=ta(s)), 출력 함수(λ)에 의해 출력을 내보내고, 내부 상태 천이 함수($\delta_{int}$)에 의해서 다른 상태로 이동하게 된다.

결합모델(Coupled Model)은 모델들을 내부적으로 연결하여 만든 모델로 모델 간 연결을 관계로 표현하고 모델 간의 우선순위를 함수로 표현한다. 그림 2는 결합모델 명세이다.

$$CM = < X, Y, \{M_i\}, EIC, EOC, IC, SELECT >$$

$X$ : Input Events Set

$Y$ : Output Events Set

$\{M_i\}$ : Component Models Set

$EIC \subseteq X \times \bigcup_i X_i$ : External Input Coupling Relation

$EOC \subseteq \bigcup_i Y_i \times Y$ : External Output Coupling Relation

$IC \subseteq \bigcup_i Y_i \times \bigcup_j X_j$ : Internal Coupling Relation

$SELECT : 2^{\{M_i\}} - \phi \rightarrow M_i$

그림 2. 결합모델의 명세
Fig. 2. Specification of Coupled Model

결합모델은 3개의 집합(X, Y, M)과 3개의 관계(EIC, EOC, IC), 하나의 함수(SELECT)로 이루어져 있다. 입력 이벤트 집합, 출력 이벤트 집합은 각각 X와 Y로 표현되고 내부 구성 요소 모델의 집합(M)은 내부적으로 포함하고 있는 모델의 집합이다. 또한, 세 개의 관계는 결합모델의 입출력과 내부 구성 요소, 혹은 구성 요소 모델 사이의 연결 관계를 표현한다. 외부 입력 연결 관계(EIC)와 내부 연결(IC), 외부 출력 연결(EOC)은 각각 외부에서 이벤트가 내부로 전달되는 이벤트가 전달되어 영향을 받는 모델을 기술하고, 내부 연결의 경우 내부 구성 요소 간의 이벤트 연결 관계, 외부 출력 연결의 경우 내부 모델에서 외부로 이벤트를 발생시킬 때의 관계를 나타낸다.

DES 형식론을 바탕으로 시뮬레이션 모델을 개발하는 경우 모델 개발자는 모사 대상 시스템

의 기본적인 행위를 원자모델을 활용하여 표현하고 원자모델과 원자모델로 구성된 결합모델들을 조합하여 더 큰 결합모델을 구성하는 형태로 시뮬레이션 시스템을 개발한다. 또한, 시뮬레이션 모델을 실행시키는 시뮬레이션 엔진은 DES 형식론 기반의 API를 제공하여 시뮬레이션 모델 개발자가 해당 함수를 호출할 수 있도록 한다. 따라서, 프로그래밍 언어로 구성된 DES 시뮬레이터를 검사하기 위해서는 모델링 환경에서 제공하고 있는 함수인 시뮬레이션 엔진 API를 올바르게 활용하였는지 확인하는 것이 필요하다. 이러한 API의 정확한 사용은 모델이 DES 형식론의 의미론적 제약을 준수하며 시뮬레이션 엔진과 올바르게 상호작용을 하도록 보장하는 데 핵심적이다.

### 2.2 시뮬레이션 소스코드 검증 도구

일반적으로 소프트웨어에서 아주 정밀한 검사 단계를 거쳐 배포된 제품이라고 하더라도 버그가 없음을 보장하기 어렵기 때문에 다양한 소스코드 검사에 관한 연구가 진행되었다.

Spin은 1980년 벨 연구소에서 처음 개발되어 전화기 스위치에서 검증하는 도구로써 사용되었다[4][5]. 1991년에 오픈소스로 전환된 Spin은 현재는 NASA/JPL에서 활발히 유지 보수를 하고 있으며 소프트웨어 멀티 스레드 실행이나 병렬 실행과 관련된 알고리즘의 검증에 활용된다. Spin 모델은 C 언어와 비슷한 체계를 갖추고 있으며, 가벼우며 모호함이 적고 검증에 특화된 Promela(Process or Protocol Meta Language)로 기술된다.

