Journal of Korea Multimedia Society Vol. 8, No. 5, May 2005(pp. 721-728)

임베디드 가상 기계를 위한 실행파일포맷

정한종†, 오세만††

요 약

가상 기계란 하드웨어로 이루어진 물리적 시스템과는 달리 소프트웨어로 제작되어 논리적인 시스템 구성을 갖는 개념적인 컴퓨터이다. 임베디드 시스템을 위한 가상 기계 기술은 모바일 디바이스나 디지털 TV 등의 다운로드 솔루션에 꼭 필요한 소프트웨어 기술이다. 현재 EVM(Embedded Virtual Machine)이라 명명된 임베디드 시스템을 위한 가상 기계에 대한 연구가 진행 중이다. 이러한 연구의 일환으로 본 논문에서는 임베디드 시스템을 위한 파일 포맷인 EFF(Executable File Format)로 정의한다. 또한 기존에 널리 사용되고 있는 클래스 파일을 EFF에 매핑 시킴으로써 EFF의 완전성을 구조적으로 증명한다.

An Executable File Format for Embedded Virtual Machine

Hangjong Cheong†, Seman Oh††

ABSTRACT

A virtual machine is a conceptual computer with a logical system configuration, made of software unlike physical systems made of hardware. Virtual machine technology for embedded systems is a requisite software technology for download solutions such as mobile devices, digital TVs, etc. At present, a research of virtual machines for embedded systems named EVM(Embedded Virtual Machine) is in progress. As a part of the research, we define the EFF(Executable File Format) as a file format for embedded systems. We also prove completeness of EFF by structurally mapping class file widely used to EFF.

Key words: EVM(Embedded Virtual Machine), EFF(Executable File Format), SIL(Standard Intermediate Language), Embedded Systems(임베디드시스템), Metadata(메타데이터), Completeness(완전성)

1. 서 론

가상 기계란 하드웨어로 이루어진 물리적 시스템과는 달리 소프트웨어로 제작되어 논리적인 시스템 구성을 갖는 개념적인 컴퓨터이다. 가상 기계 기술을 이용하면 응용 프로그램 실행 환경인 프로세서나 운영체제가 바뀌더라도 응용 프로그램을 수정하지 않고 사용할 수 있다. 대표적인 가상 기계로는 클래스 파일을 입력으로 받아 실행시키는 JVM(Java Virtual Machine)이 있다. 우리 나라에는 모바일 장치에 탑재가 가능한 GVM(General Virtual Machine)이 있다. 각각의 가상 기계들은 입력으로 받는 실행파일포맷이 있다. 예로 JVM의 클래스 파일 포맷, .NET CLR의 CLR 파일 포맷, GVM의 SGS 파일 포맷, 그리고 EVM의 EVM 파일 포맷 등이 있다[11].

임베디드 시스템이란 전용 동작을 수행하거나 또는 특정 임베디드 소프트웨어 응용 프로그램과 함께 사용되도록 디자인된 특정 컴퓨터 시스템 또는 컴퓨팅 장치를 말한다. 임베디드 시스템은 ROM 기반 운

※ 교신저자(Corresponding Author) : 오세만, 주소 : 서울시 중구 필동 3가 26번지(100-715), 전화 : 02)2260-3342, FAX : 02)2265-8742, E-mail : smoh@dongguk.edu

접수일 : 2004년 10월 11일, 완료일 : 2005년 4월 6일

† 정회원, 동국대학교 컴퓨터공학과
(E-mail : prana@dongguk.edu)
†† 정회원, 한국정보처리학회 게임연구회 위원장

※ 본 연구는 한국과학재단 특정기초연구(R01-2002-000-00041-0) 지원으로 수행되었음.

영 체제나 디스크 기반 시스템에 사용할 수 있지만, 범용 컴퓨터 또는 장치를 상업적으로 대체하여 사용할 수 없다. 이 시스템은 냉장고, 세탁기, 핸드폰, PDA, 홈 관리 시스템, 디지털 TV 등 여러 분야에 응용할 수 있다.

최근에는 GVM, KVM 등 모바일 단말기를 위한 가상 기계들이 개발되면서 그 중요성이 더욱 부각되고 있다. 특히, 임베디드 시스템을 위한 가상 기계 기술은 모바일 장치와 디지털 TV 등에 탑재할 수 있는 탑재 기술과 다운로드 솔루션을 이용한 동적인 실행 기술이 요구된다. 또한 컨텐츠 개발을 쉽게 하기 위해서 다양한 언어를 지원하고 지원된 언어들 간에 통합이 요구된다[10].

