JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 26 No. 1, pp. 171-178, January 2021
https://doi.org/10.9708/jksci.2021.26.01.171

Method for Detecting Modification of Transmitted Message in C/C++ Based Discrete Event System Specification Simulation

Hae Young Lee*

*Assistant Professor, Major of Digital Security, Cheongju University, Cheongju, Korea

[Abstract]

In this paper, the author proposes a method for detecting modification of transmitted messages in C/C++ based Discrete Event System Specification (DEVS) simulation. When a message generated by a model instance is delivered to other model instances, it may be modified by some of the recipients. Such modifications may corrupt simulation results, which may lead to wrong decision making. In the proposed method, every model instance stores a copy of every transmitted message. Before the deletion of the transmitted message, the instance compares them. Once a modification has been detected, the method interrupt the current simulation run. The procedure is automatically performed by a simulator instance. Thus, the method does not require programmers to follow secure coding or to add specific codes in their models. The performance of the method is compared with a DEVS simulator.

▸ **Key words**: Message Integrity, Secure Modeling & Simulation, Secure Coding, Discrete Event System Specification (DEVS), Simulation

[요 약]

본 논문에서는 C/C++ 기반 이산 사건 시스템 명세(DEVS) 시뮬레이션에서 전송 메시지 변경을 탐지하는 기법을 제안한다. 모델 인스턴스가 생성한 메시지가 다른 모델 인스턴스로 전달될 때, 메시지가 수신자 중 일부가 이를 수정할 수도 있다. 이러한 변경은 시뮬레이션 결과를 오염시킬 수 있으며, 오염된 시뮬레이션 결과는 잘못된 의사결정으로 이어질 수 있다. 제안 기법에서는 모든 모델 인스턴스가 전송 메시지의 사본을 저장한다. 전송 메시지가 삭제되기 전, 인스턴스는 저장된 메시지와 전송 메시지를 비교한다. 변경이 탐지되면 현재 시뮬레이션 실행을 중단시킨다. 모든 절차는 시뮬레이터 인스턴스에 의해 자동으로 수행된다. 그러므로 제안 기법은 개발자에게 시큐어 코딩을 준수하거나 특정 코드를 추가하도록 강요하지 않는다. 제안 기법의 성능을 기존 DEVS 시뮬레이터와 비교한다.

▸ **주제어**: 메시지 무결성, 시큐어 모델링 시뮬레이션, 시큐어 코딩, 이산 사건 시스템 명세, 시뮬레이션

• First Author: Hae Young Lee, Corresponding Author: Hae Young Lee
*Hae Young Lee (whichmeans@gmail.com), Major of Digital Security, Cheongju University
• Received: 2021. 01. 13, Revised: 2021. 01. 26, Accepted: 2021. 01. 26.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

컴퓨터 시뮬레이션(computer simulation, 이하 시뮬레이션)은 분석할 시스템(system under study) 대상으로 실험을 수행하기 어렵거나 분석할 시스템이 존재하지 않는 경우, 대상 시스템의 행위(behavior)를 예측하는 데 사용할 수 있다[1,2]. 예를 들어, 시뮬레이션을 사용하여 코로나바이러스감염증-19(COVID-19)의 2차 확산을 예측하거나[3], COVID-19 확산에 따른 전 세계 공급망(global supply chain)에 대한 영향을 평가할 수 있다[4]. 또는 스마트 그리드(smart grid) 대상의 사이버 공격(cyber attack)의 영향을 평가하거나[5], 사이버 공격을 기만(deception)하는 기술을 실험할 수 있다[6]. 이 외에도 재난 대비(disaster preparedness)[7], 제조(manufacturing)[8] 등의 분야에서도 시뮬레이션이 사용된다. 즉, 대상 시스템을 컴퓨터로 표현한 모델(model)을 대상으로 실험을 수행함으로써 대상 시스템의 행위를 예측하여 의사결정에 활용할 수 있다[9].