Uppaal은 시간 개념을 포함한 모델 검사를 위한 도구로 모델을 설명하고 속성을 지정할 수 있는 상호 동작(Interactive)하는 환경을 갖춘 언어를 기반으로 한다[6]. Uppaal은 Spin과 유사하나 시간에 대한 명세와 처리에 대하여 검증할 수 있다.

Alloy는 객체 및 구조에 대한 제약을 일차 논리로 기술하고 SAT 솔버를 활용하여 해를 찾는 경량 형식 검증 도구로, 데이터 구조의 일관성 검사 등에 활용된다[7]. Alloy는 링크드 리스트(Linked lists), 해시 테이블(Hash tables)과 같은 소프트웨어 데이터 구조의 일관성을 분석하고 데이터베이스의 데이터 유형 간에 설정된 관계를 분석에 사용될 수 있으나 시간에 대한 특성을 분석할 수는 없다는 한계가 있다.

앞서 제시한 소프트웨어를 검사하기 위해서는 소스코드 그 자체에 대한 분석을 수행하기보다는 검증 대상 시스템의 특징을 표현하는 모델과 요구사항을 표현하고 이를 순회하는 모델 검증(Model Checking) 방식을 활용하고 있다. 하지만 이러한 방식은 실제적인 시뮬레이션 코드 개발에서 지켜야 할 API 사용 규칙과 같은 저수준의 구현 세부 사항을 직접적으로 검증하는 데에는 상대적으로 미흡하다.

### 2.3 LLVM/Clang 정적 분석 프레임워크

본 연구에서 제안하는 검증 도구의 기술적 기반은 LLVM/Clang 프레임워크이다. LLVM(Low Level Virtual Machine)[8]은 컴파일러 및 관련 도구 개발을 위한 모듈식 재사용 가능 구성 요소들의 집합이다. LLVM은 특정 프로그래밍 언어나 아키텍처에 종속되지 않는 중간 표현(Intermediate Representation, IR)을 중심으로 설계되어, 다양한 언어의 프론트엔드와 다양한 하드웨어의 백엔드를 유연하게 결합할 수 있다.

Clang은 C, C++, Objective-C 언어를 위한 LLVM의 고성능 프론트엔드로, 소스코드를 파싱하고 분석하는 역할을 수행한다. Clang의 핵심적인 장점은 단순한 컴파일러를 넘어, 소스코드에 대한 풍부한 정보에 접근할 수 있는 라이브러리 형태로 설계되었다는 점이다. 이를 통해 IDE, 리

팩토링 도구, 정적 분석기 등 다양한 소스코드 기반 도구를 개발할 수 있다.

본 연구에서 특히 주목하는 것은 Clang에 내장된 Clang Static Analyzer(CSA)이다. CSA는 C/C++ 소스코드의 오류를 찾기 위해 설계된 강력한 정적 분석 엔진이다[9]. CSA의 핵심 기술은 경로 민감 심볼릭 실행(path-sensitive symbolic execution)이다. 이 기법은 프로그램을 실제로 실행하는 대신, 변수의 값을 구체적인 값이 아닌 심볼로 표현하고 프로그램의 제어 흐름을 따라 가능한 모든 실행 경로를 탐색한다. 각 경로에서 프로그램의 상태(예: 변수의 값, 메모리 상태)를 추상적으로 추적함으로써, 특정 조건에서만 발생하는 복잡한 오류를 탐지할 수 있다.

CSA는 다음과 같은 핵심 구성 요소를 통해 확장 가능한 분석을 지원한다.

- 추상 구문 트리(Abstract Syntax Tree, AST): Clang은 소스코드의 구문 구조를 트리 형태로 표현한 AST를 생성한다. 분석기는 이 AST를 순회하며 함수 호출, 변수 선언 등 코드의 구조적 요소를 식별한다.
- 제어 흐름 그래프(Control Flow Graph, CFG): 함수 내의 모든 가능한 실행 경로를 나타내는 그래프이다. CSA는 CFG를 기반으로 경로 민감 분석을 수행한다.
- 커스텀 체커(Custom Checkers): CSA의 가장 큰 장점 중 하나는 확장성이다. 개발자는 CSA가 제공하는 API를 사용하여 특정 유형의 버그를 찾기 위한 자신만의 '체커'를 작성할 수 있다[10].