현재 마이크로소프트사의 C# 언어와 썬 마이크로 시스템즈사의 JAVA 언어를 모두 수용할 수 있는 가상 기계에 대한 연구가 진행 중이다. 이러한 가상 기계 솔루션은 EVM(Embedded Virtual Machine)이라 명명 되었다. EVM은C#, JAVA 등 객체지향 언어뿐만 아니라 C언어 또는 Pascal과 같이 순차적인 언어들도 수용할 수 있다. 이들 언어로 작성된 프로그램들을 어셈블리 언어 형태인 SIL(Standard Intermediate Language)로 변환하여 EVM을 위한 파일 포맷의 형태인 *.evm을 생성한다. 생성된 파일은 임베디드 시스템에 탑재된 가상 기계에서 실행될 수 있도록 한다[12].

이와 같은 프로젝트의 일환으로 본 논문에서는 클래스 파일 포맷, CLR 파일 포맷 등 기존의 가상 기계를 위한 파일 포맷들의 분석을 기반으로 하여 임베디드 시스템을 위한 실행파일포맷인 EFF를 정의한다. 또한 제안한 EFF의 완전성을 증명하기 위하여, 기존에 널리 사용 되고 있는 가상 기계를 위한 실행파일 포맷인 JVM의 클래스 파일 포맷과 매핑하여 구조적으로 증명한다.

2. 관련 연구

2.1 클래스 파일 포맷

클래스 파일은 JAVA 소스 코드가 JAVA 컴파일러에 의해 생성된 파일로 확장자가 *. class이며 JAVA 가상 기계의 입력이 되어 실행된다. 클래스 파일은 8비트 단위의 스트림으로 이루어져 있으며, 16비트, 32비트, 64비트 크기를 가진 데이터들은 8비트 단위로 나누어져 높은 비트가 먼저 나오는 빅 엔디언(Big-Endian)의 순서로 저장된다. 표 1은 클래스파일의 구조를 C 언어의 구조체 형태로 나타낸 것이다.

magic은 JAVA 클래스 파일임을 나타내주는 식별자로 항상 0CAFEBABE에 값을 가진다. minor_version과 major_version은 클래스 파일의 버전을 명시해 준다. constant_pool_count는 상수 풀의 개수를 나타내며, 항상 0보다 큰 값을 갖는다. 왜냐하면 맨 첫 번째 상수 풀은 JAVA 가상 기계에서 내부적으로 사용되어지기 때문이다. constant_pool은 가변 크기를 갖는 상수 풀을 표현한 것으로 배열의 형태이며 클래스의 이름, 슈퍼클래스의 이름, 메소드의 이름, 그리고 필드의 이름 등에 대한 정보를 가지고 있다. access_flags는 현재 클래스 파일이 나타내는 클래스의 접근 권한을 나타낸다. this_class는 클래스 파일이 포함하고 있는 클래스의 정보를 표시하기 위해서 위에 상수 풀의 인덱스를 나타내며 CONSTANT_Class_info 자료형의 상수를 가리키고 있다. super_class는 이 클래스 파일의 슈퍼클래스 정보를 표시하며 상수 풀의 인덱스를 나타내며 CONSTANT_Class_info 자료형의 상수를 가리키고 있다. 만일 이 필드의 값이 0이라면 묵시적으로 이 클래스는 java.lang.Object이다. 즉 모든 클래스의 최상위의 슈퍼 클래스를 의미하거나 슈퍼 클래스가 존재하지 않는 인터페이스임을 나타낸다. interfaces_count, interfaces는 인터페이스 필드 내에 위치한 인터페이스 정

표 1. 클래스 파일의 형식

```
ClassFile {
    u4              magic;
    u2              minor_version;
    u2              major_version;
    u2              constant_pool_count;
    cp_info     constant_pool[constant_pool_count-1];
    u2              access_flags;
    u2              this_class;
    u2              super_class;
    u2              interfaces_count;
    u2              fields_count;
    field_info          fields[fields_count];
    u2              methods_count;
    method_info         methods[methods_count];
    u2                  attributes_count;
    attribute_info  attributes[attributes_count];
}
```

보 개수, 실제 인터페이스 정보를 가진 상수 풀의 인덱스를 가진 배열이 위치한다. fields_count, fields는 클래스의 필드의 개수, 실제 필드의 정보를 나타내며 상수 풀의 인덱스들이 들어 있다. methods_count, methods는 클래스의 메소드 개수, 메소드의 실제 정보를 나타내며 상수 풀의 인덱스들이 위치한다. attributes_count, attributes는 실제 바이트 코드들로 이루어진 가상 기계용 루틴이 담겨 있는 attributes의 개수와 그 내용으로 구성된다[9].

