Vol.5, No.12 pp.615~622 pISSN: 2287-5905 https://doi.org/10.3745/KTSDE.2016.5.12.615

Comparison of Path Exploration and Model Checking Techniques for Checking Automotive API Call Safety

Dongwoo Kim† · Yunja Choi††

ABSTRACT

Automotive control software can be a source of critical safety issues when developers do not comply system constraints. However, a violation is difficult to identify in complicated source code if not supported by an automated verification tool. This paper introduces two possible approaches that check whether an automotive control software complies API call constraints to compare their performance and effectiveness. One method statically analyzes the source code and explores all possible execution paths, and the other utilizes a model checker to monitor constraint violations for a given set of constraint automata. We have implemented both approaches and performed a series of experiments showing that the approach with model-checking finds constraint violations more accurately and scales better.

Keywords : Automotive Software, API, OSEK/VDX, Constraint Pattern, Static Analysis, Model Checking

차량전장용 소프트웨어의 API 제약사항 위배여부 탐지를 위한 실행경로 탐색방법과 모델검증 방법의 비교

김 동 우† · 최 윤 자††

요 약

차량전장용 제어 소프트웨어는 표준에 명시된 시스템 호출 제약사항을 위배할 경우 심각한 안전성 위협을 초래할 수 있다. 그러나 제약사항 위배는 실행경로가 복잡해질 경우 수동분석으로 색출하기 어렵고 테스팅을 통해 찾아내기도 어려워 이에 특화된 검증 방법이 필요하다. 본 연구에서는 차량전장용 제어 소프트웨어의 시스템 호출 제약사항 위배 여부를 효과적으로 검증하기 위한 두 가지 방법을 소개하고 그 효과를 실험적으로 비교하였다. 첫 번째 방법은 애플리케이션의 모든 가능한 실행경로를 탐색하고 각 경로의 제약사항 준수여부를 확인하는 방법이며, 두 번째 방법은 모델 검증 도구를 이용하여 애플리케이션이 오토마타로 표현된 제약사항을 위배하는 경우가 발생가능한지 확인하는 방법이다. 각 방법을 구현하고 실험한 결과 실행경로를 이용한 방법은 오탐을 유발하고 몇 가지 제약사항 위반을 놓치는 경우가 있는데 반해서 모델 검증을 이용한 방법은 오탐이 없었으며 비교적 큰 애플리케이션을 대상으로 보다 빠른 시간 내에 검증을 수행할 수 있음을 보였다.

키워드 : 차량전장소프트웨어, 애플리케이션 프로그래밍 인터페이스, OSEK/VDX, 제약사항 패턴, 정적분석, 모델검증

1. 서 론

차량전장용 제어 소프트웨어의 개발은 OSEK/VDX[1]와 같은 국제표준을 준수해야 하며, 이들 표준에는 전장용 소프트웨어가 준수할 시스템 서비스 호출 제약사항들이 포함되어 있다. 시스템 서비스 호출 제약사항은 위배될 시에 인명피해와 같은 심각한 문제를 유발할 수 있으며, 제약사항 준수여부의 선제적 검증은 시스템의 전반적인 안전성 보장을 위해 반드시 필요한 작업이다. 하지만 제약사항 위반은 대부분 디버깅 단계에서 색출될 수 없는 의미적 오류로써 프로그램의 크기가 커질수록 복잡해지는 제어 경로를 모두 고려한 수동 분석은 비용 및 효율의 측면에서 실용적이지 못하다.

소프트웨어 검증에 주로 사용되는 동적 테스트 기법도 내장형 소프트웨어에 적용하기에는 여러 가지 어려움이 있다. 특히, 다양한 하드웨어 플랫폼에 이식되어 하드웨어와 함께 동작하기 때문에 코드가 모두 완성되어 이식되기 전에는 테스팅을 수행할 수 없다는 단점이 있다.

본 연구에서는 이러한 문제점들을 해결하고 하드웨어에 독립적인 소프트웨어 제어논리의 독립적인 검증을 자동화하기 위한 두 가지 접근방식을 제시하고 검증능력과 성능의

※ 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학ICT연구센터 육성지원사업의 연구결과로 수행되었음(IITP-2016-H8601-16-1002, IITP-2016-H8501161012000l1002).
† 준 회 원 : 경북대학교 컴퓨터학부 석사과정
†† 정 회 원 : 경북대학교 컴퓨터학부 부교수
Manuscript Received : July 4, 2016
First Revision : October 12, 2016
Accepted : October 16, 2016
* Corresponding Author : Yunja Choi(yuchoi76@knu.ac.kr)

Table 1. OSEK/VDX API Call Constraint

No.	API Function	Constraint Description	Category
1	ChainTask TerminateTask	Ending a task function without a call to TerminateTask or ChainTask is strictly forbidden and may leave the system in an undefined state.	call sequence based
2	GetResource ReleaseResource	A critical section shall always be left using ReleaseResource.	call sequence based
3		It is not allowed to use services which are points of rescheduling for non preemptable tasks (TerminateTask, ChainTask, Schedule and WaitEvent) in critical sections.	call sequence based
4		Should not attempt to release a resource which has a lower ceiling priority than the statically assigned priority of the calling task or ISR.	Configuration
5	WaitEvent	This service shall only be called from the extended task owning the event.	Configuration
6	SetEvent	An Event of a task can be set only if a task was waiting for it by calling WaitEvent	call sequence based
7		The referenced task shall be an extended task.	Configuration
8	ClearEvent	The system service ClearEvent is restricted to extended tasks which own the event.	Configuration