이러한 LLVM/Clang의 견고한 C++ 파싱 능력, 경로 민감 심볼릭 실행 엔진, 그리고 높은 확장성은 본 연구에서 제안하는 DES 시뮬레이션 모델 특화 정적 검증 도구를 개발하는 데 이상적인 기반을 제공한다.

## 3. DES 형식론 기반 시뮬레이션 모델의 요구사항 명세 및 정적 코드 검증 도구

시뮬레이션 엔진 API를 활용하여 시뮬레이션을 개발하는 과정에서는 시뮬레이션 모델의 파라미터를 변경하는 수준이 아닌 프로그래밍 언어를 활용하여 시뮬레이션 모델을 개발하는 경우 특별한 제약사항이 없이 시뮬레이션 모델을 개발할 수 있기 때문에 다양한 요구사항들이 존재할 수 있다. 예를 들어 자동으로 코드를 생성하는 모델링 및 시뮬레이션 환경에서는 코드가 자동으로 생성되는 영역과 사용자 코드를 기술하는 영역으로 나뉠 수 있다.

이때 개발자는 시뮬레이션 엔진 API를 호출해야 하는 영역에 모의 대상 코드가 배치될 수 있으며 모의 대상 코드 내에 시뮬레이션 엔진 API가 호출되는 등 시뮬레이션 모델의 유지 보수를 어렵게 만들 수 있다. 이와 같은 환경에서는 개발되는 프로그래밍 환경에서 문법에 부합되게 코드를 작성하는 경우 소프트웨어를 실행시키기 전에는 시뮬레이션의 오동작을 확인 할 수 없어 사용자 코드 영역 내에 사용자 코드가 존재하는지, 혹은 시뮬레이션 엔진에서 API를 주어진 규칙에 맞게 호출하는지 확인하는 것이 필요하다. 그림 3은 제안하는 검사방법에 대한 개념도이다.


그림 3. 소프트웨어 검사 개념도
Fig. 3. Conceptual Diagram of Software Testing

### 3.1 DES 시뮬레이션 모델의 요구사항 명세

제안하는 검증 방법론의 첫 단계는 DES 형식론 기반 시뮬레이션의 암묵적인 의미론적 규칙들을 명시적이고 기계가 검증할 수 있는 API 사용

규칙으로 정의하는 것이다. 이는 범용 C++ 코드의 오류를 넘어, 시뮬레이션 모델과 엔진 간의 올바른 상호작용을 보장하는 데 초점을 맞춘다. 본 연구에서 제안하는 주요 요구사항 명세는 표 1과 같다.

각 규칙의 논리적 근거는 다음과 같다.

- MUST-CALL: DES 모델은 시뮬레이션 엔진에 의해 관리되는 특정 생명주기(lifecycle)를 따른다. 모델이 올바르게 동작하기 위해서는 생명주기의 특정 시점(예: 초기화)에 엔진이 제공하는 필수 API(예: 이벤트 스케줄링, 상태 등록)를 반드시 호출해야 한다. 이 규칙은 이러한 필수적인 상호작용이 누락되어 모델이 비정상적으로 동작하는 것을 방지한다.
- NEVER-CALL: 시뮬레이션의 재현성과 안정성을 보장하기 위해, 시뮬레이션 엔진은 메모리, 시간 등 핵심 자원을 독점적으로 관리해야 한다. 모델 코드가 malloc, free와 같은 저수준 메모리 관리 함수나 특정 시스템 호출을 직접 사용하면 엔진의 관리 메커니즘과 충돌하여 메모리 누수, 교착 상태 등 예측 불가능한 부작용을 일으킬 수 있다. 이 규칙은 모델이 엔진의 자원 관리 정책을 따르도록 강제한다.
- MUST-FOLLOW: fopen 후 fclose 호출과 같이, 자원을 획득(acquire)한 후에는 반드시 해제(release)해야 하는 패턴은 프로그래밍의 기본 원칙이다. 특히 장시간 실행되는 시뮬레이션에서 자원 누수는 누적되어 시스템 전체의 안정성을 심각하게 위협할 수 있다. 이 규