이산 사건 시스템 명세(Discrete Event System Specification, 이하 DEVS)[10,11]는 수학적 형식론(formalism)을 기반으로 이산 사건 시스템(discrete event system)을 모델링하고 시뮬레이션하는 방법론(methodology)이다. DEVS 역시 감염병 확산[12], 사이버 보안[13], 재난 대비[14], 제조[15] 등의 분야에서 사용된다. 보안 관점에서, 형식론으로 기술된 DEVS 모델은 무결하다고 할 수 있다. 그러나 이를 컴퓨터로 구현한 라이브러리, 모듈, 엔진 등의 DEVS 구현물(implementation)에는 보안 약점(weakness)이나 취약점(vulnerability)이 존재할 수 있다. 예를 들어, C/C++ 프로그래밍 언어로 개발된 DEVS 구현물[16,17]에는 버퍼 오버플로(buffer overflow)가 발생할 수 있으며, 버퍼 오버플로로 인해 모델의 상태 변수(state variable)가 오염(corruption)될 수 있다[18,19]. 또한, C/C++ DEVS 구현물에는 모델 간에 전달되는 메시지가 변경될 수 있는 전송 메시지 변경(transmitted message modification) 약점이 존재할 수 있다[20]. 버퍼 오버플로나 전송 메시지 변경 등이 발생하면 시뮬레이션 최종 결과가 오염될 수 있다. 오염된 시뮬레이션 결과는 잘못된 의사결정으로 이어질 수 있다.

본 논문은 C/C++로 작성된 DEVS 구현물인 adevs[16]에서 전송 메시지 변경을 탐지하는 기법을 제안한다. 제안 기법에서 메시지(사건)를 외부로 출력하는 서브 모델은 외부 출력과 동시에 해당 메시지를 백업한다. 메시지가 연결된 모델에게 전달되어 처리된 이후, 메시지를 출력한 모델은 출력한 메시지와 백업된 메시지를 비교하여 전송 메시지 변경 여부를 탐지한다. 메시지의 백업 및 비교가 자동으로 수행될 수 있도록 adevs 시뮬레이터를 수정하였다. 그러므로 개발자가 제안 기법의 적용을 위해 어떠한 코드도 수정할 필요가 없다. 또한, 제안 기법은 개발자가 시큐어 코딩(secure coding)[21]을 철저하게 준수한다고 가정하지 않는다. 그러므로 시큐어 코딩이 적용되지 않은 구현물에서도 전송 메시지 변경을 탐지할 수 있다. 시뮬레이션 수행 시간을 기준으로 제안 기법의 성능을 평가한다.

본 논문의 구성은 다음과 같다. 2절에서는 배경으로 DEVS를 간단히 소개하고, 논문에서 다루는 문제를 정의한다. 3절에서는 제안 기법을 자세히 설명한다. 다른 C/C++ 기반 DEVS 구현물에도 제안 기법의 적용이 가능하나, 개념 증명(proof-of-concept)으로 adevs에만 적용하여 설명한다. 4절에서는 제안 기법과 adevs 간의 성능 비교 결과를 기술한다. 5절에서는 결론 및 향후 연구과제를 다룬다.

II. Background and Problem Statement

1. DEVS Formalism

Zeigler가 개발한 DEVS[10,11]는 이산 사건 시스템을 정형적(formal)으로 모델링하고 시뮬레이션할 수 있는 형식론이다. DEVS에서는 (서브) 시스템의 행위(behavior)를 표현하는 원자(atomic) DEVS 모델과 (서브) 시스템의 구조(structure)를 표현하는 결합(coupled) DEVS 모델을 사용하여 대상 시스템을 모델링한다.

원자 DEVS 모델의 구조는 다음과 같다.

$$M = \langle X,\ Y,\ S,\ \delta_{ext},\ \delta_{int},\ \lambda,\ ta \rangle \quad (1)$$

위의 식에서 X는 입력 사건의 집합(set of input events), Y는 출력 사건의 집합(set of output events), S는 순차 상태의 집합(set of sequential states), $Q = \{(s, e) \mid s \in S,\ 0 \le e \le ta(s)\}$가 전체 상태의 집합(set of total states)일 때, δ_{ext}: $Q \times X \rightarrow S$는 외부 전이 함수(external transition function), δ_{int}: $S \rightarrow S$는 내부 전이 함수(internal transition function), λ: $S \rightarrow Y$는 출력 함수(output function), $\mathbb{R}_{[0,\infty]}$가 0과 ∞를 포함한 양의 실수일 때, ta: $S \rightarrow \mathbb{R}_{[0,\infty]}$는 시간 전진 함수(time advance function)이다.

원자 DEVS 모델은 외부로부터 입력 사건을 입력받으면, 외부 전이 함수에 따라 상태를 전이한다. 모델은 전이

된 상태에서 시간 전진 함수가 정의하는 시간만큼 머문 후, 내부 전이 함수에 따라 상태를 전이한다. 동시에 출력 함수를 통해 출력 사건을 외부로 내보낸다.

