# SCADE를 이용한 리눅스 디바이스 드라이버 개발

송관호[0] 심재환 안영정 최진영
고려대학교 컴퓨터학과
{ghsong[0], jhsim, yjahn, choi}@formal.korea.ac.kr

# Linux device driver development by using SCADE

Gwan-Ho Song[0] Jae-Hwan Sim Young-Jung Ahn Jin-Young Choi
Dept. of Computer Science and Engineering, Korea University

요 약

본 논문에서는 정형기법을 이용한 리눅스 디바이스 드라이버 개발에 대한 내용을 다룬다. Device Driver 는 Reactive system에 속하는 대표적인 경우이다. 또한 Reactive system을 효과적으로 명세하고 검증하기 위한 정형기법 도구로 SCADE가 있다. 본 논문에서는 SCADE를 이용하여 실제 Linux device driver중 하나를 설정하여 이를 그 툴로 명세하고 검증한 후 구현한 후 발생한 여러 문제점을 통하여 실제 정형기법이 Linux device driver에 개발에 사용될 수 있는지를 논의한다.

## 1. 서 론

1996년도에 일어난 Ariane5 로켓 발사 실패, 미국 Virtual Case File (VCF) 프로젝트 실패 등과 같은 사건의 원인은 소프트웨어 오류이다 [1, 2].

특히 우주왕복선이나 의료장비 시스템의 소프트웨어 오류는 엄청난 인명피해를 초래할 수 있다. 더욱이 소프트웨어로 인한 프로젝트 실패는 천문학적인 예산 손실을 가져올 수 있다.

이러한 문제를 해결하기 위해 많은 방법론이 제시되고 있다. 본 논문은 그 중 하나인 정형기법을 통한 문제 해결을 제시하고 있다. 정형기법은 만들고자 하는 소프트웨어를 설계 단계에서부터 정형기법 언어로 명세하고 명세된 설계를 검증하는 방법을 연구하는 학문이다.

정형기법의 대표적인 언어들로는 Z, PVS, SMV 등이 있다 [3]. 이 언어들은 설계를 명세하고 검증하는 것에 중점을 두고 있다.

위와는 달리 실행가능한 명세란 개념이 있다. 이 개념은 명세 자체를 수행가능하게 하여 작성자가 그 동작을 확인함으로써, 그 명세의 정확성에 대한 확신을 가지는 것을 도울 수 있게 한 방법론이다 [4]. 또한 명세로부터 직접 구현되어질 시스템 상에서 실행 가능한 코드를 얻어내려는 연구가 활발히 진행되고 있다. 그 연구의 결과로 만들어진 언어 및 도구들로는 StateMate, SCADE 등이 있다.

본 논문에서는 Linux device driver를 설계함에 있어서 정형기법이 적용 가능한가를 논의하고, 만약 가능하다면 그로인한 장점과 단점은 무엇인지를 논의하고자 한다.

이를 위해서 정형기법 툴로는 SCADE를 이용하고 설계할 대상으로는 usb mouse device driver를 선택하였다.

논문의 진행순서는 우선 2장에서 관련연구를 논의하도록 한다. 그 후 3장에서 실험한 내용을 설명하고 결과를 분석한다. 결과분석에 따른 문제점은 4장에서 논의된다.

## 2. 관련연구

### 2.1 SCADE

SCADE는 동기화 언어인 Lustre와 Esterel을 기본으로 하는 통합 개발 환경이다[5,6]. Lustre는 data flow 시스템을 명세 하는데 최적화된 언어이고, Esterel은 control flow 시스템을 명세하는데 최적화된 언어이다. SCADE의 특징은 이 두 언어를 이용함으로써 여러 다양한 reactive 시스템을 설계, 모의 실험, 정형 검증을 할 수 있다는 점이다. 더욱이 SCADE를 통해 생성되는 소스코드는(C 혹은 Ada) 정형검증을 거친 모델로부터 직접 생성되는 것이기 때문에 안전하다 할 수 있다.

### 2.2 Linux Device Driver

운영체제인 리눅스는 C 언어로 구성된 monolithic 커널이다. 하지만 완전한 monolithic 커널은 아니다. 그 예외가 바로 device driver이다. device driver는 그 개발 주기가 빠르다. 또한 device driver의 수는 너무나 많기 때문에 리눅스 커널 안에 device driver를 직접 넣는 것은 많은 문제점을 야기한다. 그러한 문제를 풀기 위해서 리눅스는 device driver를 loaderable module로 구성하였다[7].

device driver는 빠른 개발 주기를 가지고 있다. 또한 잘못된 driver 코드는 OS를 정지 시킬 수 있다. 즉 디바이스 드라이버의 개발엔 안전에 대한 점검이 필수요소이다. 하지만 사람이 코딩을 하는 이상 device driver는 항상 오류로부터 자유로울 수 없다. 더욱이 잘못된 device driver 설계는 운영체제를 더욱 큰 혼란에 빠뜨리게 할 수 있다.

