The Report Committee for William Daniel Leara
Certifies that this is the approved version of the following report:

Language Applications for UEFI BIOS

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____

Adnan Aziz

_____

Christine Julien

Language Applications for UEFI BIOS

by

William Daniel Leara, B.S.B.A.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Acknowledgements

I would like to thank my Supervisor, Dr. Adnan Aziz, for the support and guidance he provided me for this report, and most especially for the sense of engineering rigor his classes have inspired in my work.

I would also like to thank my Reader, Dr. Christine Julien for her input to my report and for the joy for teaching that she brings to her classes.

Thank you to all the other excellent faculty I had during my tenure in the Master's program: William Bard (thank you times three!), Dr. Vijay Garg, Dr. Joydeep Ghosh, and Dr. Sarfraz Khurshid.

Finally, I'd like to extend my appreciation to the managers from my workplace who have supported my participation in the Master's program: Dalvis Desselle, Tracy Harmer, Muhammed Jaber, Tony O'Connor, and Bill Weis.

Abstract

Language Applications for UEFI BIOS

William Daniel Leara, M.S.E.

The University of Texas at Austin, 2014

Supervisor: Adnan Aziz

The Unified Extensible Firmware Interface (UEFI) is the industry-standard Basic Input/Output System (BIOS) firmware specification used by modern desktop, portable, and server computers, and is increasingly being ported to today's new mobile form factors as well. UEFI is firmware responsible for bootstrapping the hardware, turning control over to an operating system loader, and then providing runtime services to the operating system.

ANTLR (ANother Tool for Language Recognition) is a lexer-parser generator for reading, processing, executing, and translating structured text and binary files. It supersedes older technologies such as `lex/yacc` or `flex/bison` and is widely used to build languages and programming tools. ANTLR accepts a provided grammar and generates a parser that can build and walk parse trees.

This report studies UEFI BIOS and compiler theory and demonstrates ways compiler theory can be leveraged to solve problems in the UEFI BIOS

domain. Specifically, this report uses ANTLR to implement two language applications aimed at furthering the development of UEFI BIOS implementations. They are:

1. A software complexity analysis application for UEFI created that leverages ANTLR's standard general-purpose C language grammar. The complexity analysis application uses general-purpose and domain-specific measures to give a complexity score to UEFI BIOS modules.
2. An ANTLR grammar created for the VFR domain-specific language, and a sample application which puts the grammar to use. VFR is a language describing visual elements on a display; the sample application creates an HTML preview of VFR code without requiring a developer to build and flash a BIOS image on a target machine to see its graphical layout.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 UEFI BIOS

Computers have traditionally required some kind of bootstrap firmware to initialize processor and chipset components, perform a power-on self-test (POST), hand off control to an operating system (OS) loader, and then act as an abstraction layer to hide hardware details from the OS. When the IBM PC was introduced in 1981, it termed its bootstrap firmware the ROM BIOS: Read Only Memory Basic Input/Output System.

The BIOS insulates the operating system and application software from the hardware by providing primitive I/O services and by programming the hardware's interrupt handling. By creating this abstraction between the hardware and software, the BIOS insures that computer software is compatible with future generations of computer hardware [1].

Today, there is an industry-accepted standard for BIOS firmware: Unified Extensible Firmware Interface (UEFI). Since 2004, the computer industry has made the transition from proprietary, monolithic BIOS firmware tight coupled with the Intel x86 architecture, to standards-based, modular BIOS firmware implementing UEFI. This report focuses on BIOS firmware that implements the UEFI interface.

1.2 Compiler Theory

In compiler theory, a grammar is a formal description of a programming language. In order to interpret programming language source code, two steps are needed: lexical analysis and syntax analysis. The process of grouping

characters into words (called tokens) is lexical analysis, and is performed by a program called a lexer. The lexer groups characters from a character stream into tokens. The process of grouping tokens into meaningful sentences of the programming language is called syntax analysis, or parsing. A grammar contains rules which define the tokens and sentences that make up the programming language.

Tools exist that accept a provided language grammar and automatically generate lexers and parsers that interpret input for that language. One such tool popular today is ANTLR, ANother Tool for Language Recognition. ANTLR parsers improve upon older tools like `lex/yacc` or `flex/bison` by using a new parsing technology called ALL(*). First introduced in ANTLR version 4, the ALL(*) parsing technology performs grammar analysis dynamically at runtime rather than statically, before the generated parser executes. Since ALL(*) parsers have access to actual input sentences, they can better figure out how to recognize token sequences by weaving through the entire grammar. Tools like `lex/yacc`, on the other hand, have to consider all possible (potentially infinitely long) input sentences [2]. ANTLR4 deemphasizes the need for code embedded in a grammar (called semantic predicates) and instead uses proven design patterns called Listeners and Visitors to traverse the token input.

A Language Application is defined as a program that uses a parser generator tool to auto-generate a lexer-parser, and then instantiates that lexer-parser to accomplish some useful task in a programming language. ANTLR's ALL(*) technology allows grammars to be encapsulated from application code, enabling reuse of the grammar in multiple language applications without recompiling the lexer and parser.

1.3 Objective

This report describes language applications based on grammars input to the ANTLR tool that generate parsers which parse UEFI BIOS source code. The intention is to create tools to improve the state of the art of UEFI BIOS development. After an introduction to the ROM BIOS and to compiler theory, this report demonstrates two different mini-projects:

1. A tool for UEFI module complexity based on a provided grammar
2. A grammar and tool for Visual Forms Representation (VFR)

Finally, a Conclusion summarizes the results. All the source code for the ANTLR grammars and Java applications produced for this report are found in the Appendices:

Appendix A: complexity application (Java)

Appendix B: UEFI image release frequency

Appendix C: UEFI image change frequency

Appendix D: ANTLR grammar for VFR

Appendix E: VFR application (Java)

Appendix F: HTML auto-generated by VFR application

Chapter 2: The ROM BIOS

The term BIOS was introduced into the nascent microcomputer industry by Gary Kildall, inventor of the popular operating system Control Program/Monitor (CP/M). In order for CP/M to run without modification on any of the Intel 8080-based computers of the late 1970s, Gary Kildall created an abstraction layer he called the BIOS. For independent computer vendors to ensure compatibility with CP/M, it was only necessary that they implement the specified BIOS routines in ROM.

IBM released its own Personal Computer (PC) in 1981 to great fanfare. Its launch was so successful that competitors were inspired to make compatible machines, or clones, of the IBM PC. This was possible since all the components of IBM's PC were available off the shelf from several vendors with one exception: the ROM BIOS. In 1982, however, Compaq Computer reverse-engineered the BIOS and began selling IBM PC-compatible machines. Soon, independent BIOS vendors such as Phoenix Technologies appeared offering for sale an IBM-compatible ROM BIOS to anyone desiring to sell a PC clone, and companies such as Dell, AST, and Northgate were founded. A de facto standard emerged: any machine with a BIOS that could boot Microsoft DOS and run popular IBM PC software such as Lotus 1-2-3 was considered "IBM PC Compatible", and an industry was born. This report refers to that original BIOS standard as Legacy BIOS.

2.1 Legacy BIOS

Booting a computer begins with its power supply. A computer's power supply provides the operating voltages necessary for system operation. When a

computer is turned on, a period of time is required for the power supply's output voltages to reach their proper operating levels. If the system were to begin operation before supply voltages had stabilized, erratic operation would result [3]. Therefore, the power supply asserts the processor's RESET# signal in order to hold the processor in reset thus preventing it from fetching code. Once the power supply output voltages stabilize, the power supply asserts the POWERGOOD signal, de-asserts the processor's RESET# signal, and the processor begins fetching code at an address preprogrammed by the chipset.

Intel chipsets program the processor to start fetching instructions at sixteen bytes below top of memory. The Intel 8088 processor used in the original IBM PC had a 20-bit address bus capable of addressing 1MB of memory. Therefore, the processor fetches the instruction at memory location `0xFFFF0`. The instruction found at this location is preprogrammed by the chipset components to be a jump to the BIOS, thus initiating the boot process. While many things about personal computers have changed over the decades, at least this one thing has remained constant: today's Dell, H-P, and Apple computers still start up like the 8088-based computers from 1981.

2.1.1 Power On Self-Test (POST)

There is no point in a user trusting his time and data to faulty hardware. Therefore, the first task of the BIOS is to perform a Power On Self-Test (POST) [4]. The POST tests write data to an I/O port or register and then read back data looking for an expected result. If there is a problem, the BIOS can report errors to the user using either beep codes or flashes from LEDs. The POST tests components in the following order [1]:

1. Microprocessor
2. CMOS RAM
3. ROM BIOS routines (checksum)
4. System chipset
5. Programmable timer counter #1
6. DMA controller and page registers
7. Base 64K of system RAM
8. Serial and parallel port peripherals
9. Programmable interrupt controller
10. Keyboard controller

2.1.2 Initialization

After the POST is completed, the BIOS begins initializing the computer's components. For example [5]:

- Perform any microcode updates appropriate to the microprocessor
- Setup the microprocessor's cache-as-RAM feature
- Program the microprocessor's Model Specific Registers (MSRs)
- Configure chipset registers
- Detect and configure memory, create the system memory map
- Populate BIOS data structures, such as the BIOS Data Area and BIOS Extended Data Area

These initialization steps are described in the following subsections.

2.1.2.1 Microcode Updates

In response to the Intel Pentium FDIV bug of 1994 [6], microprocessors began featuring a small, updatable firmware component called microcode. Its

intent is to make bugs field serviceable via a BIOS update, rather than recalling the processor hardware. A check is made during boot to detect if the BIOS contains a new version of the processor's microcode, and if so, an update is made.

2.1.2.2 Cache-as-RAM

When the computer starts up memory is not initialized, and therefore the processor is incapable of running code produced by a C compiler because there is no memory to support a stack—a requirement of C language programs. This forces programmers to rely only on processor registers for data storage, which is quite a limiting restriction when trying to boot strap the machine.

Intel introduced a feature in their recent models of microprocessors called cache-as-RAM, or No-Eviction Mode (NEM). This feature allows the programmer to utilize the processor's L2 cache as if it were RAM until main memory is initialized and ready to accept data. The BIOS turns the cache-as-RAM feature on to get the system up and running and off after memory initialization is complete.

2.1.2.3 Model Specific Registers (MSRs)

MSRs are configuration options for the microprocessor. Examples include features like overclocking support, the processor's margin for thermal events, and processor performance vs. energy savings trade-offs. Each model of microprocessor has different MSRs and each hardware vendor configures them for their own particular system design.

2.1.2.4 *Chipset Registers*

Modern chipsets have hundreds of configuration registers for things like power management, expansion bus configuration, wake-on-LAN support, embedded device configuration, and system manageability features. The BIOS must program all these registers appropriately for each system.

2.1.2.5 *Memory*

Taming the wide range of memory-related configuration options is a key role of the BIOS. Memory configuration includes parameters such as number of ranks, size, paging policy, and power management.

Once the memory is initialized and working, the BIOS passes to the OS a map of which memory address ranges are available and which are reserved. Memory ranges get reserved by Peripheral Component Interconnect (PCI) devices, Advanced Configuration and Power Interface (ACPI) run-time services, memory-mapped I/O regions, and BIOS run-time services, among others. Still other memory ranges might be detected during POST as defective—these get marked by the BIOS as unavailable for use by the OS.

2.1.2.6 *BIOS Data Areas*

There are several scratchpad memory address ranges reserved for the BIOS to store its run-time data. For example, the ranges `0x400-0x4FF` and `0x9FC00-0x9FFFF` define the BIOS Data Area and Extended BIOS Data Area, respectively. These areas contain BIOS status information such as whether or not a math coprocessor is installed, the current console video mode, and the number of floppy disk drives attached. After the BIOS populates these areas, application programs can make queries to test the current system configuration.

2.1.3 PC Interrupts

The original PC BIOS was tightly coupled with Intel's x86 architecture. Processors in the Intel x86 family are controlled largely through the use of interrupts, which can be generated by the processor, hardware or software [7]. There are a total of 256 possible interrupts, and each can be parameterized by placing function codes in the processor's general purpose registers before calling them. All BIOS services are invoked by interrupts. Interrupt processing is managed by a directory called the Interrupt Vector Table. Each interrupt is assigned to a particular location in the Interrupt Vector Table that contains the address of an Interrupt Service Routine associated with that interrupt. This design makes it possible for any program to request a BIOS service without knowing the specific memory location of the Interrupt Service Routine. It also allows the services to be moved around or expanded without affecting the programs that use the services [7]. During POST, the BIOS writes the Interrupt Vector Table to memory address 0x0 and initializes the addresses of all those interrupts whose Interrupt Service Routines are handled by the BIOS. Later, the OS will populate the Interrupt Vector Table with its own interrupts [8]. When an interrupt is triggered, the entry in the Interrupt Vector Table corresponding to the interrupt is called. Control of the computer is turned over to the Interrupt Service Routine at the location pointed to by the entry in the Interrupt Vector Table. Example of the beginning of a possible Interrupt Vector Table implementation:

Interrupt	Location in Interrupt Vector Table	Function	Type
0x0	0x0000	Divide by Zero	Processor
0x1	0x0004	Single Step	Processor
0x2	0x0008	Non-Maskable Interrupt	Processor
0x3	0x000C	Breakpoint	Processor
0x4	0x0010	Overflow	Processor
0x5	0x0014	Print Screen	Software
0x6	0x0018	Invalid Op Code	Processor
0x7	0x001C	Math coprocessor not present	Processor
0x8	0x0020	System Timer	Hardware
0x9	0x0024	Keyboard	Hardware
0xA	0x0028	IRQ2 cascade from 2nd PIC	Hardware
0xB	0x002C	Serial Port 2 (IRQ3)	Hardware
0xC	0x0030	Serial Port 1 (IRQ4)	Hardware
0xD	0x0034	Parallel Printer (LPT2)	Hardware
0xE	0x0038	Floppy Diskette (IRQ6)	Hardware
0xF	0x003C	Parallel Printer (LPT1)	Hardware
0x10	0x0040	Video Services	Software
…	…	…	…

Table 1: Example Interrupt Vector Table Implementation

2.1.4 Option ROMs

Next, the BIOS scans the computer buses for peripheral devices, and if present, loads option ROM routines supplied by the peripheral manufacturers. An option ROM is a set of peripheral-specific routines necessary for the hardware to function; they get loaded by the system BIOS from the peripheral add-in card's ROM into system RAM. New Interrupt Service Routines may be added, or the function of existing routines may be changed [4]. Thus, this system of option ROMs is a way to implement extensibility to the PC BIOS architecture.