측면에서 실험적인 비교 분석을 수행하였다. 첫 번째 방식은 차량전장용 소프트웨어의 실행경로를 제어흐름도 분석을 통해 탐색하여 각 실행경로가 제약사항을 위배할 가능성이 있는지를 판별하는 방식[2]으로 정형분석에 대한 지식이 없는 경우에도 누구나 적용할 수 있는 방식이다. 두 번째 방식은 C코드 대상 모델검증기인 CBMC[3]를 활용하여 차량전장용 소프트웨어가 오토마타의 형식으로 작성된 제약사항을 위배할 가능성이 있는지 모델검증기술을 이용하여 검사하는 방식이다. 이 방식은 모델검증에 대한 지식이 필요한 대신, 소스코드에 대한 별도의 분석없이 간단한 코드대체 및 삽입으로 구현될 수 있다는 장점이 있다.

실험 결과 모델검증을 이용한 방법(85.88%)이 실행경로 탐색을 이용한 방법(64.71%)에 비해 보다 많은 제약사항 위반을 탐지할 수 있음을 확인하였으며, 소스코드의 크기 및 분기문 수의 증가에 따른 검증비용도 상대적으로 완만하게 증가 함을 보였다.

본 논문은 다음과 같이 구성된다. 제 2장에서 차량전장용 소프트웨어 및 준수할 제약사항들에 대한 간략한 소개에 이어, 제 3장에서는 자연어로 기술된 API 호출상의 제약사항을 컴퓨터가 이해할 수 있는 오토마타 형식으로 표현하는 방법에 대해서 설명한다. 제 4장과 5장에서는 실행경로 탐색방법을 이용한 API 호출상의 제약사항 위반 탐지방법 및 모델 검증을 이용하여 API 호출 제약사항 위반 탐지법을 각각 설명한다. 제안된 방법들에 대한 구현 및 실험결과가 제 6장에서 설명되고, 관련연구(7장) 및 향후 연구에 대한 토의(8장)으로 결론을 맺는다.

2. 차량전장용 소프트웨어의 개발과 제약사항

2.1 OSEK/VDX 운영체제와 응용프로그램 개발

OSEK/VDX은 차량전장용 시스템에 대한 산업 표준으로 운영체제, 통신, 네트워크에 대한 표준을 정의하고 있다. OSEK/VDX 표준을 따라 제작된 운영체제는 작업 스케줄링, 자원 관리, 이벤트 관리 등의 기본적인 서비스를 제공하며 차량의 계산기기(ECU)마다 탑재되어 응용 프로그램과 하드웨어 간의 중계 역할을 수행한다.

Fig. 1은 OSEK/VDX 기반 운영체제의 구성 요소와 차량전장용 소프트웨어의 생성 과정을 나타낸다. OSEK/VDX 기반 운영체제는 표준에 따라 C언어나 어셈블리어로 작성되며, 차량전장용 응용 프로그램은 운영체제에서 제공하는 API를 이용하여 작성된다. 이렇게 작성된 응용 프로그램은 운영체제의 소스코드와 함께 컴파일되어 하드웨어 플랫폼에 이식된다.

차량전장용 응용 프로그램은 여러 개의 TASK로 구성되며 환경설정에 기술된 우선순위가 높은 순서대로 TASK를 수행한다. TASK는 이벤트 소유 유무에 따라 이벤트가 없으면 standard TASK, 이벤트가 있으면 extended TASK로 분류된다. 이벤트를 소유하지 않은 TASK는 이벤트 관련 API를 호출할 수 없다.

2.2 OSEK/VDX의 시스템 호출 서비스 제약사항

OSEK/VDX 시스템 서비스 호출 제약사항이란 OSEK/VDX 운영체제에서 제공하는 26개의 API 함수를 사용하는데 있어서 준수되어야 할 규칙이다. OSEK/VDX 운영체제 표준은 각 API 함수마다 제약사항을 명시적으로 기술한다. Table 1은 OSEK/VDX 운영체제에서 제공하는 API 함수와 이에 관련된 제약사항 중 일부를 정리해 놓았다. Table 1의 제약사항 1번은 TASK의 종료를 알리는 API 함수인 TerminateTask에 관한 제약사항이다. 이 제약사항에는 “TerminateTask 함수가 호출된 후에는 어떠한 다른 API 함수도 호출되지 않아야 된다.”고 명시한다.

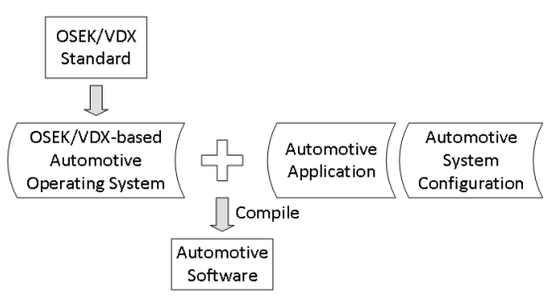

Fig. 1. OSEK/VDX Application Development

```
00: // Application code 00: // Configuration
01: Task(t1) {         01: Task t1 {
02:   ActivateTask(t2); 02:   PRIORITY = 3;
03:   WaitEvent(e1);   03:   // no resource
04:   …                04:   EVENT = e1;
05:   TerminateTask(); 05:   EVENT = e2;
06: }                  06:   AUTOSTART = yes
07: Task(t2) {         07: };
08:   …                08: Task t2 {
09:   SetEvent(t2, e1); 09:   PRIORITY = 1;
10:   if(condition)    10:   RESOURCE = r1;
11:     ClearEvent(e2); 11:   // no event
12:   TerminateTask(); 12: };
13: }                  13: Resource r0 {};
                       14: Resource r1 {};
                       15: Event e1 {};
                       16: Event e2 {};
```