2.2 CLR 파일 포맷

.NET 실행 파일은 윈도우즈 PE(Portable Executable)와 COFF(Common Object File Format)의 기준을 토대로 확장한 형태로써, CLR(Common Language Runtime) 환경에서 동작하기 위한 메타데이터와 IL(Intermediate language)를 포함하고 있다. .NET 실행 파일은 기존 윈도우즈 환경의 실행 파일과 동일하게 확장자가 *.exe 또는 *.dll이며, 낮은 비트가 먼저 나오는 리틀 엔디언 (Little-Endian)의 순서로 저장된다. 그림 1은 CLR 파일 포맷을 크게 네 부분으로 나누어 표현한 형태이다.

PE Headers는 운영체제와 파일의 속성 정보 등을 가진다. CLR Headers는 CLR 파일이 .NET 실행 파일임을 나타내는 정보들을 갖는다. CLR Data는 프로그램이 어떻게 실행될지를 결정하는 정보들로서 메타데이터와 IL 코드로 나누어진다. 메타데이터는 모듈에서 참조 또는 선언된 아이템(클래스와 멤버)들의 체계적인 설명으로서 타입 정의, 버전 정보, 외부 어셈블리 참조, 그리고 다른 표준화된 정보로 이루어져 있다. 즉 아이템들의 속성과 그들의 관계들을

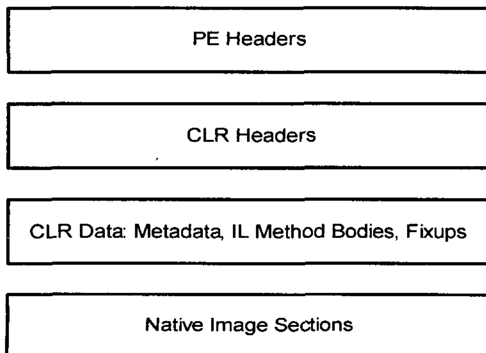

그림 1. CLR 파일 포맷의 구조

나타낸다. IL은 CLR 환경에서 실행되는 명령어들의 집합이다. Native Image Sections는 imports/exports가 포함된 데이터, 코드, 그리고 헤더들에 의해 묘사된 실질적인 데이터를 포함한다. 섹션 종류로는 data section, relocation section, unmanaged resource section, thread local storage data section 등이 있다[3].

2.3 CLR 파일 포맷과 클래스 파일 포맷 비교

CLR 파일 포맷과 클래스 파일 포맷은 가상 기계를 위한 실행파일포맷들이다. 그러나 많은 부분이 다르게 각각 구성되어 있다.

클래스 파일 포맷은 하나의 클래스 또는 인터페이스에 대해 파일로 구성되며, CLR 파일 포맷은 여러 개의 클래스들을 하나의 파일로 구성된다. 파일 내에 저장되는 비트의 순서도 서로 다른 방식을 취하고 있다. 클래스 파일 포맷의 경우에는 높은 비트가 먼저 나오는 big-endian으로 저장하며, CLR 파일 포맷의 경우에는 낮은 비트가 먼저 나오는 little-endian으로 저장한다.

파일 내에서 데이터를 참조하는 방식도 다르다. 이것은 각각의 메타데이터를 저장하는 형태가 다르기 때문이다. 클래스 파일 포맷은 상수 풀의 정보를 참조하기 위하여 상수 풀의 인덱스를 사용한다. CLR 파일 포맷은 전체적으로 파일 내에 정보를 참조하는 방식은 파일 오프셋과 크기를 이용하며, 메타데이터의 정보는 토큰타입과 인덱스가 결합된 형태인 메타데이터 인덱스를 사용한다.