표 1. 요구사항 명세
Table 1. Specification of Requirements

| 요구사항 | 검증 규칙 | 예제 |
|---|---|---|
| 반드시 OOO 함수는 호출해야 한다. | MUST-CALL | CAC의 Update 함수 안에서는 UpdateSpatial 함수를 호출해야 한다.<br>[MUSTCALL]<br>CAC::Update<br>CAC::UpdateSpatial |
| OOO 함수는 사용자 코드에서 호출하면 안 된다. | NEVER-CALL | CAC의 모든 함수 안에서는 malloc 함수를 호출하면 안 된다.<br>[NEVERCALL]<br>*malloc |
| XXX 함수 이후에는 OOO 함수를 호출해야 한다. | MUST-FOLLOW | CAC의 모든 함수 안에서는 fopen함수를 호출하였다면 fclose함수를 호출해야 한다.<br>[MUSTFOLLOW]<br>* fopen fclose |
| 모든 경로에서 OOO 함수는 한 번만 호출되어야 한다. | MUST-ONCE-CALL | CAC의 Update 함수 안에서는 UpdateSpatial 함수를 한 번만 호출해야 한다.<br>[MUSTFOLLOW]<br>CAC::Update<br>CAC::UpdateSpatial |
| 특정 영역에 사용자 코드가 있어야 한다. | MUST-BE | CAC의 Update 함수 내 사용자 코드에 코드가 존재해야 한다. |
| 특정 영역에 사용자 코드가 없어야 한다. | MUST-NOT-BE | CAC의 Update 함수 내 사용자 코드에 코드가 존재하면 안 된다. |

칙은 모든 실행 경로에서 이러한 자원 관리 프로토콜이 준수되는지를 보장한다.

- MUST-ONCE-CALL: 특정 초기화 함수나 전역 상태 설정 함수는 중복 호출될 경우, 의도치 않은 상태 변경이나 성능 저하를 유발할 수 있다. 이 규칙은 이러한 함수가 단 한 번만 호출되도록 보장하여 프로그램의 일관성을 유지한다.
- MUST-BE / MUST-NOT-BE: 코드 자동 생성 기능이 있는 모델링 환경에서는 엔진 관리 영역과 사용자 코드 영역이 명확히 구분된다. 이 규칙들은 개발자가 반드시 구현해야 할 영역(MUST-BE)과 수정해서는 안 되는 영역(MUST-NOT-BE)을 명확히 하여, 코드의 구조적 무결성을 유지하고 유지보수성을 향상시킨다.

### 3.2 LLVM/Clang 기반 정적 검증 도구

본 절에서는 3.1절에서 정의한 요구사항 명세를 자동으로 검증하는 도구의 아키텍처를 기술한다. 이 도구는 2.3절에서 설명한 LLVM/Clang 정적 분석 프레임워크, 특히 CSA의 확장 기능을 기반으로 구현된다.

도구의 전체적인 아키텍처는 규칙 설정 파서, Clang 분석기, 그리고 위반 보고 모듈로 구성된다. 이 중 핵심적인 기술적 독창성은 커스텀 체커(Custom Checkers) 구현에 있다. 각 요구사항 명세는 CSA의 API를 활용하는 하나 이상의 특화된 체커 모듈로 구현된다. 체커의 동작 방식은 다음과 같다.

- API 호출 감지: CSA가 제공하는 콜백 함수인 checkPreCall과 checkPostCall을 활용한다. 분석기가 AST를 순회하다가 특정 함수(예: fopen 또는 DES 엔진의 특정 API)의 호출을 만나면, 이 콜백 함수들이 트리거되어 등록된 체커의 분석 로직이 실행된다.
- 경로 민감 상태 추적: CSA의 핵심 객체인 ProgramState를 사용하여 규칙 검증에 필요한 문맥 정보를 경로 민감하게 추적한다. ProgramState는 특정 실행 경로상의 프로그램 상태(변수 값, 할당된 자원 등)를 추상적으로 표현하는 데이터 구조이다. 예를 들어, MUST-FOLLOW 규칙을 검증하는 체커는 fopen 호출을 감지했을 때 ProgramState에 '파일이 열렸음'을 나타내는 심볼릭 상태를 기록한다. 이후 분석기는 CFG를 따라 모든 경로를 탐색하며, 함수가 종료되기 전에 fclose 호출을 통해 이 상태가 제거되는지를 확인한다.
- 규칙 위반 판정: 각 체커는 담당하는 규칙의 의미론에 따라 AST 노드 정보, CFG 경로, 그리고 ProgramState에 추적된 상태 정보를 종합적으로 분석하여 규칙 위반 여부를 최종적으로 판정한다. 위반이 탐지되면, 위반 보고 모듈은 위반된 규칙, 관련 소스 코드 위치, 그리고 가능한 경우 위반이 발생한 실행 경로 등의 상세 정보를 사용자에게 제공한다.