DEVS에서 대상 시스템 모델은 다수의 서브 모델이 계층적(hierarchical)으로 결합한 형태로 구성된다. 모델의 결합에 결합 DEVS 모델이 사용된다. 결합 DEVS 모델의 구조는 다음과 같다.

$$N = \langle X,\ Y,\ D,\ \{M_d\},\ \{I_d\},\ \{Z_{i,d}\},\ Select \rangle \qquad (2)$$

위의 식에서 X는 입력 사건의 집합, Y는 출력 사건의 집합, D는 컴포넌트 참조의 집합(set of component references), 각 $d \in D$에 대해, M_d는 DEVS 모델, 각 $d \in D \cup \{N\}$에 대해, I_d는 d의 영향자 집합(influencer set of d), 각 $i \in I_d$에 대해, $Z_{i,d}$는 출력 번역 함수(i-to-d output traslation function), $Select$: $2^D \rightarrow D$는 타이-브레이킹 함수(tie-breaking)이다.

2. Problem Statement

DEVS 시뮬레이션이 진행되면서, 서브 모델에서 출력된 이산 사건(discrete event)은 연결된 다른 서브 모델에게 입력 사건으로 전달된다. C/C++로 개발된 DEVS 구현물에서 모델 간에 주고받는 사건의 자료형이 주소(포인터)인 경우, 전송 메시지 변경이 발생할 수 있다. 예를 들어, Fig. 1과 같은 모델 예제에서, 모델 B의 외부 전이 함수인 delta_ext가 Fig. 2와 같다면 전송 메시지 변경이 발생한다.

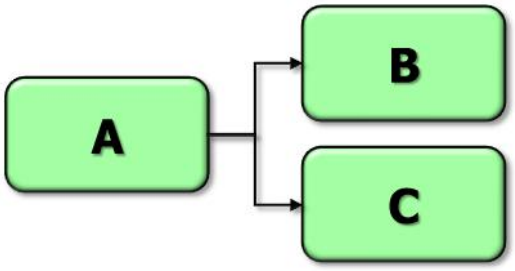

Fig. 1. Model Example

모델 A가 (사건에 대응하는 문자열이 저장된) 주소를 출력한 경우, DEVS 시뮬레이터는 우선순위에 따라 해당 주소를 연결된 모델에게 전달한다. 해당 주소가 모델 B로 먼저 전달된다고 가정하면, DEVS 시뮬레이터는 모델 B의 delta_ext를 호출하면서 경과 시간(elapsed time)과 해당 주소를 인자(argument)로 넘긴다. Fig. 2에서 B의 delta_ext는 받은 주소에 새로운 문자열을 복사한다. 이렇게 되면, 2가지 문제가 발생한다.

- 버퍼 오버플로(buffer overflow) : 버퍼 오버플로가 발생할 수 있다. 모델 A가 출력한 문자열은 해당 주소로 시작하는 버퍼 내에 있다. 만약 새로운 문자열의 크기가 해당 버퍼의 크기보다 크다면, 버퍼 오버플로가 발생한다. 버퍼 오버플로가 발생하면, 모델의 멤버 변수(내부 상태)가 오염(corruption)되거나 [18,19], 프로그램의 제어 흐름(control flow)이 변경될 수 있다[21]. 버퍼 오버플로는 완전히 별개의 주제로, 본 논문의 범위를 벗어난다.
- 전송 메시지 변경 : A가 출력하고 해당 주소에서 시작하는 문자열이 B에 의해 새로운 문자열로 변경된다. 참고로 키워드 const 때문에 해당 주소 자체를 변경할 수 없다. 모델 C는 A가 의도한 문자열이 아닌, B에 의해 변경된 문자열에 접근하게 된다. 참고로 원자 DEVS 모델의 δ_{ext}: $Q \times X \rightarrow S$이므로(즉, 외부 전이 함수가 외부 사건에 영향을 미치지 않으므로), 형식론 단계에서 전송 메시지 변경이 발생하는지 알 수 없다.

```
class B : public Atomic <char *>
{
 // 생략
 virtual void delta_ext (double e, const Bag <char *> & m)
 {
  char * p = m [0];
  ::strcpy (p, "All your model are belong to us");
  // 생략
 }
 // 생략
};
```

Fig. 2. Vulnerable Code Example

전송 메시지 변경은 모델 개발자의 실수나 고의(악의적인 개발자)로 발생할 수 있다. 전송 메시지 변경이 발생하면, 시뮬레이션 수행이 종료되거나 오염된 결과를 도출할 수 있다. 시뮬레이션 결과의 오염은 잘못된 의사결정으로 이어질 수 있다. 본 논문에서는 개발자가 시큐어 코딩을 준수하지 않더라도 전송 메시지 변경을 자동으로 탐지하는 것을 목표로 한다.