링킹과 로딩 시 클래스 파일 포맷의 경우 한 개의 클래스로 구성되어 있기 때문에 관련된 모든 클래스 또는 인터페이스들을 모두 로드 해야 함으로 많은 시간이 소요된다. CLR 파일 포맷의 경우에는 메타데이터와 IL를 분리함으로써 메모리에 적재된 클래스들 중에 필요한 클래스만 로딩이 가능하다. 클래스 파일 포맷의 메타데이터 정보는 상수 풀에 저장되어 있으나 모든 정보를 얻기 위해서는 파일 전체를 검색해야만 얻을 수 있다. 이러한 단점을 보완하기 위해서 각 정보들 앞에 해당하는 비트의 길이를 표시하여 필요치 않은 부분에 정보들을 읽지 않고 다음 정보로 넘어 갈 수 있게 구성하고 있다. CLR 파일 포맷은 특정 부분에 모든 메타데이터 정보를 저장하고 있기 때문에 메타데이터 정보 추출이 쉽다. 표 2는 위에

표 2. CLR 파일과 클래스 파일 비교

	CLR 파일	클래스 파일
클래스	1개 또는 그 이상	1개
참조	오프셋, 크기, 메타데이터 인덱스	인덱스, 정보의 길이
비트 순서	little-endian	big-endian

내용을 간단하게 표로 작성한 것이다.

2.4 EVM (Embedded Virtual Machine)

EVM은 모바일 장치, 셋톱 박스, 디지털 TV 등에 탑재되어 동적 응용프로그램을 다운로드 하여 실행 할 수 있는 가상 기계 솔루션이다. 또한 컨텐츠 개발을 쉽게 하기 위해서 다양한 언어를 지원하고 언어들 간에 통합이 가능하다.

EVM은 크게 세 부분으로 나뉘어진다. 첫 번째 부분은 C# 또는 JAVA 등의 고급 프로그래밍 언어로 작성된 프로그램을 가상 기계를 위한 표준 중간 언어(SIL)로 번역하는 Translator 부분이다. 또한, EVM은C#, JAVA 등 객체지향 언어뿐만 아니라 C언어 또는 파스칼과 같이 순차적인 언어들도 수용할 수 있다. 두 번째 부분은 SIL코드들을 입력 받아 가상 기계에서 실행 가능 한 형태인 EVM 파일 포맷(EFF)으로 변환하는 EFF Generator 부분이다. 마지막으로 세 번째 부분은 실제 하드웨어에 탑재되어 EFF를 실행하는 가상 기계(EVM) 부분이다. EVM의 가상 기계 부분은 계층적인 구조로 설계되어 재목적 과정의 부담을 최소화 한다. 그림 2는 EVM의 시스템 구성도를 나타낸 것이다.

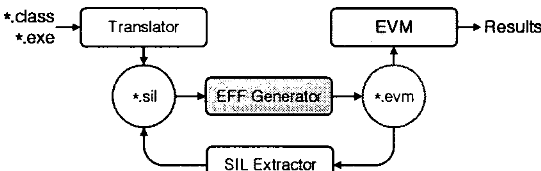

그림 2. EVM 시스템 구성도

3. EFF (Executable File Format)

3.1 설계 목표

EFF 설계 목표는 크게 4가지가 있다. 첫째, 순차적 언어인 C와 파스칼 등을 지원하며, 또한 C#과 JAVA 등 객체 지향 언어도 지원 한다. 동일한 파일 포맷과 중간 언어를 생성 함으로서 언어들간의 통합을 가능하게 한다. 둘째, 파일 포맷의 확장이 가능하고 융통성을 갖는 것이다. 차후에 추가적인 부분을 고려하여 설계한다. 셋째, 메타데이터와 중간 언어가 서로 독립적으로 구성하여 분석이 용이고 타입 체크가 편리한 구조로 설계한다. 넷째, 임베디드 시스템을 고려하고 있기 때문에 불필요한 정보를 제거하고 실행에 필요한 최소의 정보만 저장한다.

3.2 EFF의 구조

EFF는 여러 개의 클래스를 파일 안에 포함한다. EFF는 8비트들의 스트림으로 구성된다. 비트 스트림은 항상 리틀엔디언(Little Endian)으로 저장된다. 즉, 하위 바이트가 먼저 저장 되는 형태이다.