Device driver라는 프로그램은 그 driver가 맡는 하드웨어의 끊임없는 입출력을 driver가 잘 처리해서 운영체제에게 그 입출력을 넘기는 흐름을 가지고 있다. 이는 device driver가 data flow적인 흐름을 가진다고 할 수 있다[8].

device driver 개발에 정형기법이 이용되면 개발되는

driver는 안전하게 설계되고 검증될 수 있다. 하지만 이는 쉽지가 않다. Device driver의 코드의 대부분은 시스템 콜이 차지하며 또한 device driver의 실행과정은 일반적인 운영체제의 프로세스와 상당한 차이가 있다[7]. 일반적인 프로세스는 운영체제가 알아서 등록을 하지만 device driver는 driver 스스로가 등록을 해야 한다.

Device driver의 reactive한 특성은 정형기법 툴로 안전하고, 효율적으로 설계될 수 있다. 본 논문은 device driver의 reactive한 특성을 잘 명세할 수 있는 정형기법 도구인 SCADE를 사용하여 정형적으로 안전한 device driver를 개발하는 방법에 대해 논의한다.

## 3. 연구 내용

본 장의 목적은 SCADE를 사용해서 Linux device driver를 설계하고 검증한 후 SCADE를 통해서 생성된 안전한 코드를 가지고 직접 리눅스 드라이버를 구현하는 과정을 다룬다.

### 3.1 실험 시스템 및 구현 대상

본 논문에서 사용된 Linux 시스템은 -표1- 과 같다.

| | |
|---|---|
| Linux kernel 버전 | 2.6.19 |
| GCC 버전 | 4.1.1 |
| SCADE 툴 버전 | 5.0 |
| 컴퓨터 사양 | P4-2.6Ghz, 1GB ram, 120GB hdd |

- 표 1 -

본 논문에서 중점적으로 다룰 device driver는 usb mouse device driver이다. 이 device driver의 위치는 / 커널 소스/driver/usb/input/usbmouse.c 있다.

### 3.2 usbmouse.c 소스코드의 분석

대략적인 usbmouse.c의 함수들에 대한 설명을 - 표 2- 에 실었다. 각각의 함수에 대한 리턴값과 매개변수는 생략했다.

| 함수 | 설명 |
|---|---|
| usb_mouse_irq() | **device driver의 핵심 코드가 들어있는 부분** |
| usb_mouse_open() | module에 관련된 시스템 콜 함수 |
| usb_mouse_close() | module에 관련된 시스템 콜 함수 |
| usb_mouse_probe() | module dependency를 해결하기 위한 시스템 콜들 |
| usb_mouse_disconnect() | module에 관련된 시스템 콜 함수 |
| usb_mouse_init(void) | module 등록에 필요한 시스템 콜들 |
| usb_mouse_exit(void) | module 삭제에 필요한 시스템 콜들 |
| module_init(usb_mouse_init) | module 등록을 위한 표준 인터페이스 |
| module_exit(usb_mouse_exit) | module 삭제를 위한 표준 인터페이스 |

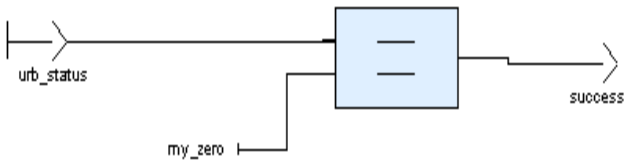
-표 2-

- 표 2- 에서 밑줄 친 usb_mouse_irq()를 제외한 함수는 단순히 운영체제 혹은 하위 모듈과의 통신을 위한 순차적인 시스템 콜들의 집합체이다. 따라서 이러한 부분은 SCADE로 설계하는 것 자체가 의미가 없다.