2.1.5 Boot-Strap Loader

Finally, the BIOS runs its boot-strap loader routine to load an operating system from disk. Bootable disks (floppy disk or hard drive) contain a boot record in their first sector to which the BIOS passes control. The boot record code then loads the operating system.

2.1.6 Runtime

Even after the OS has loaded, the BIOS is still present, interacting with the OS and device drivers, handling power management tasks, and providing run-time services for the machine.

2.1.7 Industry Standards

Computers today must support various industry standards to ensure their ability to boot a variety of operating systems, work with various hardware technologies, and support manageability features required by IT administrators. The following subsections briefly highlight some of these industry standards supported by the BIOS firmware.

2.1.7.1 Hardware Devices

Personal Computers support a wide array of hardware technologies: Peripheral Component Interconnect (PCI), Universal Serial Bus (USB), Serial AT Attachment (SATA), DisplayPort (DP) and Small Computer System Interface (SCSI) devices to name a few. The BIOS must:

1. Support the bus protocols for each of these types of devices
2. Configure the registers in these controllers and their devices

2.1.7.2 ACPI

The Advanced Configuration and Power Interface (ACPI) was developed to establish industry-standard interfaces for enabling OS-directed configuration, power management, and thermal management of mobile, desktop, and server systems. Before ACPI, power, thermal, and system configuration were poorly specified and vendor-proprietary. The ACPI specification enables new power management technologies to evolve independently in operating systems and hardware while ensuring that they continue to work together.

When first published in 1996, ACPI evolved the then-existing collection of configuration and power management BIOS code: 1) Advanced Power Management (APM) programming interfaces; 2) Plug and Play (PnP) BIOS interfaces; and 3) Multiprocessor Specification (MPS) data structures into one comprehensive power management and configuration interface specification [9]. All modern BIOS implementations implement the ACPI specification.

2.1.7.3 System Management BIOS (SMBIOS)

There is a need to standardize the mechanisms whereby computer hardware vendors expose management information about their products [10]. This way, independent software vendors can write enterprise manageability products that work equally well on all hardware vendors' machines. SMBIOS is the specification that provides this cross-vendor standardization; it is the BIOS that provides the implementation. The SMBIOS specification is maintained by the Desktop Management Task Force (DMTF), a forum of corporations throughout the IT industry given the charge to maintain several different system management interface specifications.

2.1.7.4 DASH

Desktop and Mobile Architecture for System Hardware (DASH) is another interface specification maintained by the DMTF. DASH delineates a list of requirements for out-of-band and remote management of desktop and portable systems. The implementation details are left to the hardware vendor to implement in BIOS. DASH describes BIOS requirements for features such as software deployment, remote wake-up, and remote Keyboard-Video-Mouse (KVM) capability [11].

2.2 UEFI BIOS

The aforementioned de facto standard of the IBM PC Compatible BIOS served the industry well for roughly twenty years. However, with the introduction of new Intel 64-bit microprocessors, it became apparent that the 16-bit, real-mode, poorly standardized PC BIOS needed to be replaced. Intel pioneered a new standard, and in the early 2000s donated an open source implementation and associated software tools to an industry forum. This new BIOS standard is called the Unified Extensible Firmware Interface (UEFI) and is used by all IBM PC Compatible computers of recent vintage.[1] UEFI accomplishes all the same previously discussed tasks as Legacy BIOS: POST, memory, processor and chipset initialization, loading of option ROMs, industry standards implementation, and boot strapping an OS. However, it does so in a formally defined, vendor-agnostic, modular way.

[1] When Apple Inc. switched from Motorola to Intel microprocessors, they in effect became another "IBM PC Compatible" machine—today Apple Macintosh computers also run UEFI BIOS firmware.

2.2.1 Overview

UEFI describes an interface between the BIOS and OS. The interface is in the form of data tables that contain platform-related information, and boot and run-time service calls that are available to the OS loader and the OS. Together, these provide a standard environment for boot strapping an OS [12]. UEFI specifies a series of phases that take the computer from the reset vector to OS hand-off, and then provides run-time services support. This process is depicted graphically in the following figure [13]:

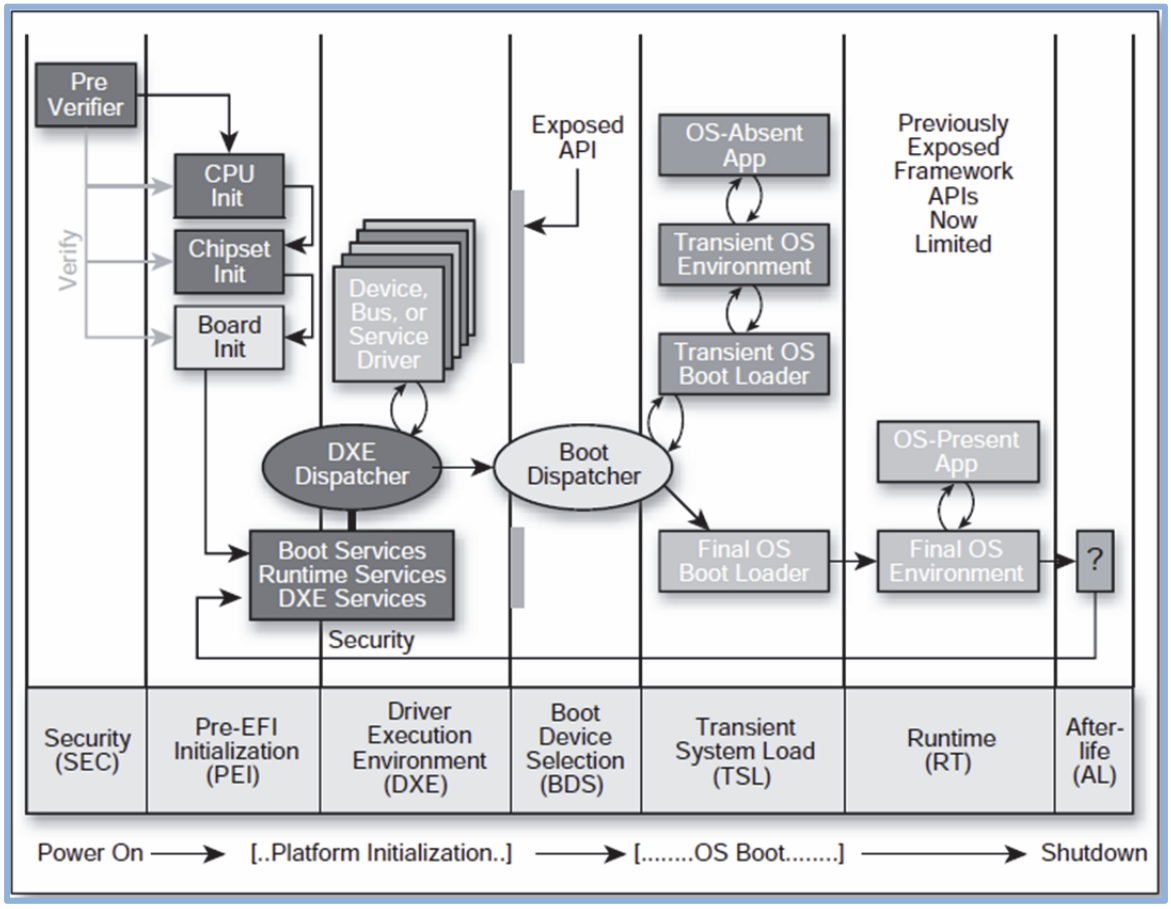

Figure 1: UEFI Boot Phases

The following subsections offer a brief summary of each phase in the UEFI boot process.

2.2.1.1 Security (SEC)

The SEC phase is named Security after its role in establishing a hardware root of trust for the system. Acknowledging that a BIOS rootkit is the ultimate rootkit—able to completely obscure itself from OS-level virus detection—SEC uses various technologies to ensure a trusted BIOS image.

Besides acting as the foundation for a hardware root of trust, SEC switches the processor to protected mode, executes any CPU microcode patch updates, and initializes the processor's No-Eviction Mode so that the L2 cache can be used as RAM [13].

2.2.1.2 Pre-EFI Initialization (PEI)

PEI performs two primary roles:

1. Determines the source of the system boot path, e.g., resume from Stand-By or cold start
2. Provides a minimum amount of permanent memory for the ensuing DXE phase

PEI's main goal is to initialize enough of the system to allow instantiation of the DXE phase [13]. Especially important is getting memory initialized. PEI takes the system from having to rely on using the L2 cache as RAM, and executing instructions in place (XIP [14]) from the ROM, to making main memory available and loading UEFI modules from ROM and into RAM. PEI also does early initialization of CPU-related functions, chipset devices, and sets up the System Management Bus (SMBus). SMBus is an I^2C derived bus used by the system for communicating power, thermal, and manageability messages.

2.2.1.3 *Driver Execution Environment (DXE)*

DXE is the workhorse of the booting process. The DXE phase discovers the resources described by the prior PEI phase, discovers any firmware volumes on the system, and then systematically goes about executing all UEFI modules (called DXE drivers during this phase) until all executable modules are exhausted from every one of the firmware volumes on the system.

2.2.1.4 *Boot Device Selection (BDS)*

BDS can be considered a subset and the very last part of the DXE phase. When the DXE dispatcher has searched all firmware volumes and dispatched all DXE drivers found, BDS takes over [13]. As its name implies, BDS selects a boot device and hands control over to an OS boot loader. The OS loader is a special type of UEFI application that is responsible for calling the `ExitBootServices()` function and starting the operating system. The `ExitBootServices()` function frees up system resources used by the BIOS and allows the OS to take over control of the system.

2.2.1.5 *Run-Time (RT)*

Although the BIOS' job in boot strapping the system has ended, the work of the BIOS is not finished. During RT the BIOS offers the OS and OS-level applications services for security, manageability, power management, and thermal control.

2.2.2 UEFI Driver Model

A key innovation of UEFI is its driver model. In Legacy BIOS, the components of pre-OS functionality were proprietary and compiled together into a monolithic whole, without industry-agreed upon interfaces. The UEFI Driver

Model is designed to support the execution of modular pieces of code, called drivers, that can leverage agreed upon interfaces to be moved from one UEFI implementation to another. These drivers may control hardware buses and devices on the platform, or they may provide some software-derived service. The UEFI Driver Model also contains information required by UEFI driver writers to design and implement combinations of bus drivers and device drivers that the computer needs to boot a UEFI-compliant OS [12]. The subsequent chapter in this report on Module Complexity analyzes UEFI drivers.

2.2.3 Architectural Protocols

UEFI abstracts the platform hardware from higher level boot and run-time services through the notion of Architectural Protocols (AP). The Architectural Protocols are like other services provided by UEFI components except that these protocols are consumed by the platform's core services. The remainder of the UEFI drivers and applications in turn call these core services to act on the platform in various ways [13]. For example, a UEFI driver that wanted to retrieve the current time and date would make a request of the DXE core services. The DXE core services relay the request to the Real Time Clock AP which understands the hardware-specific implementation of the real-time clock. Examples of APs include: the CPU AP, Monotonic Counter AP, Timer AP, Reset AP, and the Watchdog Timer AP. To port the UEFI framework to another hardware architecture, it is only necessary to port the APs—the PEI and DXE core services, and those drivers relying on them, do not have to change. Today, UEFI has been ported to the Itanium, x86, AMD64, and ARM microprocessor architectures.

Chapter 3: Compiler Theory

A compiler is a program that translates one language into another [15]. One motivation for creating a compiler is to translate a high level computer language, easy for a human to understand, into a lower-level language, which the computer understands. Translation between formats is another motivation for a compiler—for example, a program that can open a document in Rich Text Format (RTF) and save it as Hypertext Markup Language (HTML) [16].

The job of a compiler is typically broken into two parts:

1. *Lexical analysis,* also called lexing or scanning, is breaking up an input stream of characters into meaningful chunks called tokens;
2. *Syntax analysis,* also called parsing, takes the tokens output by the lexing stage and creates a parse tree in order to facilitate further action

The earliest compilers from the 1950s used ad hoc techniques to scan and translate source code. During the 1960s, compiler theory received a lot of academic attention, and by the early 1970s the concepts of lexical analysis and syntax analysis were well defined [17]. The rest of this chapter provides a brief review of compiler theory and how lexical and syntax analysis will be used to create the language applications described in this report.

3.1 Lexical Analysis

The job of lexical analysis is performed by a program called a lexer. Lexers work by recognizing patterns in characters of an input stream and grouping them together. Lexer rules define how this grouping is accomplished. Though there are several strategies, regular expressions are popularly used to express these rules [18]. When characters from the input stream match the pre-

defined regular expressions, they are classified by the lexer as tokens of the language. For example, the C programming language consists of token types such as whitespace, the various types of operators, e.g., + -> / * & !=, and identifiers. Identifiers are defined as sequences of characters that begin with an alphabetic character or underscore, followed by more alphabetic characters, or underscores, or numerical digits [19]. This can be represented by the following regular expression:

```
[a-zA-Z_] ([a-zA-Z_0-9])*
```

The set of regular expressions that define token rules are collected into a grammar. Rather than write lexers by hand, programmers typically use a lexer generator program that generates lexers based on a provided grammar [2]. Programming languages and natural languages are similar in this regard: English is a language, and it has a particular alphabet—the Latin alphabet. C is a language, and it has a particular alphabet—ASCII characters. Just as not every string of Latin characters is a valid English word, not every string of ASCII characters is a C token [18]. It is the rules of the grammar that define valid vs. invalid, both in English and in C.

In the late 1950s, computer scientists John Backus and Peter Naur developed a convention for expressing computer language grammars that has persisted to this day: Backus-Naur Form (BNF) [20]. Using BNF to express the aforementioned regular expression for identifiers in C looks like:

```
Identifier
      :    Nondigit (Nondigit | Digit)*
      ;
Nondigit
      :    [a-zA-Z_]
```

```
        ;
Digit
        :    [0-9]
        ;
```

This BNF snippet states that an *Identifier* is defined by a required *Nondigit* followed by zero or more *Nondigit*s or *Digit*s. (* represents zero or more, and | represents OR) Reading down recursively, *Nondigit* is defined as any lowercase character a-z or uppercase character A-Z, or the underscore character. Finally, a *Digit* is defined as any character between 0-9.

Lexers perform two operations: [18]

1. Decompose input into substrings corresponding to tokens
2. Record the value (*lexeme*) of the token

Finally, the lexer discards tokens that do not contribute to the syntax analysis stage, for example whitespace and comments [18]. The parser is now ready to accept the set of tokens and take control of the process.