이처럼 제안하는 도구는 LLVM/Clang이라는 강력하고 범용적인 프레임워크를 기반으로 하되, DES 시뮬레이션이라는 특정 도메인의 의미론적 제약을 이해하고 강제하는 커스텀 체커를 개발함으로써 독창성을 확보한다.

### 3.3 차별성 및 독창성

본 연구에서 제안하는 검증 도구는 기존 연구 및 범용 LLVM/Clang 도구와 다음과 같은 차별성 및 독창성을 갖는다:

- DES 특화 요구사항 명세 정의: 본 연구는 일반적인 C++ 코드의 오류를 넘어, DES 시뮬레이션 모델 개발의 특수성을 반영한 API 사용 규칙(표 1)을 체계적으로 정의한다. 이러한 규칙들은 DES 모델과 시뮬레이션 엔진

간의 올바른 상호작용을 보장하는 데 초점을 맞추고 있다.

- LLVM/Clang의 맞춤형 활용: 본 연구는 LLVM/Clang의 정적 분석 기능을 DES라는 특정 도메인 문제 해결에 맞춤형으로 적용한다. 즉, LLVM/Clang을 단순 사용하는 것을 넘어, DES 형식론 기반 시뮬레이션 모델의 요구사항 명세의 의미론적 제약을 이해하고 이를 강제하는 커스텀 체커를 개발하는 것이 핵심적인 독창성이다. 범용 LLVM/Clang 도구는 DES 요구사항 명세의 특정 사용 맥락이나 규칙에 대한 사전 지식을 가지고 있지 않다.
- 실행 전 정적 검증: 많은 DES 검증 도구들이 시뮬레이션 실행 중 또는 실행 후 분석에 의존하는 반면, 제안하는 도구는 코드 실행 전에 정적 분석을 통해 오류를 조기에 발견함으로써 개발 시간과 디버깅 노력을 크게 절감할 수 있다.

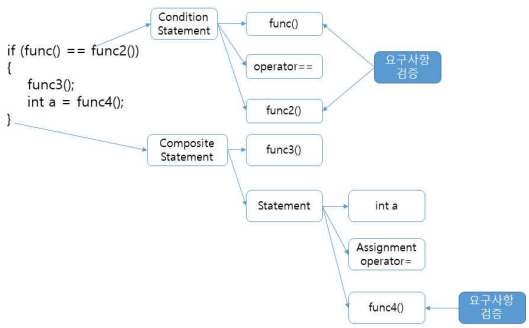


그림 4. If 구문에 대한 검증
Fig. 4. Verification of If statement

### 3.4 검증 과정 예시 및 결과

제안하는 도구가 실제로 어떻게 API 사용 규칙 위반을 탐지하는지 이해를 돕기 위해, LLVM/Clang이 코드를 분석하는 방식과 연계하여 설명한다. LLVM/Clang은 소스코드를 파싱하여 AST를 생성하고, 이를 바탕으로 CFG를 구축한다. 커스텀 체커는 이러한 AST와 CFG를 탐색하며 프로그램의 상태를 추적하고 규칙 위반을 검사한다.

예를 들어, 그림 4는 간단한 if 구문에 대한 LLVM/Clang의 구문 트리 생성 과정을 보여준다. 만약 if (func() == func2()) 조건문에서 func() 함수가 특정 조건에서는 호출되어서는 안 되는 요구사항 명세(NEVER-CALL 규칙 위반)라고 가정하자. 본 도구의 커스텀 체커는 AST를 순회하면서 func() 호출을 식별하고, 현재 제어 흐름(조건문 내)과 정의된 NEVER-CALL 규칙을 비교하여 위반 여부를 판단한다. 유사하게, 만약 if 구문의 본문 내 func3() 호출 이후에 반드시 func4()가 호출되어야 하는 MUST-FOLLOW 규칙이 있다면, 체커는 func3() 호출 시 ProgramState에 해당 정보를 기록하고, 이후 모든 실행 경로에서 func4()가 호출되는지를 CFG를 따라 검사하여 위반 시 경고를 발생시킨다.