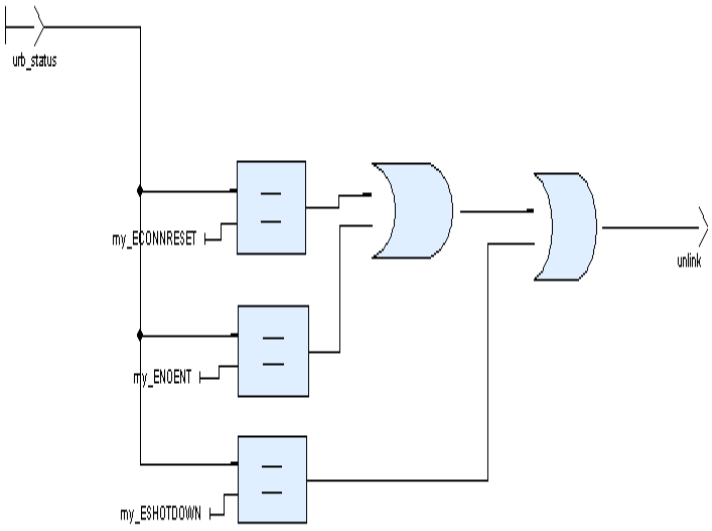
모든 usb 이벤트는 usb_core module에 의해 감지가 된다. usb_core는 만약 이 이벤트가 usb mouse와 관련된 이벤트라면 usbmouse.ko 모듈에게 전달하고 usbmouse.ko 모듈은 이벤트를 usb_core로부터 받아서 이를 처리하고 다시 usb_core로 제어권을 넘긴다. 이 때 이벤트를 처리하는 함수가 바로 usb_mouse_irq()이기 때문에 본 논문에서는 이 함수를 SCADE로 설계한다.

usb_core로부터 넘어온 이벤트 정보는 urb 구조체에 저장되며 usb_mouse_irq() 함수는 urb->status 정보를 판단하여 이벤트 정보를 usb_core로 넘긴다.

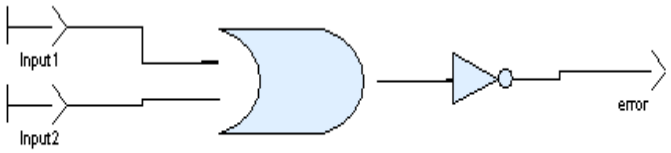
이러한 과정을 SCADE를 통해 설계를 하면 다음 그림과 같다.




-그림 1 success_node-

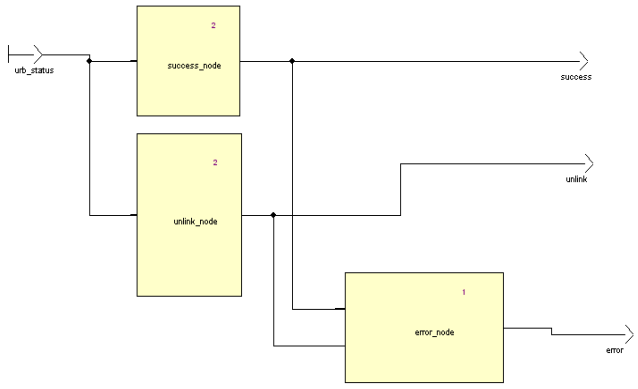



-그림 2 unlink_node-




-그림 3 error_node-

-그림 1,2,3-은 -그림 4-의 위에서 아래 순서대로의 사각형에 대응되는 node 들이다.