EFF의 구조를 표현하는 방법으로 EBNF의 변형된 형태를 사용하고 있다. 표현하고자하는 구조에 대해 시작과 끝을 표시하기 위해서 해당 구조의 이름을 '<>' 사이에 표시하여 시작부분을 나타내며, 끝부분은 이름 앞에 '/'를 표시하여 구조의 끝을 나타낸다. 구조 내에 U1_, U2_, U4_로 시작하는 엔트리들은 unsigned 1-byte, 2-byte, 4-byte 크기를 갖는 엔트리들을 표현하고 있으며, 구조 내의 '<>'로 표시되는 부분은 다시 확장된 구조를 갖는다. '{}'로 표현된 엔트리는 '{}'의 끝부분에 표시되는 엔트리의 수만큼 반복 된다. 또한 '[]'으로 표현된 엔트리는 존재할 수도 있고 존재 하지 않을 수도 있다는 의미이다. '()'으로 표현된 엔트리는 선택을 나타낸다. 표 3은 EFF의 전체 구조를 나타낸 것이다. 각각의 엔트리는 다음과 같다.

EFF을 식별하기 위한 magic엔트리는 항상 0×0E054DFF 값을 가진다. majorVersion과 minorVersion은 EFF의 주 버전과 부 버전을 나타내는 엔트리이다. module은 소스 파일의 이름을 나타내는 엔트리이며 MDTString index의 값을 가진다. language는 소스 코드의 언어를 나타내는 엔트리이며 MDTString index의 값을 가진다. 프로그램의 시작 메소드를 나타내는 entryPoint는 MDTMethod index의 값을 가진다. VMCodesCount는 코드 테이블의 총 개수를 나타내는 엔트리이다. <VMCodeInfo>는 실행 코드 테이블을 나타내는 엔트리이며 해당 메

표 3. EFF의 전체 구조

```
<EvmFile>
        U4_magic
        U2_majorVersion
        U2_minorVersion
        U2_module
        U2_language
        U2_entryPoint
        U2_VMCodesCount
        {<VMCodeInfo>}VMCodesCount
        {<MetadataInfo>}
</EvmFile>
```

소드 정보와 SIL명령어들이 저장된다. <VMCodeInfo>의 구조는 3.3절에서 설명한다. <MetadataInfo>은 메타데이터의 테이블 정보를 저장하는 엔트리이며 이 테이블의 개수와 종류는 태그의 존재에 따라 결정된다. <MetadataInfo>의 기본구조는 3.4절에서 설명한다.

3.3 VMCodeInfo

프로그램의 실행을 위한 명령어들을 저장하는 테이블이며 명령어들은 EVM을 위한 중간 코드인 SIL로 이루어져 있다. 표 4는 VMCodeInfo의 구조를 나타낸 것이다.

methodIndex는 명령어를 포함 메소드의 정보를 저장하며 MDTMethod index의 값을 갖는다. codeLength는 실제 코드가 저장된 바이트 스트림의 길이를 나타낸다. bytes는 codeLength만큼의 바이트들로 구성되며 이들은 SIL 명령어들과 해당하는 오퍼랜드들로 구성된다.

3.4 MetadataTable (MDT)

MDT는 파일의 메타데이터를 저장하는 부분이다. MDT의 각 테이블 들은 15개의 태그와 각 해당 엔트리들로 이루어져 있다. 테이블의 순서는 정해져 있지 않으며 테이블은 필요할 때 마다 삽입하면 된다. 다시 말하면, 해당 테이블의 정보가 없으면 파일 내에 테이블이 존재하지 않는다. 표 5은 메타데이터 테이블의 태그 종류와 번호를 나타낸 것이다

각 메타데이터 테이블들 간에 서로 참조를 한다. 이때 사용되는 것이 메타데이터 테이블 인덱스(MDT index)이다. MDT index는 해당 테이블의 태그와 레코드 번호로 이루어져 있어 MDT index만 보아도 어떤 테이블을 참조하고 있는지 알 수 있다. 예를 들면, MDT index의 값이 0×20000004이면, MDTString (0×2000)의 다섯 번째 레코드를 참조하는 것이다. 또한 참조하는 엔트리가 없으면 none(0×0000) 태그를 사용한다.

표 6은 MetadataInfo의 형태를 나타낸 것이다. tag는 해당 메타데이터의 태그(표 5참조)를 나타낸다. metaCount는 메타데이터 테이블의 총 레코드 수를 나타낸다. <MetaDataTable>는 메타데이터의 종류에 따라 각 테이블을 가지게 되며 각 테이블마다 크기와 엔트리의 수는 다르게 된다.