3.2 Syntax Analysis

The tokens from the lexer are fed to another program called a parser for syntax analysis. The parser feeds off the tokens, recognizes sentence structure, and builds an intermediate representation of the input in a data structure called a parse tree [16], also known as a concrete syntax tree [21]. Unlike the token stream, the parse tree records how the parser recognized the structure of the input sentence, including the ordering (association) of the tokens [22].

Just as a lexer must distinguish valid character sequences from invalid character sequences in order to form tokens, a parser must distinguish between valid and invalid strings of tokens in order to form sentences. For example, not all strings of valid C language tokens are C language programs [22]. As in the

case of the lexer, the parser follows the rules in a provided grammar which recursively defines phrases that can make up a valid sentence, and the order in which they must appear [23]. A language is defined as a set of sentences valid for that language; a sentence is made up of phrases, and a phrase is made up of sub-phrases and vocabulary symbols [2].

Parsers are not typically written manually, but are automatically generated by dedicated parser generator programs [16]. Putting lexers and parsers together, we can represent the process graphically as in the following diagram [2]:

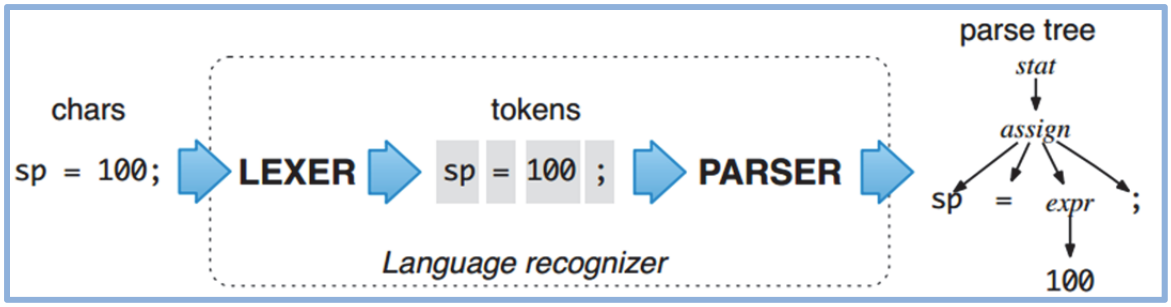

Figure 2: Stages of Compiler Design

3.3 Language Applications with ANTLR

By themselves, lexical and syntax analysis would only be of interest to compiler designers. However, these tools can be put to use by programmers of any domain by using lexer and parser generators to create language applications.

Language applications are programs that leverage lexers and parsers to do useful things like convert legacy source code into a modern equivalent, act as configuration file readers, markup wiki source code, create object-relational database mappings, or inject profiling code into high-level languages [2].

Implementing a language application entails [2]:

1. Writing a grammar to represent the language
2. Generating a lexer and parser for the grammar
3. Writing a program that utilizes the parse tree to read sentences of the language and react appropriately to the phrases and input symbols it discovers

The programmer can treat the auto-generated lexer-parser like a black box—a typical model is to subclass a type of abstract implementation and then override the methods necessary to carry out whatever language application is desired. By operating off parse trees, multiple language applications recognizing the same language can be built, all while reusing the same parse tree code [2].

This report examines the use of the popular lexer-parser generator tool ANTLR to create language applications for UEFI BIOS. ANTLR can automatically generate Listeners [2] and Visitors [24] for a parse tree. ANTLR generates Enter and Exit methods for each node of the parse tree, and then will walk each node in the tree. As ANTLR traverses each node of the parse tree, the applications discussed herein will implement Listener methods to take action on each node in order to create a useful language application.

Chapter 4: Module Complexity

4.1 Software Complexity

The study of Software Complexity has long been popular in the history of software development research. Popular metrics such as McCabe Cyclomatic Complexity and the Halstead Metrics [25] date back to the 1970s. There are several reasons for wanting to measure software complexity:

- *Gauge test effort*: When changes to software modules are made, it is advantageous to rank them according to complexity so that appropriate effort can be applied to testing—avoidance of either the risk of too little testing, or the waste of time spent on testing trivial changes.
- *Provide focus while debugging*: It is more likely for bugs to exist in complex code rather than simple code. Therefore, measuring software complexity can help narrow down a debugging effort by prioritizing the most complex software modules first.
- *Provide focus for resource intensive investigation*: Time and resource consuming activities such as peer code reviews and static source code analysis may not be practical for all software modules. By ranking modules according to their complexity, these more powerful but more resource-intensive tools can be put to their most efficient use.
- *Maintainability*: The more complex the code, the more difficult to maintain. Gauging software complexity, therefore, can help with the efficient assignment of developers to maintain a variety of software modules.

Software complexity is of special importance for BIOS. A typical BIOS source code repository is several million lines of code, maintained by a distributed partnership of hardware vendors, independent BIOS vendors, peripheral manufacturers, and system builders. Thus, gauging complexity in order to use test resources efficiently, focus debugging, and allocate resources is imperative. The intent of this chapter is to combine a general-purpose software complexity metric with UEFI BIOS domain-specific complexity metrics in order to best capture the complexity of UEFI BIOS software modules.

4.2 Definition: UEFI Image

The UEFI framework specifies two binary executable modules: Pre-EFI Initialization Modules (PEI-M) and Driver Execution Environment (DXE) drivers. This report will refer to these executable code modules collectively as UEFI images. It is the intent of this section to create a language application that parses the source code of UEFI images for the purpose of gauging their software complexity.

4.3 How to Gauge Complexity

The work of analyzing the complexity of UEFI images will be performed on a sample of ninety-three UEFI images from a major computer manufacturer. These UEFI images implement advanced, proprietary features for desktop and portable computers; i.e., they would not be classified as routine, trivial, or simple chipset-enabling UEFI images. The UEFI images in question were analyzed for the period July 2012 to March 2014. They were developed for UEFI BIOS products that began development in July 2012, shipped to customers about a year later, and then entered a Sustaining Engineering phase up until March 2014.

The following five subsections list the particular vectors of complexity selected for the analysis, and justification for each:

1. Protocol Handler Services
2. Variable Services
3. Event and Timer Callbacks
4. Block Device Accesses
5. Dynamic Memory Allocation

The various vectors that influence the complexity of UEFI images share two common denominators: a) access of the System Table, and b) access of the Handle Database. The System Table is a system-wide data structure providing UEFI images with access to various boot and runtime services via function pointers, as illustrated in the following figure [26]:

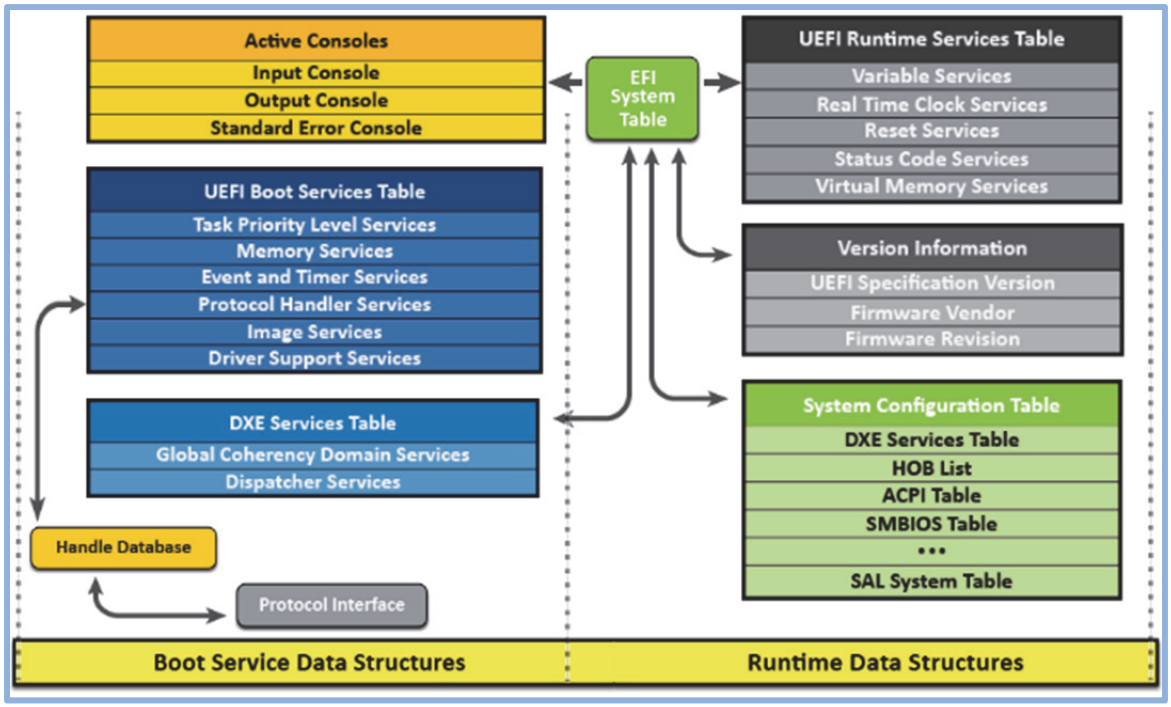

Figure 3: The System Table

A pointer to the System Table is provided to the entry point of every UEFI image.

The Handle Database is a system-wide data structure that acts like a central directory for all the various objects maintained by the UEFI BIOS for use by UEFI images. UEFI images can utilize the System Table to gain access to the Handle Database and search therein for needed resources [13].

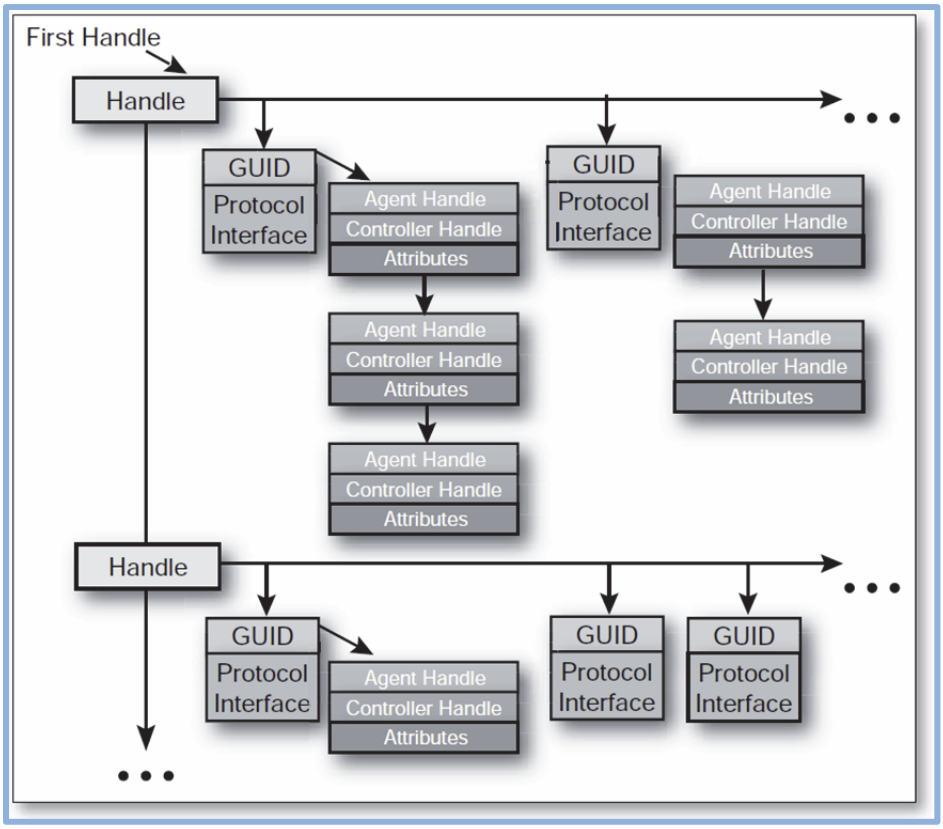

Figure 4: The Handle Database

References to the System Table and Handle Database are significant because each access requires the UEFI image to place a dependency on an external image, leave its own memory space, trust the operation of the external image, and return to its own memory space. This behavior increases the data complexity and coupling of UEFI images.

4.3.1 Protocol Handler Services

UEFI defines a protocol as a structure of function pointers and optionally some data. In reference to object-oriented programming, a protocol can be seen as analogous to a class that provides methods and instance variables. UEFI images can publish protocols for other images to consume, and can consume protocols published by other images. The Handle Database serves as the central repository for registration of protocols available to the system [13].

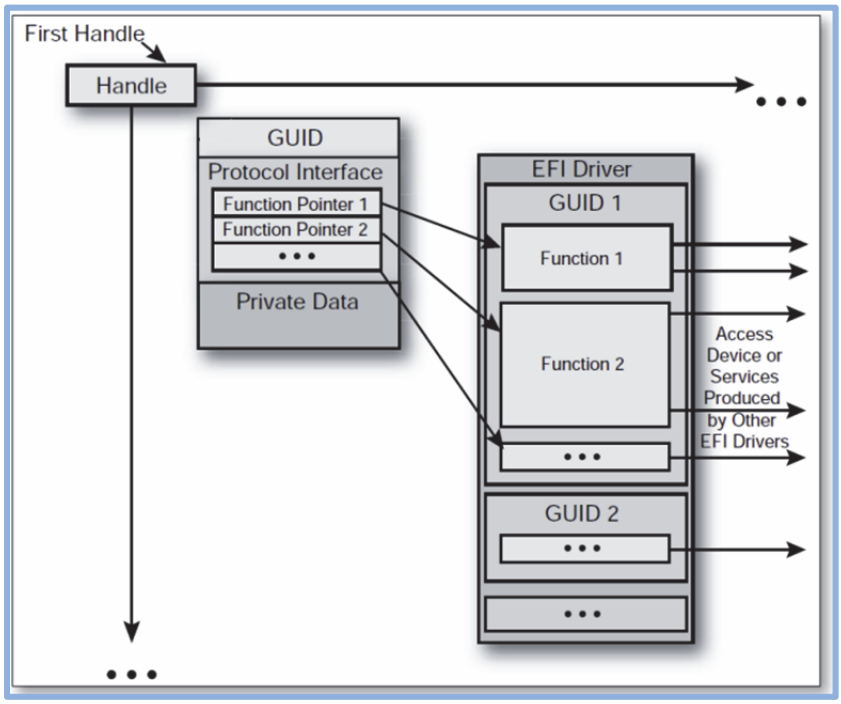

Figure 5: A Protocol

The complexity application will pay special attention to the various protocol handler services because of the overhead of accessing the System Table, locating or publishing the desired protocol, and then accessing the code and data of another UEFI image in order to get work done. The application will check for the following protocol handler services:

```
protocolHandlers.add("InstallProtocolInterface");
```

```
protocolHandlers.add("UninstallProtocolInterface");
protocolHandlers.add("ReinstallProtocolInterface");
protocolHandlers.add("RegisterProtocolNotify");
protocolHandlers.add("LocateHandle");
protocolHandlers.add("HandleProtocol");
protocolHandlers.add("LocateDevicePath");
protocolHandlers.add("OpenProtocol");
protocolHandlers.add("CloseProtocol");
protocolHandlers.add("OpenProtocolInformation");
protocolHandlers.add("ConnectController");
protocolHandlers.add("DisconnectController");
protocolHandlers.add("ProtocolsPerHandle");
protocolHandlers.add("LocateHandleBuffer");
protocolHandlers.add("LocateProtocol");
protocolHandlers.add("InstallMultipleProtocolInterfaces");
protocolHandlers.add("UninstallMultipleProtocolInterfaces");
```

4.3.2 Variable Services

Variables are defined as key/value pairs with associated attributes. They are typically non-volatile and stored in a (very slow to access) Serial Peripheral Interface (SPI) ROM. Due to the relatively long access times of SPI parts, the complexity of UEFI images is increased if they need to wait to retrieve data from ROM. Furthermore, reading runtime data into the UEFI image introduces the risk of unexpected values potentially corrupting the program state. The complexity application will check for the following variable access routines:

```
variableServices.add("GetVariable");
variableServices.add("GetNextVariableName");
variableServices.add("SetVariable");
variableServices.add("QueryVariableInfo");
```

4.3.3 Event and Timer Callbacks

UEFI images can create events and register for callback to other images' events. There is a rich set of functions related to creating and signaling events, including timer-based events.