Fig. 2. Example of Automotive Application

Table 1의 제약사항은 정의되는 범위에 따라 지역 제약사항과 전역 제약사항으로 분류된다. 하나의 TASK 안에서만 정의되는 제약사항을 지역 제약사항(1,2,3,4,5,7,8)이라 하고, 여러 개의 TASK와 연관하여 정의되는 제약사항을 전역 제약사항(6)이라 한다.

Fig. 2는 제약사항을 위반하는 OSEK/VDX 기반 차량전장용 애플리케이션과 환경설정을 나타낸다. 이 소프트웨어는 2개의 TASK t1, t2와, 2개의 리소스 r0, r1, 2개의 이벤트 e0, e1으로 구성된다. TASK t1은 두 이벤트를 모두 소유하며, TASK t2는 이벤트를 소유하지 않는다. TASK t2는 SetEvent, ClearEvent, TerminateTask를 순서대로 호출가능하며 ClearEvent 호출 시 자신이 소유하고 있지 않는 이벤트를 삭제한다. Table 1의 제약사항 8번에 따르면 standard Task가 ClearEvent를 호출해서는 안된다고 명시되어 있으며, 이러한 제약사항의 위배는 시스템 실패로 이어질 수 있음을 선행연구에서 입증하였다[4].

선행 연구에서는 OSEK/VDX 기반 오픈소스 운영체제인 Trampoline[5]을 대상으로 실험한 결과 적합성 테스트에서 발견하지 못한 네 가지 종류의 시스템 실패(failure) 시나리오를 발견하였으며, 발견한 모든 시나리오는 제약사항을 위배하는 시스템 호출 시퀀스로부터 발생함을 알 수 있었다[4]. 네 가지 시나리오는 다음과 같다.

- 운영체제의 rescheduling을 유발하는 API 함수 호출 이후에 자원의 할당과 해제를 하는 API 함수가 짝이 맞지 않는 경우
- 이벤트를 소유하지 않은 TASK가 WaitEvent를 호출한 경우
- SetEvent와 WaitEvent간에 짝이 맞지 않는 경우
- 이벤트를 소유하지 않은 TASK가 ClearEvent를 호출한 경우

이와 같은 잠재적 오류를 줄이기 위해서는 개발 단계에서부터 제약사항 위배 여부 검증 과정이 선행되어야 한다.

3. 제약사항 패턴

선행 연구[4, 6]에서는 자연어로 작성된 OSEK/VDX 운영체제의 제약사항을 1) 호출 시퀀스에 관련된 제약사항, 2) 설정에 관련된 제약사항, 3) 상태에 관련된 제약사항으로 분류하고 패턴화 하였으며, 본 연구에서는 호출 시퀀스와 설정에 관련된 제약사항을 우선적으로 검증하였다. 상태에 관한 제약사항은 운영체제 모델이 존재하여야 정확한 검증이 가능하므로 본 연구의 범위에서 제외하였다.

Definition 1. 호출 시퀀스와 관련된 제약사항 패턴

1. InPairs(f1, f2) : f1 함수와 f2 함수는 짝을 이루어서 호출된다.
2. MustEndWith(A) : Task는 A에 속하는 함수를 호출한 후 종료한다.
3. Limited(f,n) : 함수 f는 n회 이상 호출되지 않는다.
4. NotInBetween(f1,f2,A) : f1 함수와 f2 함수 사이에서는 A에 속하는 함수가 호출되지 않는다.

예를 들어, Table 1의 제약사항 1, 2, 3, 6은 호출 시퀀스와 관련된 제약사항 패턴으로 패턴화할 수 있으며, 예를 들어 제약사항 1은 MustEndWith({TerminateTask, ChainTask})로 패턴화 된다.

Definition 2. 환경설정과 관련된 제약사항 패턴

1. CallerMode(f,m) : API f를 호출하는 Task의 mode는 m이다.
2. CalleeMode(f,m) : API f가 매개변수로 사용하는 Task의 mode는 m이어야 한다.
3. CallerType(A,t) : API f∈A를 호출하는 Task의 type은 t이다.
4. OwnerOnly(f) : API f가 매개변수로 사용하는 Event를 Task가 소요하고 있다.
5. CeilingPriority(f) : API f가 매개변수로 사용하는 Resource의 priority는 API f를 호출하는 Task의 Priority보다 크거나 같다.

Table 1의 제약사항 4, 5, 7, 8은 환경설정과 관련된 제약사항 패턴으로 표현할 수 있으며, 제약사항 8은 CeilingPriority(ReleaseResource)로 패턴화 할 수 있다.