III. The Proposed Method

1. Overview

제안 기법에서는 출력 사건의 자료형이 포인터인 원자 DEVS 클래스 인스턴스가 출력 사건을 출력하면(DEVS 시뮬레이터가 인스턴스의 출력 함수를 실행한 후), 이의 사본(정확히는 포인터가 가리키는 주소에 있는 내용)을 저장한

다. 즉, 인스턴스가 메모리 주소를 출력 사건으로 출력하면, 해당 주소에 저장된 내용을 별도의 버퍼에 복사한다. 출력 사건이 연결된 DEVS 클래스 인스턴스에게 전달되어 입력 사건으로 처리된 이후(시뮬레이터가 연결된 인스턴스의 외부 전이 함수를 실행한 후), 출력 사건과 사본을 비교한다. 출력 사건으로 출력된 주소에 있는 내용과 버퍼에 있는 내용이 일치한다면, 전송 메시지 변경이 발생하지 않음을 의미한다. 만약 두 내용이 서로 다르다면, 전송 메시지 변경이 발생했음을 의미한다. 전송 메시지 변경이 발생하면, 해당 사실을 출력하거나 시뮬레이션을 종료할 수 있다.

Fig. 3은 구현 수준에서 제안 기법의 개요를 보인다. (1) DEVS 시뮬레이터는 내부 사건이 발생한 모델 A 인스턴스의 출력 함수를 호출한다. (2) 제안 기법이 적용된 DEVS 시뮬레이터는 모델 A 인스턴스에서 출력 사건을 백업하는 함수 postOutputFunction을 호출한다. (3) 출력된 사건은 연결 정보에 따라 모델 B 및 C 인스턴스에 전달되며, DEVS 시뮬레이터는 각 인스턴스의 외부 전이 함수를 호출한다. (4) 제안 기법이 적용된 DEVS 시뮬레이터는 모델 A 인스턴스에서 전송 메시지 변경 발생 여부를 확인하는 함수 preGarbageCollectionFunction을 호출한다. (5) 마지막으로 DEVS 시뮬레이터는 가비지 콜렉션(garbage collection) 함수를 호출한다.

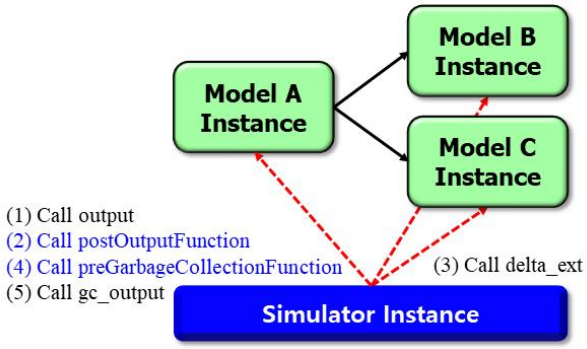

Fig. 3. Proposed Method Overview

제안 기법은 모든 개발자가 시큐어 코딩을 준수한다고 가정하지 않는다. 개발자가 입출력 사건의 자료형으로 포인터를 사용하고 실수로(혹은 고의로) 포인터가 가리키는 주소에 있는 내용을 변경하더라고 이를 탐지할 수 있다. 즉, 제안 기법이 포함된 라이브러리 또는 엔진만 사용해도 전송 메시지 변경을 탐지할 수 있다. C/C++로 작성된 다른 DEVS 구현물에도 적용할 수 있으나, adevs[16]에 우선 적용하였다.

2. Class Atomic of *adevs*

adevs에서 원자 DEVS 모델은 클래스 Atomic을 상속받아 구현한다. Fig. 4는 Atomic 일부로, 멤버 함수 delta_ext 및 output_func는 각각 외부 전이 함수 및 출력 함수다. 멤버 변수 x와 y는 모델이 입출력하는 사건이 저장된 백(bag)을 가리키는 포인터(즉, 백의 주소)이다.

```
template <typename X, typename T = double> class Atomic:
public Devs<X,T>
{
 // Omitted
 virtual void delta_ext (T e, const Bag <X> & xb) = 0;
 virtual void output_func (Bag <X> & yb) = 0;
 virtual void gc_output (Bag <X> & g) = 0;
 Bag <X> *x, *y;
 // Omitted
};
```