Events processing adds additional complexity to UEFI images because of System Table accesses, the dependency on external UEFI images to properly signal events, and the lack of certainty as to when, precisely, a depended-upon event will get triggered—a source of risk for time-sensitive UEFI images. Moreover, images that signal events assume that their subscribers' callback functions are well-behaved. The complexity application will check for the following event services:

```
callbacks.add("CreateEvent");
callbacks.add("CreateEventEx");
callbacks.add("CloseEvent");
callbacks.add("SignalEvent");
callbacks.add("WaitForEvent");
callbacks.add("CheckEvent");
callbacks.add("SetTimer");
callbacks.add("RaiseTPL");
callbacks.add("RestoreTPL");
```

4.3.4 Block Device Accesses

The UEFI Driver Model features a layered driver architecture such that devices, device types, and the buses they live on can be abstracted from one another. In this model, for example, all block devices support a common interface which any UEFI code can call, no matter whether the block device is a hard drive connected to a SATA controller via a PCI bus, or a flash RAM device connected via USB.

The complexity measurement will take into account any instances of the UEFI image communicating with a block device. Such communication greatly adds to the unpredictability of the UEFI image due to high latencies, the potential absence of media in the drive, and any errors the device may report. The application will check for the instantiation of either of these block device protocols:

```
blockIo.add("EFI_BLOCK_IO_PROTOCOL");
blockIo.add("EFI_BLOCK_IO2_PROTOCOL");
```

4.3.5 Dynamic Memory Allocation

The UEFI BIOS dynamically maintains the system's memory map during boot. UEFI images allocate and free memory they need using functions provided by the System Table. Before an executing UEFI image can exit and relinquish control, it must free all memory resources it has been granted, including memory pages, pool allocations, and open file handles [12]. There is no concept of garbage collection in UEFI BIOS.

Dynamic memory allocation has traditionally been a source of programming errors in every domain of software engineering. The complexity application will capture the impact of dynamic memory allocation by looking for the following memory allocation services:

```
memAlloc.add("AllocatePages");
memAlloc.add("FreePages");
memAlloc.add("GetMemoryMap");
memAlloc.add("AllocatePool");
memAlloc.add("FreePool");
```

4.4 Approach to the Complexity Analyzer Application

In this chapter, we will leverage the standard, canonical C language grammar provided by the ANTLR project [19] in order to create a new language application.

4.4.1 Class Hierarchy

The standard C language grammar file was fed to ANTLR to create the auto-generated lexer and parser classes, shown in the following UML diagram as `UefiLexer` and `UefiParser`. We write ourselves the class to perform the real work, called `ComplexityAnalyzer`. `ComplexityAnalyzer` subclasses `UefiBaseListener`, which implements the auto-generated `UefiListener` interface, which itself is a subclass of ANTLR's `ParseTreeListener`. The complete Java class listing is found in Appendix A. The UML diagram:

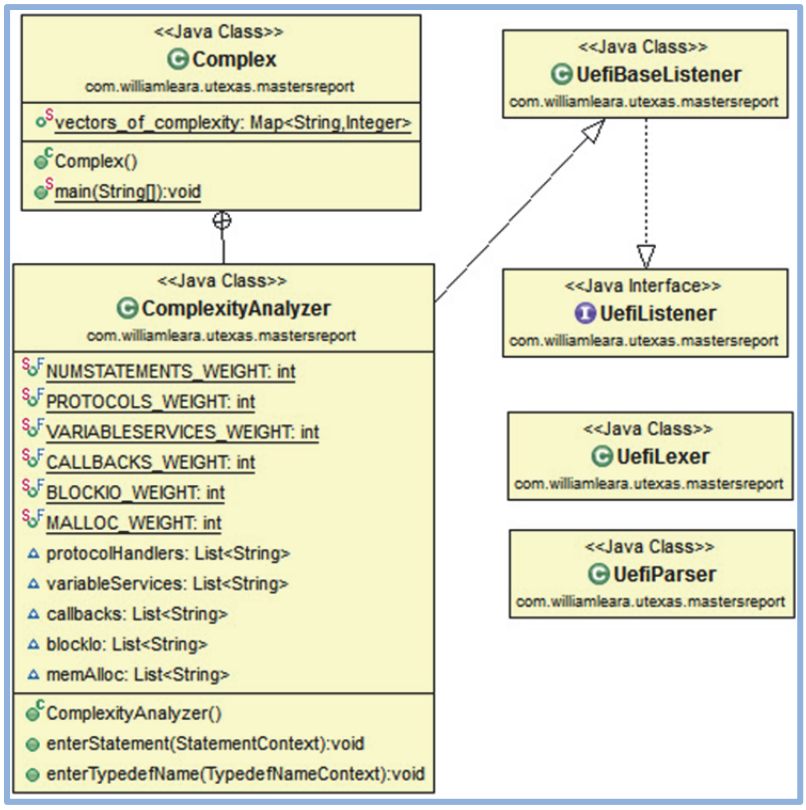

Figure 6: UML Diagram for ComplexityAnalyzer

`UefiBaseListener` creates Enter and Exit methods that get called upon each entry and exit of every node in the parse tree. In `ComplexityAnalyzer`, only those methods required to take actions relevant to the complexity application are overridden.

4.4.2 Vectors of Complexity HashMap

There is a global `HashMap` created for the application which stores the cumulative results of the analysis for each vector of complexity:

```
// data structure for parsing results
Map<String, Integer> vectors_of_complexity = new HashMap<String, Integer>();
```

The constructor for ComplexityAnalyzer initializes each vector of *`vectors_of_complexity`* to zero:

```
// initialize data structure
vectors_of_complexity.put("NumStatements", 0);
vectors_of_complexity.put("Protocols", 0);
vectors_of_complexity.put("VariableServices", 0);
vectors_of_complexity.put("Callbacks", 0);
vectors_of_complexity.put("BlockIo", 0);
vectors_of_complexity.put("MemAlloc", 0);
```

As the analysis of each translation unit (`.c` file) of the UEFI image proceeds, the value of each vector (the key in the `HashMap`) is incremented when a matching instance is discovered.

4.4.3 Definitions of Each Vector of Complexity

Each complexity vector is represented by an `ArrayList<String>`, which contains the different strings (typically function names) for the parser to match. They are defined in the following manner:

```
// data structures for each vector
List<String> protocolHandlers = new ArrayList<String>();
List<String> variableServices = new ArrayList<String>();
List<String> callbacks = new ArrayList<String>();
List<String> blockIo= new ArrayList<String>();
List<String> memAlloc= new ArrayList<String>();
```

The `ComplexityAnalyzer` constructor initializes them; as an example, the Variable Services `ArrayList` is initialized with the following function names, representing the UEFI Specification-defined functions that manipulate UEFI variables:

```
// initialize ArrayList:  variable services
variableServices.add("GetVariable");
variableServices.add("GetNextVariableName");
variableServices.add("SetVariable");
variableServices.add("QueryVariableInfo");
```

Keeping the function names in an `ArrayList` separates them from the code in the listener. Therefore, the strings to listen for can change without affecting the code which does the listening.

4.4.4 Weighting System

It may be necessary to tune the analysis in order to get accurate results. For example, after empirical research it may become apparent that one of the vectors determines the complexity of UEFI images more strongly than another. To be able to make these kinds of adjustments, the algorithm can be adjusted by changing each vector's respective weight in order to boost the influence of one or more vectors. In the following example, all the vectors have the same weight:

```
// weights for each vector of complexity
public static final int NUMSTATEMENTS = 1;
public static final int PROTOCOLS = 1;
public static final int VARIABLESERVICES = 1;
public static final int CALLBACKS = 1;
public static final int BLOCKIO = 1;
public static final int MALLOC = 1;
```

The Memory Allocation vector could take on a heavier weight by making the following adjustment:

```
public static final int MALLOC = 3;
```

4.4.5 Listener: Number of Statements

A classic software complexity metric is simply to count the number of statements. In order to baseline the domain-specific measures of complexity with a traditional general-purpose measure of complexity, the `ComplexityAnalyzer` class also counts statements. The `enterStatement()` method is overridden and increments the respective counter in the *`vectors_of_complexity`* `HashMap` upon finding a new statement:

```
// listener for number of statements
public void enterStatement(UefiParser.StatementContext ctx) {
    vectors_of_complexity.put("NumStatements",
    vectors_of_complexity.get("NumStatements") + 1*NUMSTATEMENTS);
}
```

4.4.6 Listener: Expressions

The rest of the vectors of complexity are listened for by overriding the `enterTypedefName()` method:

```
// listener for expressions
public void enterTypedefName(UefiParser.TypedefNameContext ctx) {

// protocol handler services
if (protocolHandlers.contains(ctx.getText()))
    vectors_of_complexity.put("Protocols",
    vectors_of_complexity.get("Protocols") + 1*PROTOCOLS);
// variable services
else if (variableServices.contains(ctx.getText()))
    vectors_of_complexity.put("VariableServices",
    vectors_of_complexity.get("VariableServices") + 1*VARIABLESERVICES);
// event and timer callbacks
else if (callbacks.contains(ctx.getText()))
    vectors_of_complexity.put("Callbacks",
    vectors_of_complexity.get("Callbacks") + 1*CALLBACKS);
// block io
else if (blockIo.contains(ctx.getText()))
    vectors_of_complexity.put("BlockIo",
    vectors_of_complexity.get("BlockIo") + 1*BLOCKIO);
// memory allocation
else if (memAlloc.contains(ctx.getText()))
    vectors_of_complexity.put("MemAlloc",
    vectors_of_complexity.get("MemAlloc") + 1*MALLOC);
}
```

Notice that in each case the value to increment can be boosted by changing the value of the weight.

4.4.7 main()

The `main()` function has three parts: 1) read in one or more `.c` files; 2) loop through each `.c` file, performing the complexity analysis and updating the `HashMap`; 3) print the results. These three parts are discussed in the following subsections.

4.4.7.1 Read .c File(s)

```
// check for input file
String inputFile = null;
if (args.length <= 0) {
  System.out.println("ERROR: missing input file");
  System.exit(1);
}
```

4.4.7.2 Process Each .c File

```
// main program loop
ComplexityAnalyzer cmplx = new ComplexityAnalyzer();
for (int i=0; i<args.length; i++) {
      inputFile = args[i];
    InputStream is = System.in;
    if ( inputFile != null )
      is = new FileInputStream(inputFile);

    // create lexer and parser
    UefiLexer lexer = new UefiLexer(new ANTLRInputStream(is));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    UefiParser parser = new UefiParser(tokens);
    parser.setBuildParseTree(true);
    ParserRuleContext tree = parser.compilationUnit();
    ParseTreeWalker walker = new ParseTreeWalker();

    // perform analysis
    walker.walk(cmplx, tree);
}
```

4.4.7.3 Print Results

```
// print results
System.out.println("Component Scores:");
```

```
System.out.println("NumStatements: " +
      vectors_of_complexity.get("NumStatements"));
System.out.println("Protocols: " +
      vectors_of_complexity.get("Protocols"));
System.out.println("VariableServices: " +
      vectors_of_complexity.get("VariableServices"));
System.out.println("Callbacks: " +
      vectors_of_complexity.get("Callbacks"));
System.out.println("BlockIo: " + vectors_of_complexity.get("BlockIo"));
System.out.println("MemAlloc: " +
      vectors_of_complexity.get("MemAlloc"));

System.out.println("\nComposite Score:");
System.out.println(
      vectors_of_complexity.get("NumStatements") +
      vectors_of_complexity.get("Protocols") +
      vectors_of_complexity.get("VariableServices") +
      vectors_of_complexity.get("Callbacks") +
      vectors_of_complexity.get("BlockIo") +
      vectors_of_complexity.get("MemAlloc")
);
```

4.5 Analysis of the Complexity Metrics

In order to assess the effectiveness of the module complexity algorithm used in the `ComplexityAnalyzer` class, it is necessary to benchmark it to other, more objective, measures. It is the goal of this section to measure the complexity scores output by `ComplexityAnalyzer` against two criteria: 1) the number of production releases of each module; 2) the number of source code changesets committed per module. The assumption is that there is a direct relationship between the number of times a UEFI module needs to be released and the complexity of the module; and also a direct relationship between the number of changesets committed to a module and the complexity of that module.

4.5.1 Number of Production Releases

The UEFI images under analysis were updated in order to either fix defects or implement new functionality. In the beginning stages of development, the emphasis is on new functionality and porting previous generation code to a new generation. Later in the development process, and especially after the products are shipped to customers, the emphasis changes to defect fixes.