표 7은 MetaDataTable의 형태이다. []'로 표시되는 엔트리는 존재할 수도 있고 없을 수도 있다. 즉, one or zero이다. MDTRefClass는 참조하는 클래스들의 정보를 저장하는 테이블이다. 즉, API 클래스 정보들이 여기에 저장되게 된다. MDTDefClass는 프로그래머가 정의한 클래스들의 정보를 저장하는

표 4. VMCodeInfo 구조

```
<VMCodeInfo>
        U4_methodIndex
        U2_codeLength
        {U1_bytes}codeLength
</VMCodeInfo>
```

표 5. 메타데이터 테이블 태그

태그 이름	태그 번호	태그 이름	태그 번호
MDTRefClass	0×0100	MDTString	0×4000
MDTDefClass	0×0300	MDTUserString	0×4500
MDTNestedClass	0×0500	MDTInteger	0×5300
MDTInterface	0×0600	MDTFloat	0×5400
MDTRefMember	0×0700	MDTLong	0×5500
MDTField	0×1000	MDTDouble	0×5600
MDTMethod	0×2000	MDTException	0×e000
MDTDescriptor	0×3000		

표 6. MetadataInfo 구조

```
<MetadataInfo>
        U2_tag
        U2_metaCount
        {<MetaDataTable>}metaCount
</MetadataInof>
```

표 7. MetaDataTable 구조

```
<MetaDataTable>
    [<MDTRefClass>]
    [<MDTDefClass>]
    [<MDTNestedClass>]
    [<MDTInterface>]
    [<MDTRefMember>]
    [<MDTField>]
    [<MDTMethod>]
    [<MDTDescriptor>]
    [<MDTString>]
    [<MDTUserString>]
    [<MDTInteger>]
    [<MDTFloat>]
    [<MDTLong>]
    [<MDTDouble>]
    [<MDTException>]
</MetaDataTable>
```

테이블이다. MDTNestedClass는 중첩되는 외부와 내부 클래스의 정보를 저장하는 테이블이다. MDTInterface는 인터페이스 정보와 이를 상속하는 클래스 또는 인터페이스 정보를 저장한다. MDTRefMember는 참조하는 클래스들의 멤버들의 정보를 저장하는 테이블이다. MDTField는 클래스 내에 선언한 필드들의 정보를 저장하는 테이블이다. MDTMethod는 클래스 내에 정의한 메소드의 정보를 저장하는 테이블이다. MDTDescriptor는 필드의 타입과 메소드의 시그네춰 정보를 저장하는 테이블이다. MDTInteger, MDTFloat, MDTLong, 그리고 MDTDouble은 각 타입의 상수 값을 저장한다. MDTString은 시스템에서 사용하는 스트링을 저장하며 MDTUserString은 프로그래머가 정의한 스트링을 저장한다. MDTException는 예외 처리를 위한 정보를 저장한다. 각 테이블에 따라 엔트리들이 정의되어 있으며 표 8에서 MDTDefClass를 대표적으로 나타내고 있다. 표 8은 프로그래머가 정의한 클래스들의 정보를 저장하는 테이블이다.

accessFlag는 클래스의 접근 제한자를 나타내고,

표 8. MDTDefClass 구조

```
<MDTDefClass>
    U2_accessFlag
    U4_className
    U4_superClass
    U4_fieldIndex
    U4_methodIndex
</MDTDefClass>
```

className은 클래스의 이름을 나타내며 값은 MDTString 인덱스를 갖는다. superClass는 수퍼 클래스의 정보를 저장한다. 이것의 값은 MDTRefClass 인덱스 또는 MDTDefClass 인덱스를 갖는다. fieldIndex는 클래스가 포함하고 있는 필드들의 정보를 저장하며 값은 MDTField 인덱스를 갖는다. methodIndex는 클래스가 포함하고 있는 메소드들의 정보를 저장하며 값은 MDTMethod 인덱스를 갖는다. mdthodIndex와 fielndex가 1개 이상의 경우에는 다음 MDTDefClass 테이블의 인덱스 차로 알 수 있다.