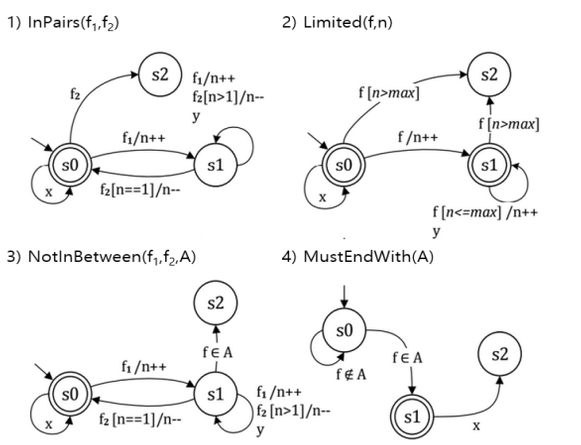

Fig. 3. Constraint Automata

Fig. 3은 제약사항 패턴을 푸시다운 오토마타의 형태로 나타낸 모습이다. 제약사항 패턴은 자유문법이나 정규문법으로 정의될 수 있으며 푸시다운 오토마타의 형태로 정형화될 수 있다. 예를 들어 Table 1의 제약사항 1번은 Figure 3의 4번에 해당하는 MustEndWith 푸시다운 상태기계로 나타낼 수 있으며 {TerminateTask, ChainTask}에 해당하는 API 함수가 호출될 경우 수용 상태인 s1으로 상태가 천이되고 그 이후에 다른 API 함수가 호출될 경우 오류상태인 s2로 천이된다.

4. 실행경로 탐색방법을 이용한 제약사항 검증

차량전장용 애플리케이션이 API 함수 호출 제약사항을 준수하는지 자동검증하기 위한 첫 번째 방법은 소프트웨어를 정적분석하여 가능한 실행경로를 모두 탐색하고, 각 실행경로와 제약사항 오토마타 간 비교 수행 시 제약사항 오토마타가 비정상 상태에 도달하는 경우가 발생하는지 확인하는 방법이다.

Fig. 4는 *실행경로 탐색방법을 이용한 제약사항 검증방법* [2]을 도식화한 것이다. 실행경로 탐색방법은 총 4단계로 구성되어, 1) OSEK/VDX 기반 차량전장용 애플리케이션을 이용하여 제어흐름그래프(Control Flow Graph)[7] 생성, 2) 제어흐름그래프로부터 실행경로 탐색, 3) 실행경로로부터 API 호출 시퀀스 추출, 그리고 4) 제약사항 검증을 수행한다.

첫 번째 단계인 *제어흐름그래프 생성*단계에서는 차량전장용 애플리케이션을 제어흐름그래프의 형태로 표현하며, 제어흐름그래프의 정의는 다음과 같다.

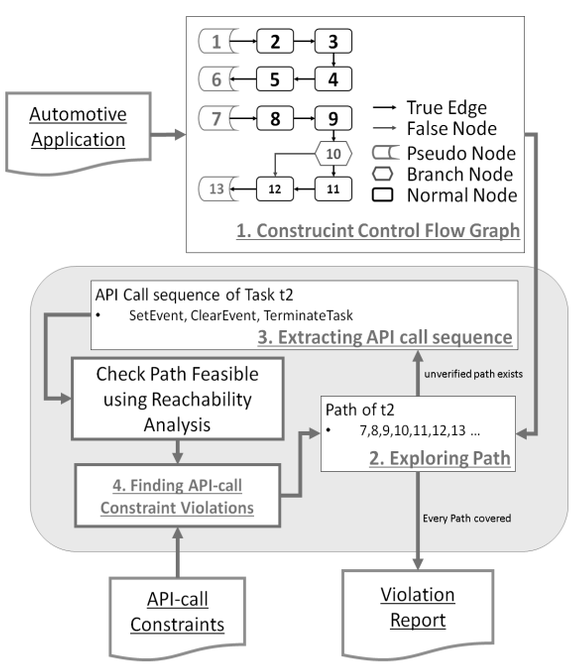

Fig. 4. Constraint Verification Using Path Explore Method

Definition 3. 제어흐름그래프 G=(N,T,L)는 다음의 조건을 만족하는 3-tuple 이다.

- N은 제어흐름그래프 노드의 집합,
- n∈N은 애플리케이션의 각 문장
- T=N×N는 제어흐름그래프의 전이집합이며 애플리케이션의 제어전환을 나타낸다.
- L: N -> API은 제어흐름그래프의 각 노드에서 호출되는 API 호출을 추출하는 함수이다.

정의된 제어흐름 그래프에서 실행경로를 탐색하기 위하여 다음과 같은 함수를 활용하였다.

- next(n)={n`|t=(n,n`) and t∈T}: 노드 n의 다음 노드들의 집합
- length(p) : 하나의 실행경로에 속한 노드들의 수

두 번째 단계인 *실행경로 탐색단계*에서는 제어흐름그래프를 깊이 우선 탐색하여 제어흐름그래프의 시작 노드부터 끝 노드까지 이어지는 노드 시퀀스를 탐색한다. 실행경로는 다음 알고리즘을 이용하여 구할 수 있다.

```
01: ExplorePath (Node n, Path p) {
02:     p.attach(n);
03:     if ( length(p)==DEPTH || next(n)==empty ) {
04:         verify(p); }
05:     else {
06:         for each m of next(n) {
06:             ExplorePath(m, p); } }
07:     p.detach(n); }
```

DEPTH는 실행경로의 길이제한이다. ExplorePath 함수는 현재 탐색하고자 하는 노드 n과 탐색 중인 실행경로 p를 입력받으며 각 노드 n에 방문할 때마다 p의 마지막에 n을 추가한다. p의 길이가 충분히 길어지거나 n의 다음 노드가 존재하지 않는다면 verify(p)를 이용하여 실행경로를 검증한다. ExplorePath 마지막에는 path p에 추가한 노드 n을 제거해 줌으로써 n을 포함한 탐색이 끝난다.