구체적인 결과 제시는 본 논문의 범위를 넘어서는 추가적인 실험 및 평가가 필요하지만, 제안하는 도구는 표 1에 명시된 다양한 유형의 API 오용 사례를 탐지할 수 있도록 설계되었다. 예를 들어, 다음과 같은 결과들을 기대할 수 있다:

- MUST-CALL 위반: DES 모델의 특정 함수(예: initialize, outputFunction) 내에서 필수적으로 호출되어야 하는 엔진 API(예: scheduleEvent)가 누락된 경우를 탐지한다.
- NEVER-CALL 위반: 모델 코드 내에서 직접적인 동적 메모리 할당 함수(malloc, new 등)가 사용된 경우를 탐지하여 엔진의 메모리 관리 정책을 따르도록 유도한다.
- MUST-FOLLOW 위반: fopen 후 fclose가 누락되거나, 특정 자원 획득 API 호출 후 해당 자원 반환 API 호출이 누락된 경로를 탐지한다.

- MUST-ONCE-CALL 위반: 특정 초기화 API가 불필요하게 여러 번 호출되는 경우를 탐지한다.

이러한 정적 분석 결과는 개발자에게 코드 실행 전에 잠재적인 오류 위치와 원인을 명확히 제시함으로써, 디버깅 시간을 단축하고 시뮬레이션 모델의 전체적인 품질과 신뢰성을 향상시키는 데 직접적으로 기여할 수 있다. 이는 특히 복잡하고 대규모의 국방 M&S 모델 개발에서 효과적일 것으로 예상된다. 본 연구에서 제안하는 도구의 프로토타입을 개발하고 실제 프로젝트 코드에 적용하여 정량적인 효과를 측정하는 것은 향후 중요한 연구 과제가 될 것이다.

## 4. 결론 및 향후 연구

본 연구는 이산사건시스템 형식론 기반 시뮬레이션 모델의 C++ 구현 과정에서 발생할 수 있는 API 오용 문제를 해결하기 위해, DES 특화 요구사항 명세와 LLVM/Clang 기반의 정적 소스 코드 검증 도구를 제안했다. 제안된 방법론은 시뮬레이션 모델과 엔진 간의 의미론적 제약을 명시적인 API 사용 규칙으로 정의하고, 이를 LLVM/Clang의 경로 민감 심볼릭 실행 엔진과 커스텀 체커를 통해 코드 실행 전에 자동으로 검증한다. 이를 통해 기존의 모델 체킹이나 범용 정적 분석이 놓치기 쉬운 도메인 특화 오류를 효과적으로 탐지하여, 시뮬레이션 모델의 신뢰성과 유지보수성을 높이고 개발 효율성을 향상시키는 데 기여할 수 있음을 보였다. 본 연구는 특정 도메인의 지식을 정적 분석 기술에 접목하여 소프트웨어 품질을 향상시키는 실용적인 접근법의 성공적인 사례가 될 수 있다.

본 연구를 기반으로 다음과 같은 방향으로 연구를 확장할 수 있다.

- 규칙 발견 자동화: 현재 수동으로 이루어지는 요구사항 명세 작성 과정을 개선하기 위해, 대규모의 기존 시뮬레이션 모델 코드 코퍼스를 분석하여 DES 특화 API 사용 패턴을 통계적으로 마이닝하는 기법을 연구할 수 있다. 이를 통해 새로운 규칙을 자동으로 발견하고 기존 규칙을 검증하는 데 활용될 수 있다.
- 고급 분석 기법 적용: LLVM의 전역 최적화 단계에서 사용되는 호출 그래프 분석이나 요약 기반 분석 기법을 도입하여 여러 함수와 파일에 걸친 복잡한 오류 패턴을 추적하는 연구를 수행할 수 있다.