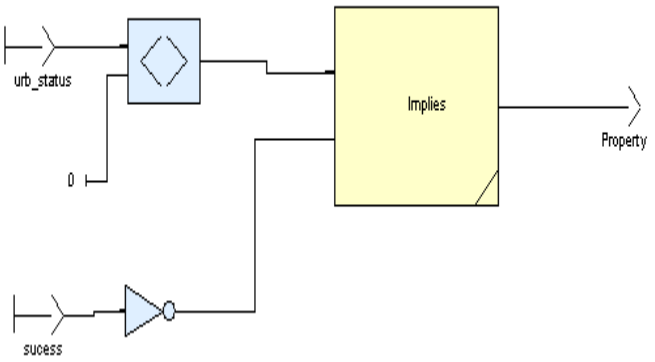



-그림 4 완성된 node-

### 3.3 설계된 node의 검증

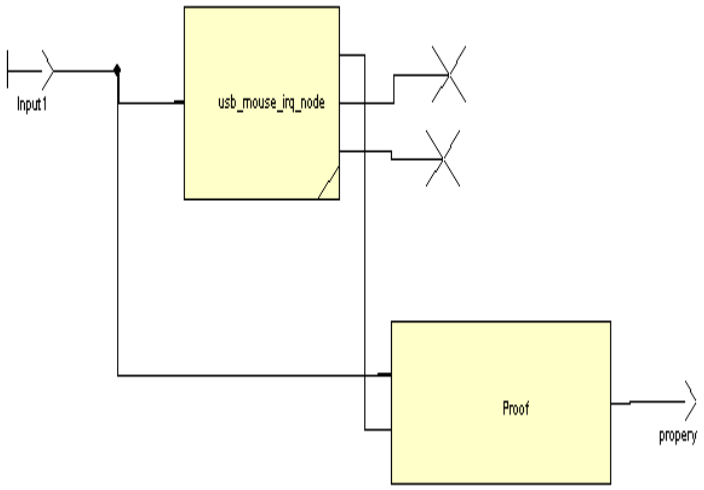
urb->status 값은 0일시에 정상적인 mouse event를 처리한다. 만약 urb->status 의 값이 -104, -2, -108 의 값이면 device driver의 연결을 끊게 된다. 그 이외의 값이 오면 단순한 error로 인식하고 해당 이벤트를 무시하게 된다. urb->status 의 값이 0이면 -그림4- 의 맨 위의 success 출력값은 1이며 다른 값들은 0이다. urb->status 의 값이 -104, -2, -108 이면 -그림4- 의 중간의 unlink 는 1이고 나머지는 0이다. 그 이외의 값이면 -그림4-의 맨 아래의 error는 1이고 나머지는 0이다.

여기에서 알 수 있는 검증 속성은 다음과 같이 정할 수 있다. '만약 mouse device driver가 error 혹은 unlink 이벤트를 받으면 그 driver는 정상적인 이벤트 등록을 하지 말아야 한다.' 즉 unlink가 1이거나 error가 1이라면 success 는 항상 0이여야 한다. 이를 SCADE의 검증 도구인 design verifier를 이용하여 명세하면 그림 5와 같다.




-그림 5 정형 명세-

-그림4- 와 -그림5-를 검증하는 과정은 -그림 6-에 나타내었다.




-그림 6 정형 검증-

### 3.4 구현

지금까지 usbmouse.c 를 SCADE로 설계하고 검증하는 작업을 하였다. 이제 남은 것은 검증이 끝난 설계로부터 안전한 코드를 얻고 이를 실제 Linux 드라이버 코드로 구현하는 것이다.

SCADE에 의해 생성된 코드는 일단 usb_mouse_irq_node.h 로 합쳤고 이를 usbmouse.c 안에 다 include 시켰다. 그 후 얻어진 코드와 usbmouse.c의 코드를 접목시키는 방법을 아래 -표3-에 나타내었다.

```
static void usb_mouse_irq(struct urb *urb)
{
        struct usb_mouse *mouse = urb->context;
        signed char *data = mouse->data;
        struct input_dev *dev = mouse->dev;
        int status;

1)      _C_usb_mouse_irq_node imsi;
2)      usb_mouse_irq_node_init(&imsi);

3)      imsi._IO_urb_status=urb->status;
4)      usb_mouse_irq_node(&imsi);

...
```

-표 3 구현-

1) 은 SCADE에 의해 구현된 자료구조를 선언 하는 과정이다. 2)는 선언된 자료를 초기화 하는 과정이다. 3)은 urb_status의 값을 자료구조에 넣는 과정이다. 4)의 과정은 입력된 자료를 바탕으로 이벤트를 실제로 처리하는 함수이다. 이 함수가 원래 usb_mouse_irq()가 하는 일을 대신한 함수이다. 즉 4)번의 함수는 SCADE를 통해 구현된 안전한 소스코드이다.

위와같이 usbmouse.c 를 수정한 후에 커널 모듈을 컴파일하면 usbmouse.ko라는 device driver 모듈을 얻을 수 있다.

### 3.5 비교평가

원래 usbmouse.ko 와 실험한 usbmouse.ko 의 크기를 -표 4- 에 나타내었다.

| | .ko 모듈 크기 | load시 크기 |
|---|---|---|
| 원본 usbmouse.c | 124228bytes | 6272bytes |
| 변경된 usbmouse.c | 127635bytes | 6912bytes |

-표 4 크기비교-

원본과 변경본의 용량차이는 약 3% 정도이다. 원본 usbmouse.c 는 비록 효율적이기는 하나 사람이 직접 작성한 코드이며 그로 인하여 항상 버그로부터 자유로울 수 없으며, 만들어진 코드 자체가 안전한 것인가는 전혀 알 수 없다. 반면에 본 논문에서 작성한 usbmouse.c 코드는 비록 원본보단 용량이 3% 크지만 사람으로부터 유발되는 버그를 확실히 피할 수 있다. 더욱이 검증된 모델로부터 나온 검증된 코드이기 때문에 코드의 행위자체를 추적할 수 있다. 따라서 코드 자체가 안전하다고 할 수 있다. 또한 해당 소스의 유지 보수도 수작업보다 훨씬 더 직관적인 GUI로 할 수 있기 때문에 쉽고 정확하고 빨리 device driver를 개발할 수 있다.