4. EFF의 완전성

임베디드 시스템을 위한 실행파일포맷(EFF)을 설계 목표에 따라 정의하였으나 제안한 EFF의 완전성 증명이 요구된다. 따라서, EFF의 완전성을 증명하기 위하여 클래스 파일을 이용해서 구조적으로 증명한다. 클래스 파일 포맷의 구조는 썬사에서 제공하는 자료를 기준으로 한다. 매핑이 이루어지지 않는 부분은 직사각형으로 표현한다. 또한 매핑 관계에 대해 1:1, 1:n, n:1, 그리고 n:m을 선의 종류에 따라 표시하였다.

그림 3은 클래스 파일로부터 EFF로의 매핑을 나타낸 것이다. 클래스 파일은 1개의 클래스 단위로 구성되나 EFF는 1개 이상의 클래스를 포함한다. 이러한 이유 때문에 클래스 파일의 많은 부분들이 EFF의 MDT로 매핑 되고 있음을 보여주고 있다. 클래스 파일에서 EFF로의 매핑이 완전히 이루어지는 것을

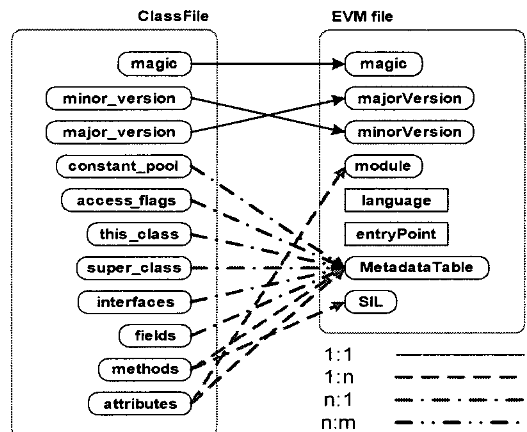

그림 3. 클래스 파일에서 EFF로의 매핑

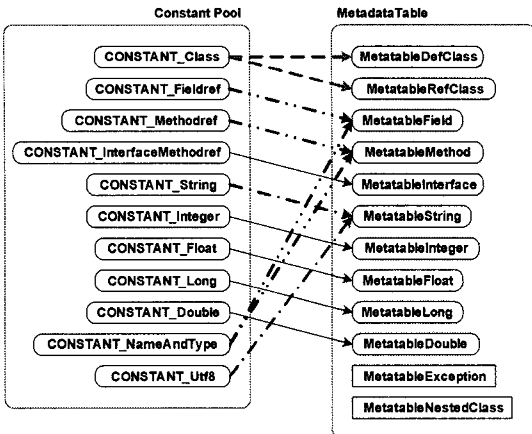

그림 4. 상수 풀에서 MDT로 매핑

그림 3을 통해서 알 수 있다.

그림 4는 클래스 파일의 상수 풀에서 EFF의 MDT로의 매핑을 나타낸 것이다. EFF의 MDT로 매핑이 되지 않는 부분은 클래스의 속성 부분에서 매핑이 이루어진다. 클래스의 상수풀이 EFF의 MDT로 완전하게 매핑 되는 것을 그림 4에서 볼 수 있다.

그림 5는 클래스 파일의 속성에서 EFF의 MDT로의 매핑을 나타낸 것이다. 클래스 파일의 속성에서 매핑이 이루어지지 않는 부분은 디버깅 정보, 불필요한 정보들, 그리고 상위 구조에서 매핑이 이루어진 부분들이다. EVM에서 필요치 않는 정보들을 제외하고 클래스 파일의 모든 부분들이 EFF에 매핑 되는 것을 알 수 있다.

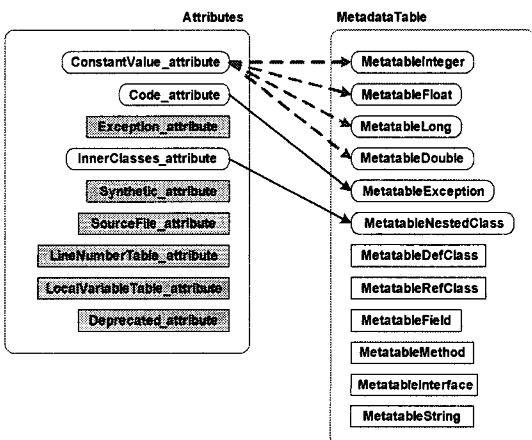

그림 5. 속성에서 MDT로 매핑

5. 결론 및 향후 연구

가상 기계 기술은 응용 프로그램의 이식성 확보라는 장점을 가진 기술이다. 특히, 임베디드 시스템을 위한 가상 기계 기술은 모바일 디바이스와 디지털 TV등에 탑재할 수 있는 핵심 기술로서 다운로드 솔루션에서는 꼭 필요한 소프트웨어 기술이다.