이 논문은 2024년도 교육부 및
한국연구재단의 국립대학육성사업으로
지원된 연구임

## 참 고 문 헌

[1] S. Park, “Modeling & Simulation Technology, How is it used in the defense field?”, Science & Technology Policy, 25(10), pp.18-21, 2015, URL : https://www.stepi.re.kr/site/stepiko/PeriodicReportView.do?reIdx=207&pageIndex=3&cateCont=A0504

[2] H. Oh, D. Kim, Y. Kim, “An Integrated Simulation Method for M-on-N Engagement Analysis using AddSIM”, Proceedings of Korean Society for Aeronautical and Space Sciences Spring Conference, pp.830-835, Apr. 4, 2012, URL: https://www.dbpia.co.kr/journal/articleDetail?nodeId=NODE01838655

[3] Bernard P. Zeigler, Hae Sang Song, Tag Gon Kim, Herbert Praehofer, “DEVS framework for modelling, simulation, analysis, and design of hybrid systems”, Proceedings of Hybrid Systems II,

pp.529-551, Oct. 28, 1994, DOI : https://doi.org/10.1007/3-540-60472-3_27
[4] Gerard J. Holzmann, Smith Margaret, “Software model checking: Extracting verification models from source code”, Software Testing, Verification and Reliability, 11(2), pp.65-79, 2001, DOI : http://dx.doi.org/10.1002/stvr.228
[5] Gerard J. Holzmann, Rajeev Joshi, “Model-driven software verification”, Proceedings of International SPIN Workshop on Model Checking of Software, pp.76-91, Apr. 1, 2004, DOI : https://doi.org/10.1007/978-3-540-24732-6_6
[6] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, Wang Yi, “UPPAAL—a tool suite for automatic verification of real-time systems”, Proceedings of Hybrid Systems III, pp.232-243, Oct. 22, 1995, DOI : https://doi.org/10.1007/BFb0020949
[7] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, Daniel Jackson, “Alloy*: A general-purpose higher-order relational constraint solver”, Formal Methods in System Design, 55, pp.1-32, 2019, DOI : https://doi.org/10.1007/s10703-016-0267-2
[8] Chenyang Li, Jiye Jiao, “LLVM Framework: Research and Applications”, Proceedings of the 19th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), pp.1-6, July 29, 2023, DOI: https://doi.org/10.1109/ICNC-FSKD59587.2023.10281186
[9] Kristóf Umann, Zoltán Porkoláb, “Detecting Uninitialized Variables in C++ with the Clang Static Analyzer”, Acta Cybernetica, 25(4), pp.923 - 940, DOI: https://doi.org/10.14232/actacyb.282900
[10] Gabor Horvath, Reka Kovacs, Richard Szalay, Zoltan Porkolab, “Implementing and Executing Static Analysis Using LLVM and CodeChecker”, arXiv:2408.05657, pp.1-63, 2024, URL : https://arxiv.org/abs/2408.05657

## 저 자 소 개

최창범(Changbeom Choi)

2005.2 경희대학교 컴퓨터공학과 학사
2007.2 한국과학기술원 전산학 석사
2014.8 한국과학기술원 전자공학 박사
2014.9-2021.2 한동대학교 부교수
2021.3-현재 : 국립한밭대학교 부교수
<주관심분야> 체계공학, 모델검증, 국방 모델링 및 시뮬레이션

양승호(Seungho Yang)

2009.2 서울대학교 지구환경시스템공학부 졸업
2014.8 서울대학교 건설환경공학부 박사
2015.3-2017.4 한아도시연구소 팀장
2017.5-2018.3 고양시정연구원 연구원
2018.6-2021.1 York University 박사후 연구원
2021.3-현재 : 국립한밭대학교 교수
<주관심분야> 도시설계, 스마트도시

장수영(Sooyoung Jang)

2006.2 한국과학기술원 산업공학과 졸업
2008.8 한국과학기술원 산업및시스템공학과 석사
2014.2 한국과학기술원 산업및시스템공학과 박사
2014.3-2017.9 삼성전자 책임연구원
2017.10-2023.2 한국전자통신연구원 선임연구원
2023.3-현재 : 국립한밭대학교 조교수
<주관심분야> 강화학습, 인공지능응용