세 번째 단계와 네 번째 단계는 verify(p)를 호출할 때 수행되며, 세 번째 단계에서는 path p로부터 API 호출 시퀀스 $L(p) = \bigcup_{i=1}^{length(p)} L(n_i)$를 추출한다.

마지막으로 네 번째 단계인 *제약사항 검증 단계*에서는 앞서 추출한 API 호출 시퀀스를 제약사항 오토마타의 입력시퀀스로 적용한 결과가 제약사항 오토마타를 안전상태에 이르게 하는지 확인한다. 제약사항 오토마타가 안전상태에 이르지 않을 경우 API 호출 시퀀스와 제약사항 위반이 유발된 위치를 기록하여 제약사항 위반 보고서를 생성하며 완성된 제약사항 위반 보고서는 사용자에게 전달된다.

예를 들어, CallerMode(ClearEvent, Extended) 제약사항 오토마타에 t2의 API 호출 시퀀스 중 하나인 SetEvent, ClearEvent, TerminateTask를 제약사항 오토마타의 입력시퀀스로 입력하면 두 번째 호출된 API인 ClearEvent를 입력

할 때에 제약사항 위반을 탐지할 수 있다. 이후 나머지 API 호출 시퀀스를 마저 입력한 후에도 안전상태에 이르지 않는다. 제약사항 위반 보고서에는 제약사항을 위반한 t2의 실행경로와 제약사항을 위반한 API 호출(2번째 API 호출인 ClearEvent)을 기록한다.

5. 모델 검증을 이용한 제약사항 검증

5.1 C 모델 검증 도구 CBMC를 이용한 모델 검증

모델 검증[8, 9] 기술은 검증대상인 시스템을 수학적으로 모델링하고, 해당 시스템이 요구되는 속성을 만족하는지 엄밀하게 검사하는 기법이다. 하드웨어 디자인이나 소프트웨어 알고리즘 등을 검증대상으로 모델링하며 불변식, assert, 시제논리 등을 이용하여 시스템 속성을 표현한다. 모델 검증 수행 결과 검증대상이 시스템 속성을 만족한다는 결과가 나온 경우 해당 검증대상은 어떠한 경우에도 시스템 속성을 위반하지 않음을 알 수 있으며 이러한 특징은 안전성 검증에 주로 활용되어 특정 시스템이 시스템 실패를 유발하는지 교착 상태에 도달하지는 않는지 확인하는데 사용된다.

CBMC[3]는 검증대상과 시스템 속성을 부울 대수식으로 변경하고 충족가능성(satisfiability)을 확인함으로써 검증대상이 시스템 속성을 만족하는지 확인하는 모델 검증 도구이다. CBMC는 assert문이 실행되기 위한 도달 조건을 수집한 후 assert 조건과 비교하여 assert 문장이 위반될 수 있는 실행경로를 찾는다.

5.2 CBMC를 이용한 제약사항 위반 검출 방법

모델 검증을 이용한 제약사항 위반 검출은 검증대상 소프트웨어에서 호출하는 API 함수들을 제약 사항 오토마타의 관점에서 모니터링하는 라이브러리 함수들을 작성하고 시스템 속성을 assert 구문으로 코드에 삽입하여 모델검증을 수행한다. Fig. 5는 *모델 검증을 이용한 제약사항 위반 검출 방법*을 나타내며 이는 3단계로 구성된다. 각각의 단계에서는 1) 제약사항을 이용한 C언어 형태의 제약사항 오토마타 라이브러리 생성, 2) 기존의 OSEK/VDX 기반 애플리케이션의 annotation, 3) 검증 모델 생성 및 모델 검증을 수행한다.

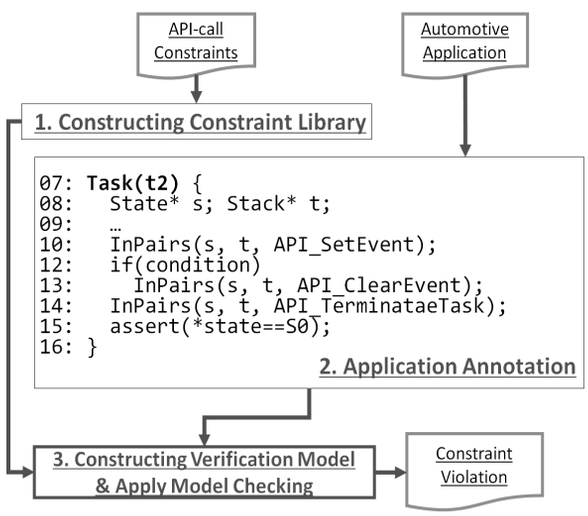

Fig. 5. Constraint Verification Using Model Checking

첫 번째 단계인 *제약사항 라이브러리 생성단계*에서는 제약사항 오토마타를 C언어의 형태로 나타낸다. 예를 들어 Table 1의 제약사항 4번은 GetResource와 ReleaseResource가 서로 짝을 맞추어 호출되어야 한다는 내용이며 이는 InPairs (GetResource, ReleaseResource)로 나타낼 수 있다. 제약사항 오토마타로 표현하면 Fig. 6과 같다.