Fig. 4. Atomic Class of *adevs*

Fig. 5는 adevs에서 Fig. 1의 모델 간에 사건이 전송될 때의 메모리 블록을 나타낸 그림으로, Atomic 서브클래스 A, B, C의 인스턴스가 생성된 상태이다. 각 인스턴스가 생성될 때, adevs 시뮬레이터가 출력 사건을 저장하는 고유의 백을 외부에 생성하고, 멤버 변수 y에 고유의 백 주소를 저장한다. 만약 A의 인스턴스에서 내부 사건이 발생하면, adevs 시뮬레이터는 해당 인스턴스의 *y를 인자로 넘기면서 인스턴스의 output_func를 호출한다. A의 인스턴스와 연결된 B 및 C의 인스턴스가 연결되어 있으므로, adevs 시뮬레이터는 해당 인스턴스의 delta_ext를 차례로 호출한다. 이때 각 인스턴스의 x에 A의 인스턴스가 출력한 사건이 저장된 백의 주소를 저장하고, *x를 delta_ext의 인자로 넘긴다. 연결된 인스턴스의 delta_ext 호출이 완료되면, adevs 시뮬레이터는 A 인스턴스의 gc_output 함수를 호출하여, 고유의 백에 있는 사건을 삭제하도록 한다. gc_output에서 일반적으로 new 등의 연산자로 생성된 사건을 delete 등의 연산자로 삭제하여 메모리 누수(memory leak)를 방지한다. gc_output은 Atomic 서브클래스를 정의할 때 개발자가 직접 구현한다.

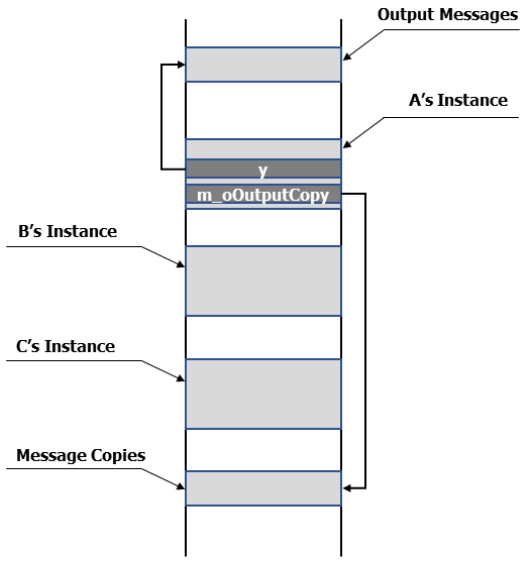

Fig. 5. Model Instances and Event Bags

3. Modification of Class Atomic

제안 기법에서는 출력 사건의 자동 사본 생성 및 확인을 위하여, Fig. 6과 같이 Atomic에 1개의 멤버 변수와 2개의 멤버 함수를 추가하였다. 멤버 변수 m_oOutputCopy는 출력 사건의 자료형이 포인터면 이의 사본을 저장하기 위한 백이다. 즉, 출력 사건들의 주소가 저장된 *y의 사본을 저장하기 위한 백이다. 멤버 함수 postOutputFunction는 adevs 시뮬레이터가 output_func를 호출한 직후 호출하는 함수로, 출력 사건의 자료형이 포인터면 사본을 m_oOutputCopy에 저장한다. 멤버 함수 preGarbageCollectionFunction은 adevs 시뮬레이터가 gc_output 호출 직전에 호출하는 함수로, 출력 사건의 자료형이 포인터면 m_oOutputCopy의 내용과 *y의 내용을 비교하여 전송 메시지 조작의 발생 여부를 확인한다.

```
template <typename X, typename T = double> class Atomic:
public Devs<X,T>
{
 // Omitted
 Bag <void *> m_oOutputCopy;                    // Proposed
 void postOutputFunction (void);                // Proposed
 void preGarbageCollectionFunction (void);      // Proposed
 // Omitted
};
```