## 4. 결론

본 논문에서 정형기법이 Linux device driver 개발에 사용될 수 있는지 살펴보았고 어떠한 툴이 device driver적합한지를 알아보았다.

device driver는 reactive system에 속한다고 볼 수 있다[8]. SCADE는 Reactive system을 정형기법적으로 설계, 검증, 구현에 최적화된 도구이기 때문에 본 논문에서는 SCADE를 이용하여 device driver를 설계하였다.

Device driver를 설계하면서 나타났던 문제점 중 가장 큰 문제는 바로 '많은 수의 운영체제 시스템 콜을 어떻게 처리할 것인가?' 이다. Device driver는 일반 프로그램과는 다른 흐름을 가지고 있다. 모듈 실행의 큰 틀은 등록과 삭제이다. 실제 module memory에 등록하거나 삭제하는 함수들은 단순히 운영체제 시스템 콜 집합에 불과하기 때문에 본 논문에서는 그런 등록 혹은 삭제 함수들은 구현하지 않았다.

Device driver는 이벤트의 전송과 처리 그리고 재전송이라는 큰 틀을 가지고 있기 때문에 이벤트를 처리하는 함수를 찾아야 한다. 함수를 찾는 과정은 일일이 device driver 소스코드를 해석하는 과정을 통해서 찾아낼 수 있었다.

그 후 찾은 함수를 SCADE를 이용하여 설계하고, 검증한 후 자동 생성된 안전한 소스코드를 가지고 device driver를 구현해 보았다.

구현하는 과정에서 운영체제 시스템 콜을 사용하는 부분들은 어쩔 수 없이 일일이 찾아서 세심하게 생성된 소스코드에 반영하였다. 이 과정에서 알 수 있는 사실은 시스템 콜과 생성된 코드의 인터페이싱이 큰 이슈가 된다는 것이다. 더욱이 이 인터페이싱은 사람이 해줘야 하기 때문에 이로 인한 오류의 가능성도 무시할 수 없다. 즉 정형기법 툴을 이용해서 Linux device driver를 개발할 때 가장 문제가 되는 점은 바로 시스템 콜과 생성된 코드와의 인터페이싱 문제이다.

하지만 만약 이 인터페이싱 문제만 해결된다면 정형기법을 이용한 Linux device driver 개발은 -표4-에서 보는 바와 같이 약간의 용량을 지불하고 대신 더 쉽고 빠른 그리고 안전한 device driver를 개발할 수 있는 이점을 얻을 수 있다.

본 논문은 정형기법을 이용한 Linux device driver 개발은 효율적이며 안전하다는 것을 보였다. 하지만 인터페이싱 문제만큼은 정형기법자체만으론 해결할 수 없다는 문제점도 발견하였다. 이를 위한 해결 방법중 하나는 device driver를 구조화 시킬 때 시스템 콜부분과 실제 이벤트 처리를 담당하는 부분으로 나눔으로써 가능한 한 인터페이싱을 단순하게 하는 것이다.

인터페이싱의 문제만 보완이 된다면 정형기법은 Linux device driver를 개발하는데 커다란 도움을 줄 있다.

## 5. 참고문헌

[1] Peter B. Ladkin, "The Ariane 5 Accident: A Programming Problem?", http://www.rvs.uni-bielefeld.de/publications/Reports/ariane.html#References

[2] Harry Goldstein, "Who Killed the Virtual Case

File?", IEEE Spectrum September 2005.
[3] 조지호, “보안운영체제의 검증을 위한 정형기법 및 도구에 관한 연구”, 2003년 한국정보과학회 가을 학술발표논문집 Vol. 30. No. 2.
[4] 조승모, 차성덕, "LUSTRE를 이용한 SCR 명세의 구현," 정보과학회논문지(B) 제 26 권 제 2 호(1999.2).
[5] N. Halbwachs, P. Caspi, P.Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre," Proceedings of the IEEE, 79(9):1305-1320, September 1991.
[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: a declarative language for programming synchronous systems," In 14th ACM Symposium on Principles of Programming Languages, POPL'87, Munchen, January 1987.
[7] Jonathan Corbet, "LINUX DEVICE DRIVERS", O'REILLY, 3rd Edition.
[8] Gerard Berry, "The Esterel v5 Language Primer Version v5_91", July 27, 2000