Fig. 6은 제약사항 오토마타의 상태를 나타내는 변수와 Stack을 나타내는 변수(제약사항 오토마타는 푸시다운 오토마타이기 때문에 Stack이 필요), API 호출을 매개변수로 입력받고 제약사항 오토마타의 다음 상태를 결정한다. 예를 들어 현재 상태가 S0이고 API 호출이 ReleaseResource인 경우 제약사항 오토마타의 다음 상태는 S2이다.

두 번째 단계인 *소스코드 annotation 단계*에서는 1) 검증특성 삽입, 2) 제약사항 오토마타를 위한 변수 선언, 3) API 호출문 변경을 수행한다. 검증특성 삽입 단계에서는 Task의 마지막에 제약사항 오토마타가 안전상태(safe state)에서 종료된다는 속성을 assert 문장의 형태로 삽입한다. 제약사항 오토마타 변수 선언 단계에서는 제약사항 오토마타의 상태를 나타내는 변수와 Stack을 나타내는 변수를 Task의 시작 지점(entry point)에 선언한다. API 호출문 변경단계에서는 각각의 API 호출문을 제약사항 라이브러리 호출로 변경한다. 예를 들어 SetEvent 호출문을 InPairs(state, stack, API_SetEvent)와 같이 변경한다.

세 번째 단계인 *검증 모델 생성 및 검증 수행 단계*에서는 앞서 생성한 제약사항 라이브러리와 annotated 소스코드를 통합하고 CBMC로 모델 검증한다. 제약사항 위반을 발견하면 CBMC에서 도출한 assert 문장을 위반한 실행경로를 이용하여 제약사항 위반 보고서를 생성한다.

```
01: #define API_f1 API_GetResource
02: #define API_f2 API_ReleaseResource
03: void Inpairs(State* state, Stack* stack, Call call) {
04:   if ( *state == S0 ) {
05:     if ( call.api == API_f1 ) {
06:       *state=S1; push(stack, call); }
07:     else if ( call.api == API_f2 ) {
08:       *state=S2; } }
09:   else if ( *state == S1 ) {
10:     if ( call.api == API_f1 ) {
11:       push(stack, call);
12:     else if ( call.api == API_f2 ) {
13:       peek(stack).param == call.param ?
              pop() : *state=S2;
14:       if ( empty(stack) == TRUE ) {
15:         *state=S0; } } } }
```

Fig. 6. Constraint Automata Library - InPairs (GetResource, ReleaseResource)

6. 구현 및 실험

6.1 구현

본 연구를 통해 생성한 두 검증도구는 자바 언어로 구현하여 웹 기반 환경 및 플러그인 환경 등 어떤 환경에서도 검증을 수행할 수 있도록 설계되었다. 실행경로 탐색을 이용한 검증방법에서 사용한 제어흐름그래프는 선행 연구에서 개발한 정적분석도구인 CodeAnt[7]의 기능을 활용하였다. CodeAnt는 사용자가 작성한 애플리케이션 코드를 입력 받은 후 제어흐름그래프를 생성하였으며, 생성한 제어흐름그래프를 상기한 알고리즘으로 탐색하여 실행경로를 추출하였다. 반복문의 수행횟수에 따라 무한히 많은 실행경로가 생성되는 것을 방지하기 위해 탐색과정에서 반복문의 최대 실행횟수를 제한하였다. 제약사항 위배 여부 확인은 각각의 실행경로와 제약사항 오토마타를 비교함으로써 확인하였으며 검증 결과 보고서를 생성하여 제약사항을 위반한 실행경로와 제약사항 위반이 발생된 위치를 기술하였다.

모델 검증을 이용한 검증 방법에서 사용한 제약사항 라이브러리는 수작업으로 생성하였으며 annotation을 수행할 때는 java.regex의 문자열 패턴 매칭 모듈을 활용하여 API 호출문을 제약사항 라이브러리 호출문으로 변경하였다. 제약사항 라이브러리와 annotation이 이루어진 애플리케이션은 함께 CBMC(C Bounded Model Checker)에 입력되어 제약사항 위반을 탐색하는데 사용되었으며, 검증 결과 보고서는 CBMC에서 출력하는 실행경로를 이용하여 생성하였다.

6.2 실험

본 연구에서는 각 검증 방법의 성능을 비교하기 위해 다음 두 가지 의문점에 관한 실험을 하였다.

- RQ1. 다양한 제약사항 위반을 얼마나 탐지하는가?
- RQ2. 애플리케이션의 크기에 따라 얼마나 많은 검증 시간을 필요로 하는가?

첫 번째 실험에서는 다양한 제약사항을 위반하는 애플리케이션을 얻기 위해 선행연구에서 개발한 TSG(Test Sequence Generator)를 이용하였다. TSG는 제약사항 오토마타의 각 상태나 전이를 포함하는 애플리케이션을 생성한다. 본 연구에서는 TSG를 이용하여 15개의 검증 대상을 생성하였으며 수작업으로 분석한 결과 총 85개의 제약사항 위반을 확인하였다. 모델 검증을 이용한 방법은 이 중 73개(85.88%)를 탐지하였으며, 실행경로 탐색방법은 55개(64.71%)를 탐지하였다. 모델 검증에서 탐지하지 못한 제약사항 12개는 Table 1의 제약사항 6번이며 이 제약사항은 전역 제약사항이므로 이를 검증하기 위해서는 검증대상에 운영체제가 포함되어 있어야한다. 하지만 두 도구는 Quick & Easy 검증을 진행하기 위해서 운영체제를 검증대상에 포함하지 않아 이 제약사항 위반을 탐지해 낼 수 없었다.