Fig. 6. Model Instances and Event Bags

멤버 함수 postOutputFunction의 구현은 Fig. 7과 같다. postOutputFunction은 입출력 사건의 자료형(X)이 포인터인 경우에만 실질적으로 동작한다. 출력 사건의 백에는 사건 메시지가 저장된 주소 목록이 저장된다. 주소 목록을 출력 사건 사본의 백에 저장하는 형태로는 전송 메시지 변경을 탐지할 수 없다. 전송 메시지 변경은 주소를 변경하지 않으며, 주소가 가리키는 메모리 블록의 내용을 변경한다. 그러므로 postOutputFunction은 주소가 가리키는 메모리 블록과 같은 크기의 메모리 블록을 생성하고, 메모리 블곡의 내용을 복사한다. 표준 C++에서 포인터가 가리키는 메모리 블록의 크기를 알아낼 수 없다. 제안 기법에서는 포인터가 가리키는 메모리 블록의 크기를 알아내기 위해 윈도즈(Windows) 운영체제에 종속적인 함수 _msize를 사용하였다. 다른 운영체제로 이식(porting)하는 경우에는 해당 운영체제에서 지원하는 함수로 변경하면 된다. 예를 들어, 리눅스 운영체제로 이식하는 경우라면 함수 malloc_usable_size를 사용할 수 있다.

```
void Atomic <X, T>::postOutputFunction (void)
{
 if (std::is_pointer <X>::value)
 {
  int iBlockSize;
  void * pBlockCopy;
  for
  (
   typename Bag <X>::iterator pOutputIterator = y -> begin
();
   pOutputIterator != y -> end ();
   pOutputIterator ++
  )
  {
   iBlockSize = ::_msize (* pOutputIterator);
   pBlockCopy = new char [iBlockSize];
   ::memcpy (pBlockCopy, * pOutputIterator, iBlockSize);
   m_oOutputCopy . insert (pBlockCopy);
  }
 }
}
```

Fig. 7. Implementation of postOutputFunction

멤버 함수 preGarbageCollectionFunction의 구현은 Fig. 8과 같다. preGarbageCollrectoinFunction은 출력 사건의 백과 출력 사건 사본의 백을 비교한다. 두 백에 저장된 주소는 서로 다르지만, 해당 주소가 가리키는 메모리 블록의 내용은 일치해야 한다. 내용이 일치하지 않는 블록이 존재한다면, 전송 메시지 변경이 발생하였음을 의미한다. 전송 메시지 변경이 발생하면, 경고 메시지를 출력하고 시뮬레이션을 종료한다. 전송 메시지 변경 탐지 시의 동작은 응용에 따라 변경될 수 있다. 내용이 일치하는 블록에 대해서는 delete 연산자로 사본 메모리 블록을 해제하고 출력 사건 사본의 백을 비운다.

```
void Atomic <X, T>::preGarbageCollectionFunction (void)
{
 if (std::is_pointer<X>::value)
 {
  int iBlockSize;
  typename Bag <X>::iterator pOutputIterator = y -> begin
();
  typename Bag <void *>::iterator pCopyIterator =
m_oOutputCopy . begin ();
  for
  (
   ;
   pOutputIterator != y -> end ();
   pOutputIterator ++, pCopyIterator ++
  )
  {
   iBlockSize = _msize (* pCopyIterator);
   if (::memcmp (* pCopyIterator, * pOutputIterator,
iBlockSize))
   {
    std::cout << std::endl << "[ !! CAUTION !! ] SDEVS has
detected manipluation of a message!" << std::endl;
    ::exit (-1);
    }
   delete [] * pCopyIterator;
  }
  m_oOutputCopy . clear ();
 }
}
```

Fig. 8. Implementation of preGarbageCollectionFunction

4. Modification of *adevs* Simulator

Subsect. 3.3에서 설명한 함수 postOutputFunction과 preGarbageCollectionFunction을 자동으로 호출하기 위하여 adevs의 클래스 Simulator를 수정하였다. Fig. 9는 Simulator의 함수 visit으로, 내부 전이가 발생한 클래스 인스턴스의 출력 함수를 호출하고, 연결된 클래스 인스턴스에게 출력 사건의 백을 전달한다. 그림과 같이 출력 함수 output_func를 호출한 직후 postOutputFunction을 호출하여 자동으로 출력 사건 백을 백업하도록 하였다.

```
void Simulator <X, T>::visit (Atomic <X,T> * model)
{
 // Omitted
 model -> output_func (* (model -> y));
 model -> postOutputFunction (); // Proposed
 for
 (
  typename Bag <X>::iterator y_iter = model -> y -> begin
();
  y_iter != model -> y -> end ();
  y_iter ++
 )
 // Omitted
}
```

Fig. 9. Modification of Simulator::visit of *adevs*

Fig. 10은 Simulator의 함수 clean_up으로, 더 이상 사용되지 않는 출력 사건을 정리하도록 한다. 그림과 같이 출력 사건 가비지 컬렉션 함수 gc_output의 호출 직전에 preGarbageCollection을 호출하여 전송 메시지 변경이 발생했는지 자동으로 확인한다.

```
void Simulator <X ,T>::clean_up (Devs <X, T> * model)
{
 // Omitted
 if (amodel -> y != NULL)
 {
  amodel -> preGarbageCollectionFunction ();  // Proposed
  amodel -> gc_output (* (amodel -> y));
 // Omitted
}
```

Fig. 10. Modification of Simulator::clean_up of *adevs*

IV. Performance Evaluation

제안 기법의 성능 평가를 위해, 제안 기법이 적용된 adev와 적용되지 않은 adevs의 시뮬레이션 수행 시간을 비교하였다. Fig. 1의 간단한 모델을 기준으로, 1백만, 1천만, 1억 건의 사건이 생성될 때, 시뮬레이션 시작에서 종료까지 걸리는 시간을 측정하였다. Table 1은 실험환경을 요약한 표이다.