본 논문에서는 기존의 가상 기계를 위한 실행파일 포맷들의 분석을 기반으로 하여, 임베디드 시스템을 위한 실행파일포맷인 EFF를 설계 목표에 따라 정의하였다. EFF는 메타데이터와 중간 언어가 독립적으로 구성하여 파일 분석이 용이한 형태이다. 더불어 임베디드 시스템을 고려하고 있기 때문에 불필요한 정보를 제거하고 실행에 필요한 최소의 정보만을 저장함으로 파일의 크기를 줄일 수 있었다.

제안한 EFF가 임베디드 시스템을 위한 실행파일 포맷임을 증명하기 위하여, 가장 널리 알려진 가상 기계를 위한 실행 파일인 클래스 파일을 매핑 시킴으로써 EFF의 완전성을 구조적으로 증명하였다. 구조적으로 매핑이 이뤄졌으나 연구 과정에서 각각의 내부 데이터에 대한 매핑을 증명하였다. EFF는 가상 기계를 위한 실행파일포맷으로서 그 역할을 할 수 있다고 판단된다.

앞으로의 연구 과정에서 CLR 파일의 커버링 문제를 해결하기 위하여 EFF의 보완과 수정이 필요하겠고 EFF를 직접 실행할 수 있는 가상 기계 구현에 대한 지속적인 연구가 필요하겠다.

참 고 문 헌

[1] Edward G. Nilges, *Build Your Own .NET Language and Compiler,* Apress, Berkeley, 2004.

[2] David Stutz, *Shared Source CLI Essentials*, Sebastopol, OREILLY, 2003.

[3] James S. Miller, *The Common Language Infrastructure Annotated Standard*, Addison Wesley, Boston, 2003.

[4] Jeffrey Richter, *.NET FRAMEWORK PROGRAMMING*, Microsoft Press, Seoul, 2002.

[5] Joshua Engel, *Programming for the Java*

Virtual Machine, Addison-Wesley, Seoul, 2000.

[6] Kevin Burton, *.NET Common Language Runtime*, SAMS, Indiana, 2002.

[7] Serge Lidin, *.NET IL Assembler*, Microsoft Press, Canada, 2002.

[8] Thuan Thai, *.NET Framework Essentials 2nd Edition*, OREILLY, Sebastopol, 2002

[9] Tim Lindholm, *The Java Virtual Machine Specification*, SUN, Palo Alto, 1999.

[10] 남동근, 오세만, "가상 기계 코드의 커버링 문제," 한국정보과학회 가을 학술발표논문집(I), 제30권, 제2호, pp.247-249, 2003.

[11] 오세만, 컴파일러 입문 개정판, 정익사, 서울, 2004.

[12] 오세만 외 2인, 임베디드 시스템을 위한 가상 기계의 설계 및 구현, 한국과학재단 특정 기초 연구, 2차 중간보고서, 서울, 2004.

[13] 오세만 외 3인, JAVA 입문 개정판, 생능 출판사, 서울, 2002.

[14] 정한종, 윤성림, 오세만, "가상 기계를 위한 실행 파일포맷," 한국정보처리학회 추계학술발표논문집, 제 10권, 제2호, pp.647-650, 2003.

[15] 정한종, 오세만, "EVM 파일 포맷의 정의와 커버링 문제," 한국정보과학회 2004 봄 학술발표 논문집(B), 제 31권, 제 1호, pp.844-846, 2004.

정 한 종

2003년 상지대학교 전자계산공학과(학사)
2005년 동국대학교 컴퓨터공학과(석사)

관심분야 : 컴파일러, 프로그래밍 언어

오 세 만

1985년 3월~현재 동국대학교 컴퓨터공학과 교수
1993년 3월~1999년 2월 동국대학교 컴퓨터 공학과 대학원 학과장
2002년 11월~2003년 11월 한국정보과학회 프로그래밍 언어연구회 위원장
2003년 11월~현재 한국정보처리학회 게임연구회 위원장
관심분야 : 컴파일러, 프로그래밍 언어, 모바일