두 번째 실험에서는 다양한 크기의 검증대상을 생성하기 위해서 랜덤 애플리케이션 생성기를 구현하여 검증대상을 생성하였다. 이 도구는 OSEK/VDX API, 분기문(while, if), 할당문(assignment) 등을 임의 선택하여 애플리케이션을 생성한다. 본 실험에서는 API/while/if/할당문을 각각 25%의 확률로 생성하도록 하였으며 최대 3%의 while과 if를 가지도록 설정하였다. 각 방법에서 검증을 수행할 때에는 실행경로의 길이를 10000으로 한정하고 진행하였으며 실행경로를 이용한 방법에는 반복문의 최대 수행횟수를 1, 모델 검증을 이용한 방법에는 100을 허용하였다.

Fig. 7은 실행경로 탐색방법과 모델 검증을 이용하여 검증을 수행하였을 때 소요된 시간을 그래프로 나타낸 모습이다. 실행경로 탐색방법을 이용하여 검증을 수행한 결과(점선) 소스코드의 크기가 500 LOC 보다 작을 때는 모델 검증을 이용한 방법에 비해 적은 시간을 소요하지만 소스코드의 크기가 커짐에 따라 소요시간이 급진적으로 증가함을 알 수 있었으며 1300 LOC 크기의 애플리케이션은 13시간의 검증 작업에도 검증이 끝나지 않았다. 모델 검증을 이용한 방법(실선)은 소스코드의 크기가 600 LOC 이상인 소스코드를 대상으로 효율적이었으며 검증대상의 크기가 커지더라도 안정적으로 작동하였다.

메모리 사용은 두 도구 모두 비슷하였다. 실행경로 탐색방법에서는 Java 가상 머신에서 1.06GB를 소요하였으며 모델 검증을 이용한 방법에서는 Java 가상 머신에서 1.03GB, CBMC 프로세스에서 0.4GB 정도를 소요하였다.

본 연구팀은 추가적으로 실행경로 탐색방법을 이용하여 추출한 실행경로 수에 따른 검증시간을 측정해 보았다. Fig. 8은 Path 수에 따른 검증시간을 나타내며, 두 축은 모두 Log 스케일이다. 이 방법은 100,000개의 Path까지는 어느 정도 급진적으로 시간이 증가하다가 이후에는 일정하게 증가함을 볼 수 있다. 실행경로가 100,000개 보다 많을 정도로 큰 검증 대상은 각각의 실행경로가 depth 제한에 한정되어 길이가 10000을 넘지 않기 때문에 시간 소요가 일정하게 증가하는 것으로 보인다.

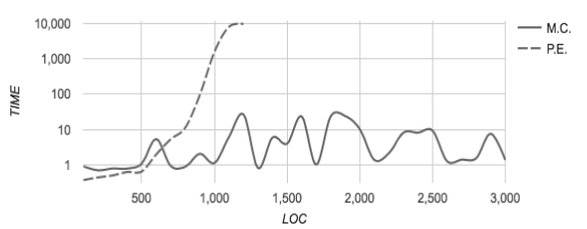

Fig. 7. Experiment Result w.r.t. Size of Application

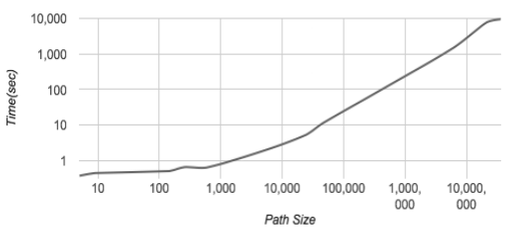

Fig. 8. Verification Time w.r.t Path Size Using Path Explore Method

모든 실험은 OS X ElCaptain 10.11.4를 탑재한 iMac에서 진행되었으며 CPU는 3.4GHz Intel Core i5, 메모리는 16GB 1600 MHz DDR3이다.

7. 관련 연구

본 연구와 가장 관련이 있는 연구로는 Typestate Analysis [10-12]와 SLAM[13, 14]이 있다. Typestate Analysis는 소프트웨어가 호출하는 함수/메서드의 시퀀스가 오류상태에 도달하는 경우가 있는지 확인하는 기법이다. 이 기법에서는 함수/메소드의 시퀀스가 프로그램을 오류상태에 이르게 하는지 확인하기 위해서 Typestate라고 불리는 오토마타를 활용한다. 이 방식은 시스템을 검증하기 위해서 사용자가 알맞은 함수/메소드 시퀀스를 직접 작성해야 하는 단점이 있는 반면에 본 연구에서 제안한 방법은 사용자가 직접 작성하기 어려운 제약사항 오토마타를 미리 입력해 두고 라이브러리로 활용함으로써 사용자의 불편을 덜어주었다.

SLAM은 마이크로소프트에서 개발한 디바이스 드라이버 검증모듈이다. 이 도구는 API 사용 규칙과 API stub을 사용자가 작성한 디바이스 드라이버에 인코딩하고 모델 검증 기술을 활용하여 API 사용 규칙 준수 여부를 확인한다. API 사용 규칙은 함수 형식으로 인코딩되며 디바이스 드라이버가 규칙을 준수하지 않는다고 판단되는 경우 인코딩된 함수는 error()를 호출한다. SLAM은 error()에 대한 도달 조건 분석을 수행함으로써 디바이스 드라이버가 API 사용 규칙을 준수하는지 확인한다. SLAM에서는 각 API 사용 규칙을 개별적인 함수로 표현하였으나, 본 연구에서는 제약사항 패턴을 사용하여 필요한 라이브러리 함수를 최소화하였다는 차이가 있다.