Table 2는 성능 비교 결과로, 제안 기법이 약 50%의 시간을 더 소요하는 것으로 측정되었다. 이는 제안 기법이 출력 사건이 발생할 때마다 동적으로 백업을 위한 메모리 블록을 할당하고 메시지 변경 여부 확인 후에 다시 메모리 블록을 해제(즉, 빈번한 동적 메모리 할당/해제)하기 때문이라 보인다. 이는 향후 풀(pool)을 사용하여 메모리 블록을 해제하지 않고 재사용함으로써 성능 향상을 이룰 수 있을 것으로 예상한다.

Table 1. Environment for Experiment

Description	Specifications
CPU	i7-10510U
RAM	16GB
Compiler	Microsoft Visual C++ 2019
Configuration	Release/x86

Table 2. Performance Comparison

No. of Events	Time to		O/H(%)
	adevs	Proposed	
1M	0.175	0.262	49.2
10M	1.668	2.545	53.7
100M	17.165	25.395	48.1

제안 기법으로 인한 비용이 작다고는 할 수 없지만, 제안 기법은 전송 메시지 변경을 방지한다. 즉, 전송 메시지 변경으로 인한 시뮬레이션 결과의 오염을 예방할 수 있다. 반면 제안 기법이 적용되지 않은 경우, 성능은 향상되나 전송 메시지 변경이 발생할 수 있다. 만약 전송 메시지 변경이 발생하면, 오염된 시뮬레이션 결과가 잘못된 의사결정으로 이어질 수 있다.

V. Conclusions and Future Work

본 논문에서는 C/C++ 기반 DEVS 구현물에서 전송 메시지 변경을 시큐어 코딩의 적용이나 구문 추가 없이 자동으로 탐지하는 기법을 제안하였다. 출력 사건이 출력된 직후, 해당 사건을 백업한다. 사건이 해제되기 직전, 출력한 사건과 백업한 사건을 비교한다. 두 사건이 불일치하면 전송 메시지 변경이 발생하였음을 의미한다. 제안 기법을 adevs에 적용하였다. 그러나 CD++[17] 등과 같은 다른 C/C++ 기반 DEVS 구현물에도 적용할 수 있다. 제안 기법의 적용으로 시뮬레이션 성능 자체는 떨어질 수밖에 없다. 그러나 제안 기법의 적용으로 전송 메시지 변경을 탐지할 수 있으며, 전송 메시지 변경으로 인한 시뮬레이션 결과 오염을 예방할 수 있다.

제안 기법의 현재 구현은 사건의 백업을 위해 동적 메모리 할당 및 해제를 반복하여 성능 저하가 발생한다. 성능 저하를 최소화하기 위하여, 메모리 블록 풀(pool) 등을 적용할 계획이다. 또한, 전송 메시지 조작 외의 포맷 스트링, 정수 오버플로 등 다양한 보안 약점 및 취약점에 관해서도 연구할 계획이다.

ACKNOWLEDGEMENT

This work was supported by the research grant of Cheongju University (2019.03.01.~2021.02.28.) **

REFERENCES

[1] A. Maria, “Introduction to modeling and simulation,” Proc. of Winter Simulation Conference (WSC), pp. 7-13, Atlanta, GA, USA, December 1997. DOI: https://doi.org/10.1145/268437.268440

[2] A.M. Law, *Simulation Modeling and Analysis*, 5th Ed., McGraw-Hill Education, 2014.