The following graph shows the number of releases for the top thirty-two most popularly released UEFI images:[2]

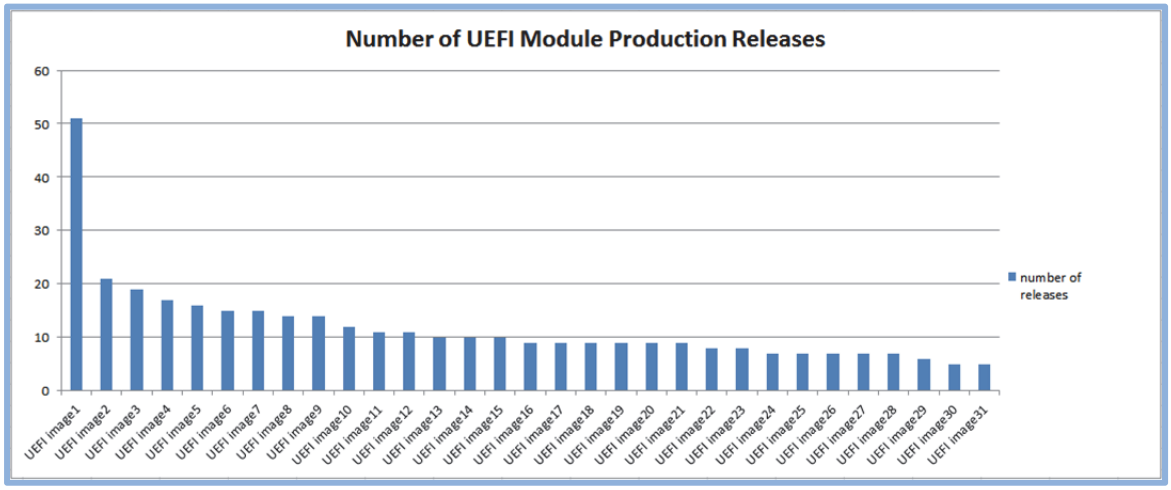

Figure 7: Number of UEFI Image Releases

4.5.2 Number of Changesets Committed

Another related, but different, metric to benchmark the software complexity algorithm is the number of changesets committed to the source code repository for each UEFI image. The assumption is that the more often the source code has to change, the more complex and brittle the UEFI image.

The following graph shows the number of changesets committed to the top thirty-two most popularly changed UEFI images:[3]

[2] This data can be found in table form in Appendix B

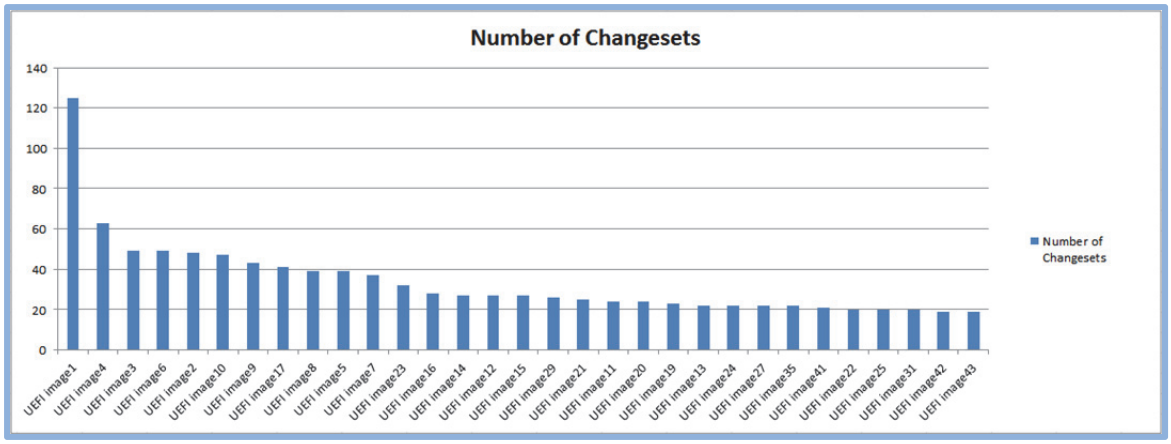

Figure 8: Number of Changesets Committed per Image

4.5.3 Agreement Between Number of Releases/Chanesets

There is substantial agreement between the number of production releases and the number of changesets committed to the UEFI images. Of the top thirty-two images, twenty-seven exist on both lists; only five do not. Specifically, the drivers existing in the top thirty-two most often released list but not in the most often changed list are:

- UEFI image18
- UEFI image26
- UEFI image28
- UEFI image30
- UEFI image32

The UEFI modules existing in the top thirty-two most often changed but not in most often released are:

- UEFI image35
- UEFI image41
- UEFI image42

[3] This data can be found in table form in Appendix C

- UEFI image43
- UEFI image47

4.5.4 Agreement With Complexity Scores

This final subsection of the chapter on Module Complexity explains how the `ComplexityAnalyzer` Java application was run, the results, and a summary of the conclusions.

4.5.4.1 Running the Algorithm

Running the analysis consisted of several interesting scripts. First was to run a regular expression search and replace in all the Microsoft Visual C project files in order to turn on the preprocessor flags `PreprocessToFile` and `PreprocessSuppressLineNumbers`. The ANTLR C grammar is specific to the C language, which technically is distinct from the grammar of the C preprocessor. Therefore, in order to resolve all the `#define`, `#include`, and other preprocessor directives, the C compiler needed to be configured to run through the `.c` files, run just the preprocessor without compiling or linking, and output the result into a new file. These preprocessed files would then be acceptable to the ANTLR C grammar and an analysis of their complexity made.

Once these preprocessor flags were set, a script was run to build all the projects, which in this case means to run the preprocessor on all the `.c` files:

```
@echo off
for /f %%a in (cmplx_prjs.txt) do
msbuild %%a /t:Rebuild /p:Configuration=Release /verbosity:q
```

The output of the preprocessor-only step is a series of text files with a `.i` extension. The catalog of ninety-three UEFI images produced a total of 842 `.i` files, the largest being 717KB and over 40000 lines long. Next a script was run to

extract the `.i` files from their location in each build's intermediate output directory and copy them to a unique location for each UEFI image:

```
@echo off
setlocal enabledelayedexpansion
for /f %%a in (cmplx_modules.txt) do (
    cd %%a
    set mod_name=%%a
    set mod_name=!mod_name:C:\Users\WilliamLeara\src\modules\=!
    set mod_name=!mod_name:\Trunk\module=!
    mkdir c:\users\williamleara\ifiles\!mod_name!
    for /r %%f in (*.i) do @copy /y "%%f"
      c:\users\williamleara\ifiles\!mod_name!
    )
endlocal
```

Finally, the `ComplexityAnalyzer` application was run on each UEFI image-specific directory, producing a `score.txt` file:

```
@echo off
for /f %%a in (mod_list.txt) do java
      com.williamleara.utexas.mastersreport.Complex l:\ifiles\%%a\*.i >
      l:\ifiles\0scores\%%a_score.txt
```

For example, the `score.txt` for the UEFI image9 looks like:

```
Component Scores:
NumStatements: 3282
Protocols: 806
VariableServices: 200
Callbacks: 395
BlockIo: 0
MemAlloc: 243

Composite Score: 4926
```

The entire process to build and analyze all ninety-three UEFI images took about seven minutes on an eight-core 3GHz Intel Core i7 Extreme Edition processor-based machine.

4.5.4.2 *Results*

The following table lists the results sorted by score, high to low. The first column is the name of the UEFI image, the second is its composite score output by the algorithm, and the last two columns signify whether the UEFI image was included in the list of top thirty-two most-often released images and top thirty-two most-often changed images, respectively.

Table 2: UEFI Module Complexity Scores

UEFI image name	score	#Rel	#Chg
UEFI image1	383471	X	X
UEFI image27	14817	X	X
UEFI image24	6405	X	X
UEFI image4	5855	X	X
UEFI image85	5392		
UEFI image7	5145	X	X
UEFI image9	4926	X	X
UEFI image8	4295	X	X
UEFI image3	3341	X	X
UEFI image23	2954	X	X
UEFI image6	2779	X	X
UEFI image10	2732	X	X
UEFI image2	2703	X	X
UEFI image18	2181	X	
UEFI image17	2079	X	X
UEFI image39	2078		
UEFI image87	1928		
UEFI image36	1531		
UEFI image47	1494		X
UEFI image14	1493	X	X
UEFI image80	1406		
UEFI image21	1224	X	
UEFI image38	1208		
UEFI image12	1151	X	X
UEFI image15	1120	X	
UEFI image74	883		
UEFI image42	841		X
UEFI image33	829		
UEFI image22	791	X	X
UEFI image20	791	X	X
UEFI image31	788	X	X
UEFI image57	765		
UEFI image32	758	X	X
UEFI image25	745	X	X
UEFI image5	687	X	X
UEFI image56	671		
UEFI image52	636		
UEFI image76	621		
UEFI image77	600		
UEFI image13	574	X	X
UEFI image19	524	X	X
UEFI image45	481		
UEFI image92	471		
UEFI image16	461	X	X
UEFI image72	430		
UEFI image70	429		
UEFI image93	411		
UEFI image59	409		
UEFI image50	398		
UEFI image78	377		
UEFI image46	377		
UEFI image90	367		
UEFI image63	366		

Table 2, cont.

UEFI image48	350		
UEFI image67	297		
UEFI image11	292	X	X
UEFI image54	276		
UEFI image64	266		
UEFI image68	266		
UEFI image28	249	X	
UEFI image41	248		X
UEFI image75	245		
UEFI image43	240		X
UEFI image79	232		
UEFI image82	228		
UEFI image49	227		
UEFI image44	215		
UEFI image88	215		
UEFI image30	213	X	
UEFI image51	207		
UEFI image91	193		
UEFI image34	191		

UEFI image65	181		
UEFI image29	169	X	
UEFI image26	150	X	
UEFI image66	141		
UEFI image83	138		
UEFI image62	126		
UEFI image40	122		
UEFI image81	118		
UEFI image53	101		
UEFI image84	96		
UEFI image71	88		
UEFI image73	80		
UEFI image86	66		
UEFI image69	57		
UEFI image55	53		

4.5.4.3 Summary

Of the top thirty-two most popularly released UEFI modules, twenty-one of them were also so-ranked by the `ComplexityAnalyzer` algorithm. Of the top thirty-two UEFI modules with the most changesets, twenty of them we also so-ranked by the `ComplexityAnalyzer` algorithm.

There is one outlier that needs to be addressed. The UEFI Image85 scored high (number five) on the algorithmic scoring, but was not in the top thirty-two most often released/changed. After further review, this is a large library that was delivered by a third party as a turn-key solution. Therefore, it has not been released or changed frequently, but its high complexity score is likely warranted.

Looking further down the list, twenty-eight of the top thirty-two most often released are found within the top forty-five UEFI images ranked by the algorithm. Twenty-seven of the top thirty-two most often changed are found with the top forty-five UEFI images ranked by the algorithm. These results indicate a good clustering of results between the algorithm and benchmarks.

Chapter 5: Visual Forms Representation

5.1 Background

Another innovation of UEFI BIOS over Legacy BIOS is in the specification of building blocks for user interfaces. There are various scenarios where a platform component might want to interact in some fashion with the user. Examples of this are when presenting a user with several choices of information, (e.g., a boot menu) sending information to the display, (e.g., system status, logo) or offering a user menus for configuring the system—a BIOS Setup program [12]:

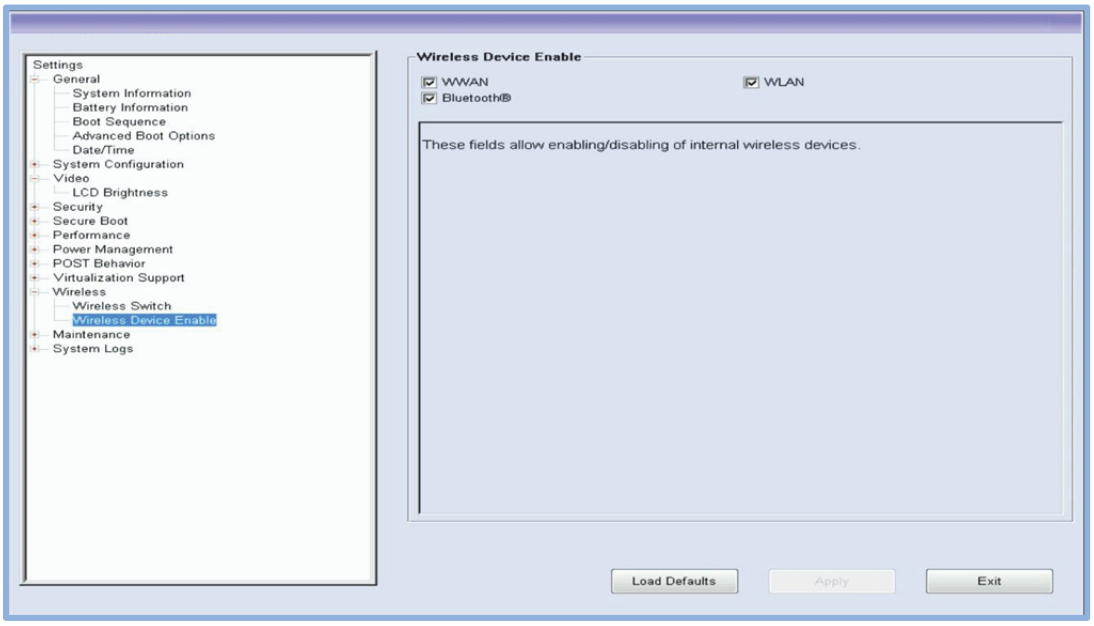

Figure 9: An Example of a BIOS Setup Program

The potentially arduous task of managing user interface elements is simplified for UEFI modules requiring such functionality by UEFI providing some basic graphical elements like forms, strings, images, and fonts.

UEFI calls this user interface support Human Interface Infrastructure (HII). HII is a set of protocols that allow a UEFI module to register user interface and configuration content with the BIOS [12].

5.1.1 The HII Database

The HII database is the resource that serves as the repository of all the form, string, image, and font data for the system. Drivers that contain information destined for the end user will store their data in the HII database.