8. 결론 및 향후 연구

본 연구에서는 OSEK/VDX 기반 차량전장용 소프트웨어가 표준에 기술된 제약사항을 준수하는지 검증하는 두 가지 방법을 소개하였으며 각각의 방법을 이용하여 검증을 수행하는 두 도구를 개발하고 성능을 비교하였다.

본 연구에서 고안한 각 방법의 검증 능력을 측정하기 위해 1) 제약사항 탐지 능력을 측정하는 실험과 2) 제약사항 검증 시간을 측정하는 실험을 설계하였다. 실험 결과 모델 검증을 이용한 방법이 실행경로 탐색을 이용한 방법에 비해 보다 많은 제약사항(85.88% 대비 65.71%)을 탐지할 수 있음을 확인하였고, 실행경로 탐색방법에 비해 크기가 큰 애플리케이션에 대해서도 안정적으로 작동함을 확인하였다.

모델 검증 방식의 또 다른 장점은 오탐 및 미탐을 고려하지 않아도 된다는 것이다. 실행경로 탐색을 이용하여 추출한 API 호출 시퀀스는 도달가능성 분석의 정확도에 따라 프로그램 실행 시 실제로 일어나지 않는 경로가 포함될 가능성이 높으며 이 경로가 제약사항 위반으로 판정되면 오탐을 유발할 수 있다. 반면, 모델 검증을 이용한 방법에서는 CBMC가 할당문과 조건문을 모두 고려하여 도달 조건을 생성하기 때문에, 애플리케이션의 행위를 그대로 반영하고 있어 실행경로 탐색 방법과 달리 오탐이 발생하지 않는다.

제안된 두 방식은 모두 탐색 경로의 길이를 제한하고 있기 때문에 미탐의 여지가 있으며 운영체제의 스케쥴링에 따른 TASK 수행순서를 고려하지 않았기 때문에 전역 제약사항 검증 시에는 오탐 및 미탐 발생 가능성이 있다. 이 문제를 해결하기 위하여 운영체제 모델을 함께 고려하여 소프트웨어의 제약사항을 검증하는 연구를 진행하고 있다.

References

[1] OSEK/VDX [Internet], http://www.osek-vdx.org/.

[2] D. Kim and Y. Choi, “Light-weight api-call safety checking for automotive control software using constraint patterns,” in *6th International Conference on IT Convergence and Security*, ICITCS’16, pp.314-318, 2016

[3] E. Clarke and D. Kroening, “A tool for checking ANSI-C programs,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp.168-176. Spinger, 2004.

[4] T. Byun and Y. Choi, “Automated system-level safety testing using constraint patterns for automotive operating systems,” in *Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC ’15*, pp.1815-1822, 2015.

[5] Jean-Luc Bechennec, et al., “Trampoline an OpenSource Implementation of the OSEK/VDX RTOS Specification,” *IEEE Conference on Emerging Technologies and Factory Automation,* 2006.

[6] Y. Choi, “Constraint specification and test generation for osek/vdx-based operating systems,” in *Proceedings of the 11th International Conference on Software Engineering and Formal Methods*, pp.305-319, Sept., 2013.

[7] M. Park, D. Kim, and Y. Choi, “CodeAnt : Code Slicing Tool for Effective Software Verification,” *KIPS Transactions on Software and Data Engineering*, Vol.1, No.1, pp.1-8, 2012.

[8] K. L. McMillan, “Symbolic Model Checking,” Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[9] E. M. Clarke, O. Grumberg, and D. Peled, “Model checking,” MIT press, 1999.

[10] R. E. Strom and S. Yemini, “Typestate: A programming language concept for enhancing software reliability,” *IEEE Trans. Software Eng.*, Vol.12, No.1, pp.157-171, 1986.

[11] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, “Effective typestate verification in the presence of aliasing,” *ACM Trans. Softw. Eng. Methodol.*, Vol.17, No.2, pp.9:1-9:34, May, 2008.

[12] J. Field, D. Goyal, G. Ramalingam, and E. Yahav, “Typestate verification: Abstraction techniques and complexity results,” *Sci. Comput. Program.*, Vol.58, No.1-2, pp.57-82, 2005.

[13] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvery, B. Ondrusek, S. K. Rajamani, and A. Ustuner, “Thorough Static Analysis of Device Drivers,” *ACM SIGOPS/EuroSys European Conference on Computer Systems*, Vol.40, Issue 4, pp.73-85, 2006.

[14] T. Ball, E. Bounimova, R. Kumar, and V. Levin, “SLAM2: Static driver verification with under 4% false alarms,” *Conference on Formal Methods in Computer-Aided Design*, pp.35-42, 2006.

김 동 우

e-mail : kdw9242@gmail.com

2015년 경북대학교 컴퓨터학부(학사)

현 재 경북대학교 컴퓨터학부 석사과정

관심분야 : 소프트웨어 안전성 분석, 정적 분석, 모델 검증

최 윤 자

e-mail : yuchoi76@knu.ac.kr

2003년 미국 미네소타대학 전산과(박사)

2003년~2006년 독일 프라운호퍼연구소 연구원

현 재 경북대학교 컴퓨터학부 부교수

관심분야 : 소프트웨어 안전성 분석, 정형검증, 모델기반 개발방법론