[3] K. Iwata, C. Miyakoshi, “A Simulation on Potential Secondary Spread of Novel Coronavirus in an Exported Country Using a Stochastic Epidemic SEIR Model,” J. of Clinical Med., vol. 9, no. 4, March 2020. DOI: 10.3390/jcm9040944

[4] D. Ivanov, “Predicting the impacts of epidemic outbreaks on global supply chains: A simulation-based analysis on the coronavirus outbreak (COVID-19/SARS-CoV-2) case,” Transportation Research Part E: Logistics and Transportation Rev., no. 136, April 2020. DOI: 10.1016/j.tre.2020.101922

[5] E. Hammad *et al*., “Implementation and development of an offline co-simulation testbed for studies of power systems cyber security and control verification,” Int’l J. of Elec. Power & Energy Syst., no. 104, pp. 817-26, January 2019. DOI: 10.1016/j.ijepes.2018.07.058

[6] P. Aggarwal *et al*., “HackIT: A Human-in-the-Loop Simulation Tool for Realistic Cyber Deception Experiments,” Advances in Intelligent Systems and Computing, vol. 960, pp. 109-12, June 2020. DOI: 10.1007/978-3-030-20488-4_11

[7] M. L. Cronqvist *et al*., “Development and Initial Validation of a Stochastic Discrete Event Simulation to Assess Disaster Preparedness,” Prehospital and Disaster Medicine, vol. 34, May 2019. DOI: 10.1017/S1049023X19002528

[8] D. Mourtzis, “Simulation in the design and operation of manufacturing systems: state of the art and new trends.” Int’l J. of Production Research, vol. 58, no. 7, pp. 1927-1949, April 2020. DOI: 10.1080/00207543.2019.1636321

[9] J.S. Carson, “Introduction to modeling and simulation,” Proc. of Winter Simulation Conference (WSC), pp. 9-16, Washington, DC, USA, December 2004. DOI: 10.1109/WSC.2004.1371297

[10] B. P. Zeigler *et al*., *Theory of Modeling and Simulation*, 3rd Ed., Academic Press, 2018.

[11] B.P. Zeigler, A. Muzy, “From Discrete Event Simulation to Discrete Event Specified Systems (DEVS),” IFAC-PapersOnLine, vol. 50, no. 1, pp. 3039-3044, July 2017. DOI: 10.1016/j.ifacol.2017.08.672

[12] M. Mokaddem *et al*., “DEVSServer: ambient intelligence and DEVS modelling-based simulation server for epidemic modelling,” Int’l J. of Simulation and Process Modelling, vol. 13, no. 6, pp. 557–581, October 2018. DOI: 10.1504/IJSPM.2018.095875

[13] J. Kim, H.J. Kim, “DEVS-based Modeling Methodology for Cybersecurity Simulations from a Security Perspective,” KSII Trans. on Internet and Infor. Syst., vol. 14, no. 5, pp. 2186-2203, May 2020. DOI: 10.3837/tiis.2020.05.018

[14] D.H. Lee *et al*., “A Unity-based Simulator for Tsunami Evacuation with DEVS Agent Model and Cellular Automata,” J. of Korea Multimedia Society, vol. 23, no. 6, pp. 772-783, June 2020. DOI: 10.9717/kmms.2020.23.6.772

[15] L. Rajaoarisoa, M. Sayed-Mouchaweh, “Adaptive online fault

diagnosis of manufacturing systems based on DEVS formalism," IFAC-PapersOnLine, vol. 50, no. 1, pp. 6825-6830, July 2017. DOI: 10.1016/j.ifacol.2017.08.1202

[16] J. Nutaro, adevs: A Discrete EVent System simulator, https://web.ornl.gov/~nutarojj/adevs/

[17] G. Wainer, CD++, http://cell-devs.sce.carleton.ca/

[18] H.Y. Lee, J.M. Kim, "Systematic Detection of State Variable Corruptions in Discrete Event System Specification Based Simulation," IEICE Trans. on Infor. & Syst., vol. E103-D, no. 7, pp. 1769-72, July 2020. DOI: 10.1587/transinf.2019EDL8219

[19] H.Y. Lee, "Buffer overflow detection in DEVS simulation using canaries," Proceedings of Winter Simulation Conference (WSC), pp. 4554-4555, Las Vegas, LV, USA, December 2017. DOI: 10.1109/WSC.2017.8248201.

[20] H.Y. Lee, "Potential Vulnerabilities in C/C++ Based Simulation," Proc. of Fall Conference of the Korea Society for Simulation, pp. 24-26, November 2017.

[21] R. Seacord, *Secure Coding in C and C++ (SEI Series in Software Engineering*), 2nd Ed., Addison-Wesley Professional, 2013.

Authors

Hae Young Lee received his B.S. degree in Electrical and Computer Engineering and Ph.D. degree in Computer Engineering from SKKU, in 2003 and 2009, respectively. He is an Assistant Professor of Digital Security at Cheongju University (CJU). Before joining CJU, he worked for several organizations, such as DuDu IT, Seoul Women's University, and Electronics and Telecommunications Research Institute (ETRI). His research interests include secure modeling and simulation, embedded system security, and security education.