For example, one UEFI module might implement the BIOS Setup program, allowing the user to configure motherboard component settings. Additionally, add-in cards may contain their own UEFI drivers, which, in turn, have their own BIOS Setup-related data. All the UEFI modules that contain BIOS Setup-related data can include their information in the HII database [12]. This architecture is summarized in the following figure:

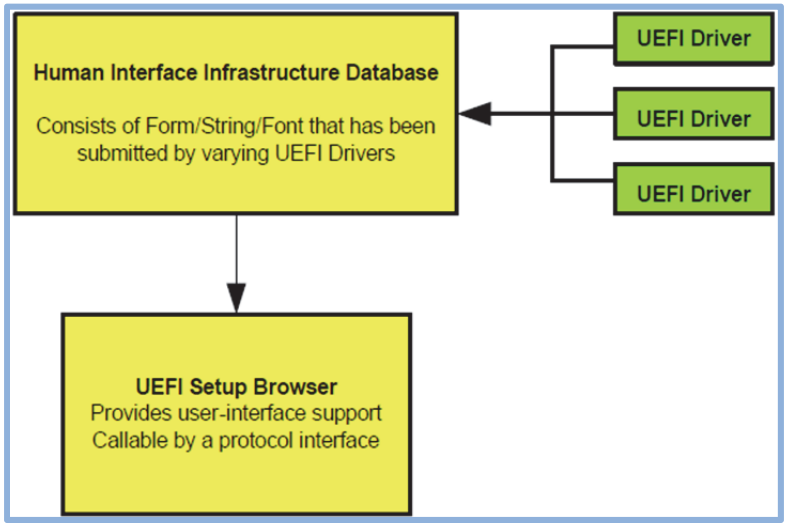

Figure 10: Human Interface Infrastructure

5.1.2 Forms

The UEFI specification describes how a UEFI module can present a forms-based interface to a user, a UI element akin to Windows' `CFrameWnd` or Java's `JFrame`. This forms-based interface assumes that each window or screen consists of some window overhead, such as a title and buttons, and a list of user interface controls. For example, these controls could represent individual configuration settings for a UEFI application or driver [12].

Associated with the notion of a form is a Forms Browser—the entity that reads the form data and presents a graphical representation on the display. The Forms Browser provides a forms-based renderer which understands how to read the contents of the forms, interact with the user, and save any resulting values. The Forms Browser uses forms data installed by UEFI modules in the HII database. The Forms Browser organizes the forms so that a user can navigate between the forms, select individual controls, and change the values using a keyboard, touch digitizer, or mouse. When the user has finished making modifications, the Forms Browser saves the values to NVRAM [12].

5.1.3 Strings

Strings in the UEFI environment are defined using the 16-bit UCS-2 character encoding. Strings are another one of the types of resources installed into the HII Database. In order to facilitate localization, programmers reference each string by a unique identifier defined as part of the strings package installed by the UEFI image. Each identifier may have several translations associated with it, e.g., English, French, and Traditional Chinese. When displaying a string, the Forms Browser selects the text to display based on the current platform language setting [12].

The actual text for each language is stored in a separate file, which makes it possible to add and remove language support just by including or excluding language-specific files. Moreover, each string may have font information, including the font family name, font size and font style, associated with it [12].

5.1.4 Images

UEFI supports storing images in the HII database. The format of images stored in the HII database was created to conform to the industry standard 1-bit, 4-bit, 8-bit, and 24-bit video memory layouts [12].

5.1.5 Fonts

UEFI specifies a standard font which is required for all systems that support text display on bitmapped output devices. The standard font, named "`system`", is a fixed pitch font, where all characters are either narrow (8x19 pixels) or wide (16x19 pixels). UEFI also allows for the display of other fonts, both fixed-pitch and variable-pitch. Platform support for fonts beyond `system` is optional [12].

5.2 VFR Grammar

An ANTLR grammar was created for the VFR language. The grammar is based on the *VFR Programming Language* specification, published by Intel Corporation as part of the EFI Development Kit II.[4] The specification is quite lengthy—sixty pages of BNF-styled rules. The ANTLR grammar created to meet this specification is over 1200 lines long, or about 50% larger than the ANTLR

[4] https://sourceforge.net/projects/edk2/files/Specifications/VFR_V1.7.pdf/download

grammar for the C programming language. Due to its length, a walk-through of the grammar is not practical. The complete grammar is found in Appendix D.

To briefly demonstrate the grammar, consider this extremely simple VFR form definition:

```
form formid = 0x8d71,
    title = STRING_TOKEN(0x1);

    subtitle text = STRING_TOKEN(0x2);
    subtitle text = STRING_TOKEN(0x3);

    grayoutif ideqval SetupAccess == 0,
    checkbox varid  = WirelessEnableWLAN,
        prompt   = STRING_TOKEN(0x8),
        help     = STRING_TOKEN(0x9),
        default value = 1,
    endcheckbox;
    endif;
endform;
```

This VFR code represents a form that allows a user to select or deselect a checkbox to enable/disable a Wireless LAN (WLAN) radio.

The form has a `title`, defined by the string designated `0x1`. There are two `subtitles` to the form defined by the strings `0x2` and `0x3`. VFR's `grayoutif` command stipulates scenarios in which the checkbox should be shown to the user but grayed out. In this case, the checkbox is grayed out when a variable called `SetupAccess` is set to `0`. Next, the `checkbox` is defined: its text is defined by the `prompt` keyword, which resolves to string `0x8`. There is `help` text associated with the checkbox which is defined by string `0x9`. Finally, the checkbox's default value is `1`.

Graphically depicting how the ANTLR grammar interprets this VFR code provides a good summary of its architecture. Running ANTLR's `TestRig`[5] tool produces the following parse tree:

[5] http://www.antlr.org/api/Java/org/antlr/v4/runtime/misc/TestRig.html

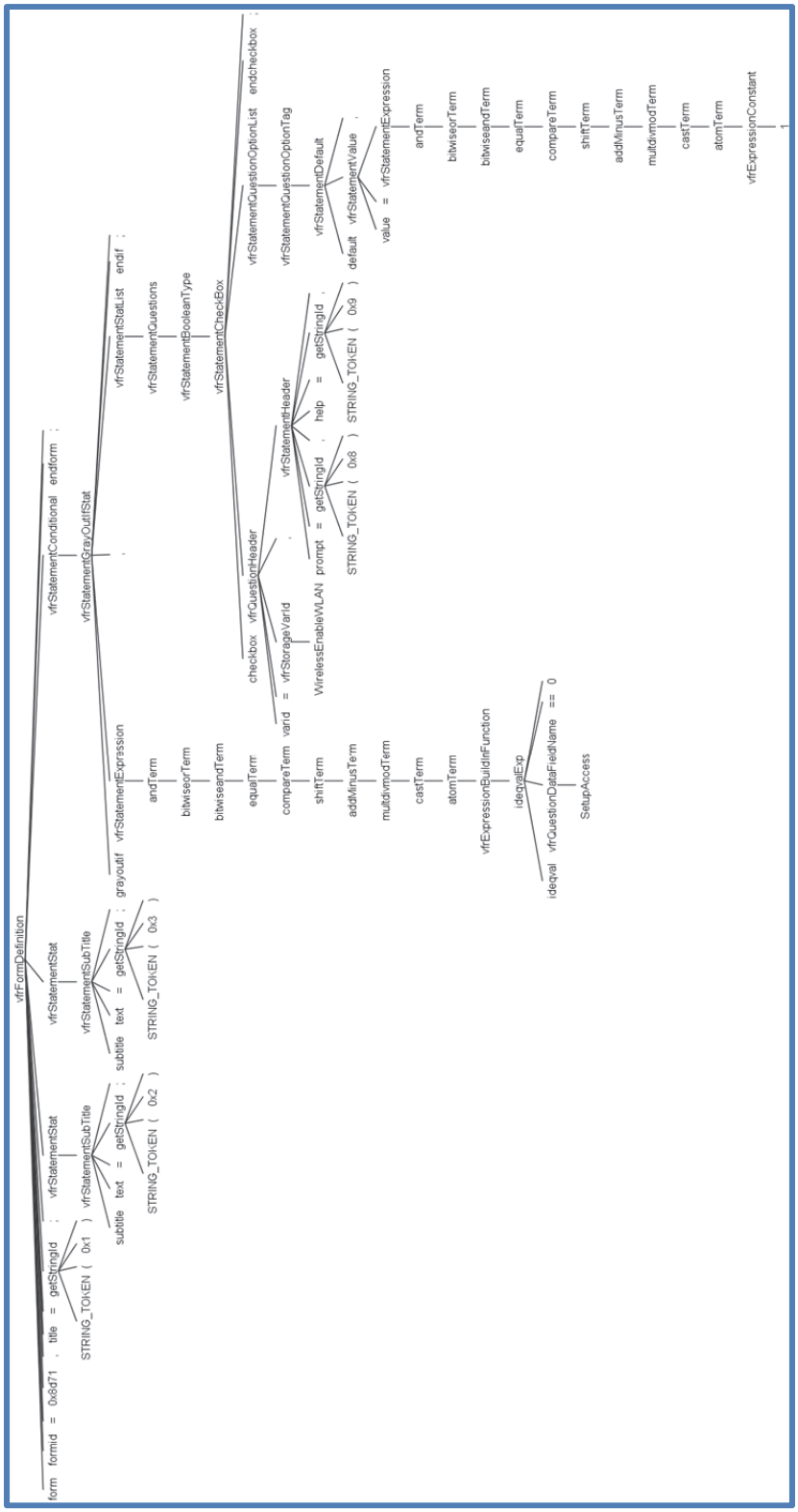

Figure 11: A Sample VFR Parse Tree

5.3 VFR HTML Preview Application

5.3.1 Justification

Unlike modern integrated development environments (IDE) like Eclipse or Microsoft Visual Studio, there is no IDE for VFR development. In order to see how the VFR code for the forms, strings, and images will actually appear, a developer has to compile the VFR, compile a BIOS image containing the compiled VFR, flash the BIOS on a machine, and boot the machine into the UEFI application that renders the VFR code. This process can typically take fifteen or more minutes. It would be desirable, therefore, to have a way to get a quick preview of the VFR before running through the entire build and flash process. This chapter of the report demonstrates a simple application to export VFR code to HTML so it can be instantly previewed by a web browser.

5.3.2 Approach to the VFR Application

The following subsections describe the strategy taken in implementing the VFR application. The VFR grammar created for ANTLR generates the `vfrLexer`, `vfrListener`, `vfrParser`, and `vfrBaseListener` classes. The `Vfr` and `VfrToHTML` classes were written to do the work of the HTML application. The following UML diagram depicts the class hierarchy:

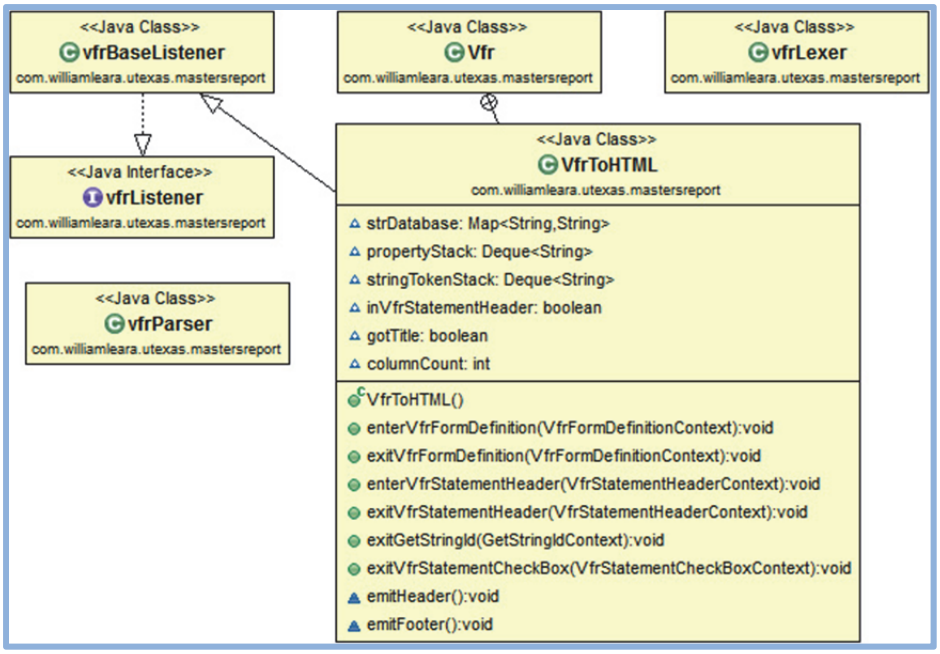

Figure 12: UML Diagram for VFR to HTML Application

5.3.2.1 String Database

As mentioned above in the section on VFR strings, the strings used in VFR forms are stored in separate files and referenced in VFR source by a numbered code. The VFR source needs to resolve the numbered code-to-string mapping in order to output the correct string. The VFR to HTML application follows the same model; the strings file used in the example looks like:

```
0x1, Wireless Device Enable
0x2, UWB
0x3, WWAN
0x4, WLAN
0x5, Bluetooth
0x6, The UWB checkbox enables/disables the UWB radio
0x7, The WWAN checkbox enables/disables the WWAN radio
0x8, The WLAN checkbox enables/disables the WLAN radio
0x9, The Bluetooth checkbox enables/disables the Bluetooth radio
```

The VfrToHTML constructor will locate the strings file and load the strings into a HashMap where the key is the string ID number and the value is the actual string.

```
// database of identifier to string mappings
Map<String, String> strDatabase = new HashMap<String, String>();

try {
    reader = new BufferedReader(new FileReader(f));
    String stringMapping = null;
    while ((stringMapping = reader.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(stringMapping, ",");
      String key = st.nextToken();
      String value = st.nextToken();
      strDatabase.put(key, value);
    }
```

This way, when the listener methods discover strings during traversal of the parse tree, they can retrieve the string's value from the HashMap using the number code and output the correct string to the screen.

5.3.2.2 Creation of HTML Skeleton

As the parser enters and exits a `vfrFormDefinition`, listeners create the header and footer of the HTML pages, respectively. The HTML preview page is represented as a Formset with two frames: a Head frame, which lists the user controls, and a Body frame, which lists the corresponding help text.

```
// write beginning contents of the main frameset
try {
      mainWriter = new PrintWriter(new BufferedWriter(new
      FileWriter(fileNameMain, true)));
} catch (Exception ex) {
      System.out.println(ex);
}
```

```
mainWriter.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0
      Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
      transitional.dtd\">");
mainWriter.write("<html xmlns=\"http://www.w3.org/1999/xhtml\">");
mainWriter.write("<frameset rows=\"150,*\">");
mainWriter.write("<frame src=\"head.html\" scrolling=\"no\">");
mainWriter.write("<frame src=\"body.html\" scrolling=\"no\">");
mainWriter.close();

// write beginning contents of the header frame
try {
      headWriter = new PrintWriter(new BufferedWriter(new
      FileWriter(fileNameHead, true)));
} catch (Exception ex) {
      System.out.println(ex);
}
headWriter.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0
      Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
      transitional.dtd\">");
headWriter.write("<html xmlns=\"http://www.w3.org/1999/xhtml\">");
headWriter.write("<body bgcolor=\"#dde3f1\">");
headWriter.write("<form action=\"\">");
headWriter.close();

// write beginning contents of the body frame
try {
      bodyWriter = new PrintWriter(new BufferedWriter(new
      FileWriter(fileNameBody, true)));
} catch (Exception ex) {
      System.out.println(ex);
}
bodyWriter.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0
      Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
      transitional.dtd\">");
bodyWriter.write("<html xmlns=\"http://www.w3.org/1999/xhtml\">");
bodyWriter.write("<body bgcolor=\"#dde3f1\">");
bodyWriter.close();
```

The footer code follows the same model. With the header and footer of the HTML pages in place, the listeners can add the guts of the HTML pages during traversal of the parse tree.

5.3.2.3 *Matching Up Properties-Strings*

A challenge to matching up the graphical controls with their associated strings is that the declaration of properties and strings are not at the same level in the parse tree. For example, the specification of the name of a checkbox is done through the property `prompt`. The string associated with `prompt` is neither a parent, nor a child, nor a peer of `prompt`; it is a *child of a peer* of `prompt`:

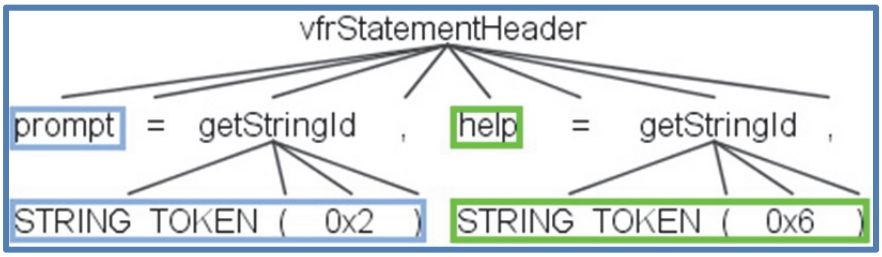

Figure 13: Incongruity Between Properties and Their Strings

This fact precludes the listener methods from simply taking the string value from within their own context. To resolve the problem, the HTML application uses two stacks—one for the properties and one for the strings of those properties. As the parse tree is traversed, property names, e.g., `prompt`, `help`, get pushed on the properties stack, and strings get pushed on the strings stack.

```
// stack of properties
Deque<String> propertyStack = new ArrayDeque<String>();

// stack of strings that go with each property
Deque<String> stringTokenStack = new ArrayDeque<String>();
```

```
/* upon entering a VFR statement, we can ascertain whether or not
 * "prompt" and/or "help" properties exist.  So, look for them
 * here and push them on the stack.
 */
public void enterVfrStatementHeader(vfrParser.VfrStatementHeaderContext
      ctx) {
      inVfrStatementHeader = true;  // ensure we don't push/pop outside
      checkboxes
      for (int i=0; i<ctx.getChildCount(); i++) {
            if (ctx.getChild(i).getText().equals("help"))
                  propertyStack.push("help");
            if (ctx.getChild(i).getText().equals("prompt"))
                  propertyStack.push("prompt");
      }
}

/* leaving GetStringId we know the value of the string,
so push it onto the stack. */
if (inVfrStatementHeader == true) {
      String stringId = ctx.getText();
      stringId = stringId.replace("STRING_TOKEN", "");
      stringId = stringId.replace("(", "");
      stringId = stringId.replace(")", "");
      stringTokenStack.push(strDatabase.get(stringId));
}
```

Later, when these strings need to be resolved and associated with the correct property in order to complete a checkbox control, a pop is made from each stack and the correct association is maintained in a HashMap:

```
Map<String, String> checkboxProperties = new HashMap<String, String>();

// populate the HashMap with the properties discovered during tree
      traversal
while (!propertyStack.isEmpty() && !stringTokenStack.isEmpty())
      checkboxProperties.put(propertyStack.pop(),
      stringTokenStack.pop());
```

Now, the correct string can be written out to HTML:

```
writer.write("<input type=\"checkbox\" checked=\"checked\" name=\"vfr\"
      value=\"value\">" +
            checkboxProperties.get("prompt"));
```

5.3.3 Results

The following VFR code was used as input to the application:

```
form formid = 0x8d71,
    title = STRING_TOKEN(0x1);

    suppressif ideqval WirelessSupportedWWAN == 0,
    grayoutif ideqval SetupAccess == 0,
    checkbox varid  = WirelessEnableWWAN,
        prompt   = STRING_TOKEN(0x3),
        help     = STRING_TOKEN(0x7),
        default value = 1,
    endcheckbox;
    endif;
    endif;

    grayoutif ideqval SetupAccess == 0,
    checkbox varid  = WirelessEnableWLAN,
        prompt   = STRING_TOKEN(0x4),
        help     = STRING_TOKEN(0x8),
        default value = 1,
    endcheckbox;
    endif;

    grayoutif ideqval SetupAccess == 0,
    checkbox varid  = WirelessEnableBT,
        prompt   = STRING_TOKEN(0x5),
        help     = STRING_TOKEN(0x9),
        default value = 1,
    endcheckbox;
    endif;
endform;
```

This code creates a form called Wireless Device Enable with three checkboxes. Each checkbox can enable or disable a particular type of wireless radio. This code rendered on a real, live system from a popular computer manufacturer looks like:

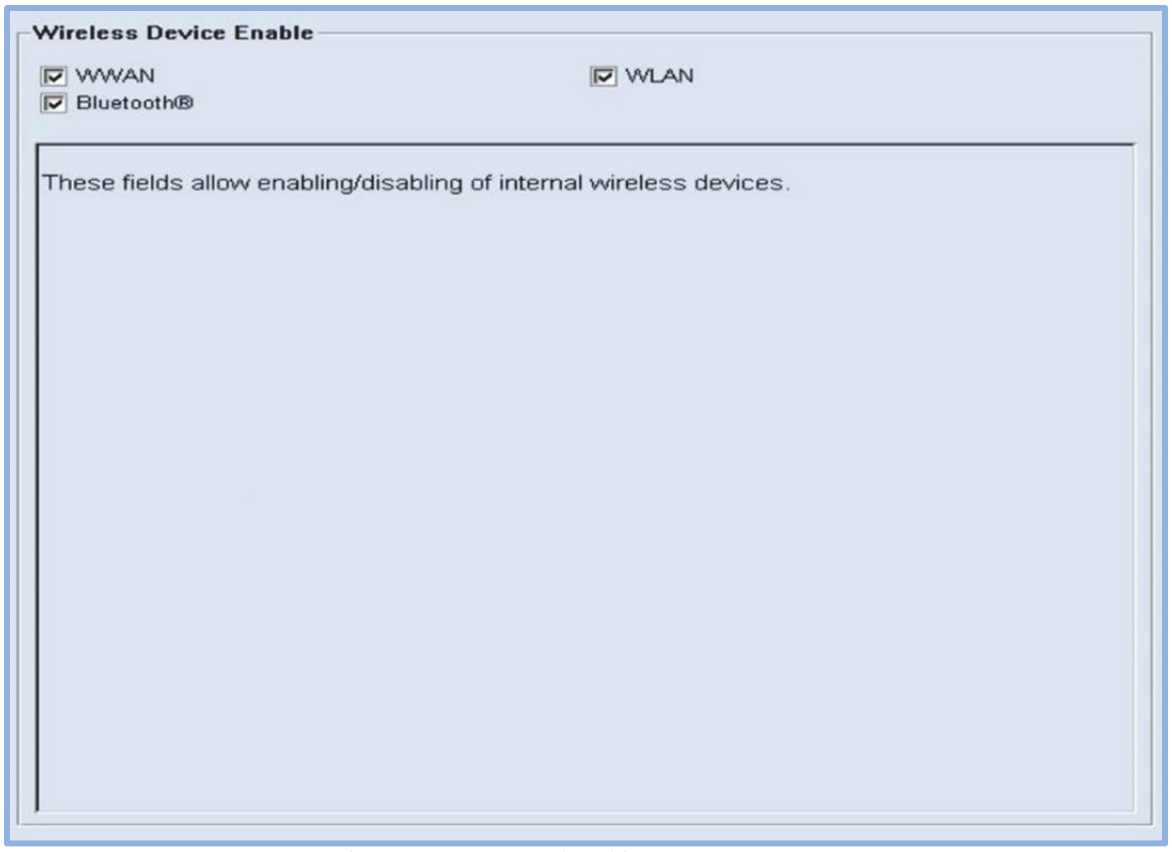

Figure 14: VFR Rendered in UEFI Computer

The same VFR code (with improved help text) output from the HTML preview application:[6] [7]

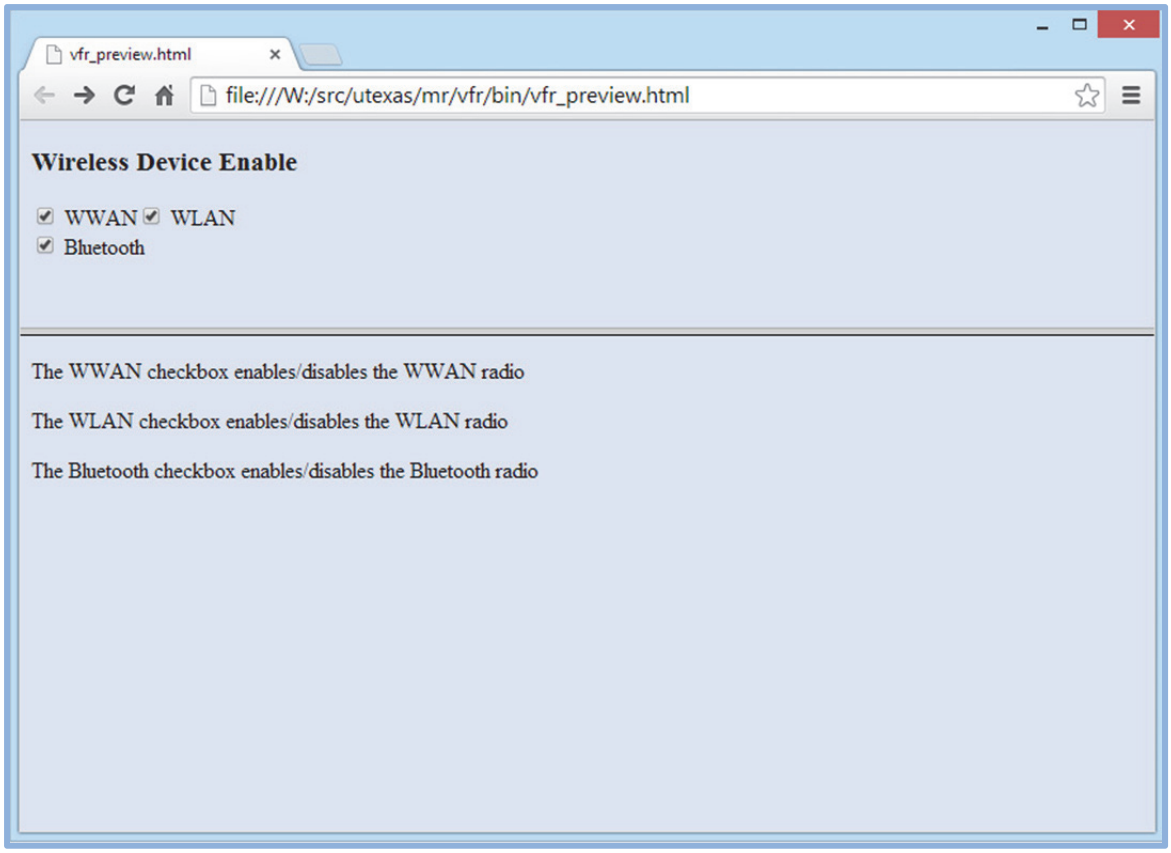

Figure 15: VFR Rendered in HTML Browser

[6] The Java class for the application is listed in Appendix E

[7] The HTML auto-generated by the application is listed in Appendix F

Chapter 6: Conclusion

This report has demonstrated how compiler theory, in particular the use of the ANTLR parser generator, can be used to solve problems in UEFI BIOS programming. After some background into the BIOS and compiler theory, the report presented three example language applications for UEFI BIOS.

6.1 Contributions

6.1.1 Module Complexity

The report produced a Java application for gauging the complexity of UEFI images using both general-purpose and domain-specific complexity measures. The application's algorithm was designed around a set of vectors of complexity that influence the risk and maintainability of UEFI images. Comparisons were made between the scores generated by the application and the number of releases per UEFI image and number of changesets per UEFI image. The application did well, matching the benchmark measure in twenty to twenty-one out of thirty-two popular UEFI images.

6.1.2 Visual Forms Representation (VFR)

An ANTLR grammar was created to represent the VFR DSL using Intel's BNF-like language specification. A sample application was created in order to give the user instant preview, negating the need for a lengthy VFR compile, build, and flash cycle. The application instantiated the ANTLR grammar and then translated VFR code to HTML which can be viewed by any web browser.

6.2 Related Work

UEFI BIOS is not a popular area of research in academia. Research and development in this space comes from:

- Intel
- Microsoft
- Independent BIOS vendors (IBVs)
    - Phoenix Technologies, AMI, Insyde Software
- The computer hardware companies
    - Dell, H-P, Apple, NVidia, AMD

SourceForge hosts the UEFI development kits, tutorials, and some open source implementation of parts of UEFI.[8] A good source for research into UEFI comes from the blog posts of people who work in the field.[9] [10] [11] [12]

6.3 Future Work

UEFI BIOS is becoming the firmware foundation for all computing devices today, from multi-socketed enterprise servers to smart phones. Yet, there is little recognition of this technology in the industry. Performing a search of IEEE's digital library, there are only eight papers in some way touching on UEFI BIOS, out of a total of 3.6 million available documents. Below are ways the ideas presented in this paper can be developed further.

[8] https://sourceforge.net/apps/mediawiki/tianocore/index.php
[9] http://vzimmer.blogspot.com/
[10] http://uefi.blogspot.com/
[11] https://twitter.com/Intel_UEFI
[12] https://twitter.com/uefibios

6.3.1 Module Complexity

Using the initial weights, the algorithm produced good results, but this algorithm could be improved by the application of statistical methods. A good approach would be to re-run the analysis over and over while varying the weights. From this database of statistical information, data mining tools could run a regression analysis and determine the contribution each of the vectors of complexity has on the UEFI images. This would make the algorithm more precise, and at the same time possibly result in the reduction of some of the attributes.

6.3.2 Visual Forms Representation

The VFR HTML preview application was only a small proof-of-concept demonstration of what can be done with a VFR grammar. Also possible are HTML preview support for VFR's C-style preprocessor, sets of multiple forms, and various controls other than checkboxes.

Besides the HTML preview idea, there is a need for Forms Browsers that display VFR in environments beyond the local console. For example, an IT administrator accessing the content from a smart phone app for purposes of remote administration. A company might want to give users the ability to change firmware values from within the operating system, rather than accessing them at the pre-OS phase. With the same VFR grammar, more Forms Browsers could be created running on more types of environments, both local and remote.

Finally, the sample application took VFR code and produced an HTML representation. Performing the reverse operation is equally compelling: creating a graphical layout application that saves its layout as well-formed VFR ready for compilation into a BIOS image. For example, an application could allow the user

to create text, checkboxes, radio buttons, and place them on a grid. The application could save the graphical representation as legitimate VFR code, thus eliminating the need to write the VFR by hand.

Appendix A. Complexity Java Application

```
package com.williamleara.utexas.mastersreport;

import org.antlr.v4.runtime.ANTLRInputStream;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
import org.antlr.v4.runtime.ParserRuleContext;

import java.io.FileInputStream;
import java.io.InputStream;
import java.util.*;

public class Complex {

    // data structure for parsing results
    public static Map<String, Integer> vectors_of_complexity = new
HashMap<String, Integer>();

    public static class ComplexityAnalyzer extends UefiBaseListener {

        // weights for each vector of complexity
        public static final int NUMSTATEMENTS_WEIGHT = 1;
        public static final int PROTOCOLS_WEIGHT = 1;
        public static final int VARIABLESERVICES_WEIGHT = 1;
        public static final int CALLBACKS_WEIGHT = 1;
        public static final int BLOCKIO_WEIGHT = 1;
        public static final int MALLOC_WEIGHT = 1;

        // data structures for each vector
        List<String> protocolHandlers = new ArrayList<String>();
        List<String> variableServices = new ArrayList<String>();
        List<String> callbacks = new ArrayList<String>();
        List<String> blockIo= new ArrayList<String>();
        List<String> memAlloc= new ArrayList<String>();

        public ComplexityAnalyzer() {
            // initialize data structure
            vectors_of_complexity.put("NumStatements", 0);
            vectors_of_complexity.put("Protocols", 0);
            vectors_of_complexity.put("VariableServices", 0);
            vectors_of_complexity.put("Callbacks", 0);
            vectors_of_complexity.put("BlockIo", 0);
            vectors_of_complexity.put("MemAlloc", 0);

            // initialize ArrayList:  protocol handlers
```

```
                protocolHandlers.add("InstallProtocolInterface");
                protocolHandlers.add("UninstallProtocolInterface");
                protocolHandlers.add("ReinstallProtocolInterface");
                protocolHandlers.add("RegisterProtocolNotify");
                protocolHandlers.add("LocateHandle");
                protocolHandlers.add("HandleProtocol");
                protocolHandlers.add("LocateDevicePath");
                protocolHandlers.add("OpenProtocol");
                protocolHandlers.add("CloseProtocol");
                protocolHandlers.add("OpenProtocolInformation");
                protocolHandlers.add("ConnectController");
                protocolHandlers.add("DisconnectController");
                protocolHandlers.add("ProtocolsPerHandle");
                protocolHandlers.add("LocateHandleBuffer");
                protocolHandlers.add("LocateProtocol");
                protocolHandlers.add("InstallMultipleProtocolInterfaces");

protocolHandlers.add("UninstallMultipleProtocolInterfaces");

                // initialize ArrayList:  variable services
                variableServices.add("GetVariable");
                variableServices.add("GetNextVariableName");
                variableServices.add("SetVariable");
                variableServices.add("QueryVariableInfo");

                // initialize ArrayList:  callbacks
                callbacks.add("CreateEvent");
                callbacks.add("CreateEventEx");
                callbacks.add("CloseEvent");
                callbacks.add("SignalEvent");
                callbacks.add("WaitForEvent");
                callbacks.add("CheckEvent");
                callbacks.add("SetTimer");
                callbacks.add("RaiseTPL");
                callbacks.add("RestoreTPL");

                // initialize ArrayList:  block io devices
                blockIo.add("EFI_BLOCK_IO_PROTOCOL");
                blockIo.add("EFI_BLOCK_IO2_PROTOCOL");

                // initialize ArrayList:  memory allocation
                memAlloc.add("AllocatePages");
                memAlloc.add("FreePages");
                memAlloc.add("GetMemoryMap");
                memAlloc.add("AllocatePool");
                memAlloc.add("FreePool");
```

```
        }

        // listener for number of statements
        public void enterStatement(UefiParser.StatementContext ctx) {
            vectors_of_complexity.put("NumStatements",
vectors_of_complexity.get("NumStatements") + 1*NUMSTATEMENTS_WEIGHT);
        }

        // listener for expressions
        public void enterTypedefName(UefiParser.TypedefNameContext ctx)
{

        // protocol handler services
        if (protocolHandlers.contains(ctx.getText()))
            vectors_of_complexity.put("Protocols",
vectors_of_complexity.get("Protocols") + 1*PROTOCOLS_WEIGHT);
        // variable services
        else if (variableServices.contains(ctx.getText()))
            vectors_of_complexity.put("VariableServices",
vectors_of_complexity.get("VariableServices")                              +
1*VARIABLESERVICES_WEIGHT);
        // event and timer callbacks
        else if (callbacks.contains(ctx.getText()))
            vectors_of_complexity.put("Callbacks",
vectors_of_complexity.get("Callbacks") + 1*CALLBACKS_WEIGHT);
        // block io
        else if (blockIo.contains(ctx.getText()))
            vectors_of_complexity.put("BlockIo",
vectors_of_complexity.get("BlockIo") + 1*BLOCKIO_WEIGHT);
        // memory allocation
        else if (memAlloc.contains(ctx.getText()))
            vectors_of_complexity.put("MemAlloc",
vectors_of_complexity.get("MemAlloc") + 1*MALLOC_WEIGHT);
        }
    }

    public static void main(String[] args) throws Exception {

        // check for input file
        String inputFile = null;
        if (args.length <= 0) {
            System.out.println("ERROR: missing input file");
            System.exit(1);
        }
```

```
        // main program loop
        ComplexityAnalyzer cmplx = new ComplexityAnalyzer();
        for (int i=0; i<args.length; i++) {
            inputFile = args[i];
            InputStream is = System.in;
            if ( inputFile != null )
                is = new FileInputStream(inputFile);

            // create lexer and parser
            UefiLexer lexer = new UefiLexer(new ANTLRInputStream(is));
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            UefiParser parser = new UefiParser(tokens);
            parser.setBuildParseTree(true);
            ParserRuleContext tree = parser.compilationUnit();
            ParseTreeWalker walker = new ParseTreeWalker();

            // perform analysis
            walker.walk(cmplx, tree);
        }

        // print results
        System.out.println("Component Scores:");
        System.out.println("NumStatements:              "              +
vectors_of_complexity.get("NumStatements"));
        System.out.println("Protocols:                "                +
vectors_of_complexity.get("Protocols"));
        System.out.println("VariableServices:             "            +
vectors_of_complexity.get("VariableServices"));
        System.out.println("Callbacks:                "                +
vectors_of_complexity.get("Callbacks"));
        System.out.println("BlockIo:                 "                 +
vectors_of_complexity.get("BlockIo"));
        System.out.println("MemAlloc:                "                 +
vectors_of_complexity.get("MemAlloc"));

        System.out.print("\nComposite Score: ");
        System.out.println(
                vectors_of_complexity.get("NumStatements") +
                vectors_of_complexity.get("Protocols") +
                vectors_of_complexity.get("VariableServices") +
                vectors_of_complexity.get("Callbacks") +
                vectors_of_complexity.get("BlockIo") +
                vectors_of_complexity.get("MemAlloc")
        );
    }
}
```

Appendix B. UEFI Image Release Frequency

The following is the complete listing of all ninety-three UEFI images and how often they were released, sorted in descending order.

UEFI image	number of releases
UEFI image1	51
UEFI image2	21
UEFI image3	19
UEFI image4	17
UEFI image5	16
UEFI image6	15
UEFI image7	15
UEFI image8	14
UEFI image9	14
UEFI image10	12
UEFI image11	11
UEFI image12	11
UEFI image13	10
UEFI image14	10
UEFI image15	10
UEFI image16	9
UEFI image17	9
UEFI image18	9
UEFI image19	9
UEFI image20	9
UEFI image21	9
UEFI image22	8
UEFI image23	8
UEFI image24	7
UEFI image25	7
UEFI image26	7
UEFI image27	7
UEFI image28	7
UEFI image29	6
UEFI image30	5
UEFI image31	5
UEFI image32	4
UEFI image33	4
UEFI image34	4
UEFI image35	4
UEFI image36	4
UEFI image37	3
UEFI image38	3
UEFI image39	3
UEFI image40	3
UEFI image41	3
UEFI image42	3
UEFI image43	3
UEFI image44	3
UEFI image45	3
UEFI image46	3
UEFI image47	2
UEFI image48	2
UEFI image49	2
UEFI image50	2
UEFI image51	2
UEFI image52	2
UEFI image53	2
UEFI image54	2
UEFI image55	2
UEFI image56	2
UEFI image57	2
UEFI image58	2
UEFI image59	2
UEFI image60	2
UEFI image61	2
UEFI image62	2

UEFI image63	2
UEFI image64	1
UEFI image65	1
UEFI image66	1
UEFI image67	1
UEFI image68	1
UEFI image69	1
UEFI image70	1
UEFI image71	1
UEFI image72	1
UEFI image73	1
UEFI image74	1
UEFI image75	1
UEFI image76	1
UEFI image77	1
UEFI image78	1
UEFI image79	1

UEFI image80	1
UEFI image81	1
UEFI image82	1
UEFI image83	1
UEFI image84	1
UEFI image85	1
UEFI image86	1
UEFI image87	1
UEFI image88	1
UEFI image89	1
UEFI image90	1
UEFI image91	1
UEFI image92	1
UEFI image93	1

Appendix C. Changes to UEFI Images

The following is the complete listing of all ninety-three UEFI images and how often they were changes; i.e., number of committed changesets, sorted in descending order.

UEFI Image	Number of Changesets
UEFI image1	125
UEFI image4	63
UEFI image3	49
UEFI image6	49
UEFI image2	48
UEFI image10	47
UEFI image9	43
UEFI image17	41
UEFI image8	39
UEFI image5	39
UEFI image7	37
UEFI image23	32
UEFI image16	28
UEFI image14	27
UEFI image12	27
UEFI image15	27
UEFI image29	26
UEFI image21	25
UEFI image11	24
UEFI image20	24
UEFI image19	23
UEFI image13	22
UEFI image24	22
UEFI image27	22
UEFI image35	22
UEFI image41	21
UEFI image22	20
UEFI image25	20
UEFI image31	20
UEFI image42	19
UEFI image43	19
UEFI image47	18
UEFI image32	18
UEFI image30	18
UEFI image28	17
UEFI image40	16
UEFI image67	15
UEFI image34	15
UEFI image36	15
UEFI image33	14
UEFI image51	14
UEFI image53	14
UEFI image45	14
UEFI image62	14
UEFI image66	13
UEFI image48	13
UEFI image39	13
UEFI image18	13
UEFI image26	13
UEFI image44	13
UEFI image63	13
UEFI image37	12
UEFI image49	12
UEFI image56	12
UEFI image57	12
UEFI image46	12
UEFI image59	12
UEFI image92	12
UEFI image93	12
UEFI image65	11

UEFI image38	11
UEFI image77	11
UEFI image52	11
UEFI image64	10
UEFI image68	10
UEFI image75	10
UEFI image79	10
UEFI image80	10
UEFI image81	10
UEFI image84	10
UEFI image70	9
UEFI image72	9
UEFI image50	9
UEFI image76	9
UEFI image82	9
UEFI image54	9
UEFI image86	9
UEFI image87	9

UEFI image90	9
UEFI image91	9
UEFI image73	8
UEFI image74	8
UEFI image78	8
UEFI image83	8
UEFI image55	8
UEFI image88	8
UEFI image85	7
UEFI image89	7
UEFI image60	6
UEFI image69	5
UEFI image71	5
UEFI image61	5
UEFI image58	4

Appendix D. ANTLR Grammar for VFR

```
grammar vfr;

vfrFormDefinition
    :     Form  FormId  Equals  Number  Comma  Title  Equals  getStringId
SemiColon
    (
    vfrStatementImage
        | vfrStatementLocked
        | vfrStatementRules
        | vfrStatementDefault
        | vfrStatementStat
        | vfrStatementQuestions
        | vfrStatementConditional
        | vfrStatementLabel
        | vfrStatementBanner
        | vfrStatementExtension
        | vfrStatementModal
    )*
    EndForm SemiColon
    ;

vfrStatementStat
    : vfrStatementSubTitle
    | vfrStatementStaticText
    | vfrStatementCrossReference
    ;

vfrStatementCrossReference
    : vfrStatementGoto
    | vfrStatementResetButton
    ;

vfrStatementGoto
    : Goto
    (
        (
            DevicePath Equals getStringId Comma
            FormsetGuid Equals guidDefinition Comma
            FormId Equals Number Comma
            Question Equals Number Comma
        )
        |
        (
            FormsetGuid Equals guidDefinition Comma
```

```
            FormId Equals Number Comma
            Question Equals Number Comma
        )
        |
        (
            FormId Equals Number Comma
            Question Equals
            (
                StringIdentifier Comma
                | Number Comma
            )
        )
        |
        (
            Number Comma
        )
    )?
    vfrQuestionHeader
    ( Comma Flags Equals vfrGotoFlags )?
    ( Comma Key Equals Number )?
    ( Comma vfrStatementQuestionOptionList )?
    SemiColon
;

vfrGotoFlags
    : gotoFlagsField ( OrBitwise gotoFlagsField )*
    ;

gotoFlagsField
    : Number
    | questionheaderFlagsField
    ;

vfrStatementResetButton
    : ResetButton
    DefaultStore Equals StringIdentifier Comma
    vfrStatementHeader Comma
    ( vfrStatementStatTagList Comma )?
    EndResetButton SemiColon
    ;

vfrStatementStatTagList
    : vfrStatementStatTag ( Comma vfrStatementStatTag )*
    ;

vfrStatementStaticText
```

```
    : Text
    Help Equals getStringId Comma
    Text Equals getStringId
    ( Comma Text Equals getStringId )?
    (
    Comma    Flags    Equals    staticTextFlagsField    (    OrBitwise
staticTextFlagsField )*
    Comma Key Equals Number
    )?
    ( Comma vfrStatementStatTagList )? SemiColon
    ;

staticTextFlagsField
    : Number
    | questionheaderFlagsField
    ;

vfrStatementSubTitle
    : Subtitle
    Text Equals getStringId
    ( Comma Flags Equals vfrSubtitleFlags )?
    ( Comma vfrStatementStatTagList )? SemiColon
    ;

vfrSubtitleFlags
    : subtitleFlagsField ( OrBitwise subtitleFlagsField )* SemiColon
    ;

subtitleFlagsField : Number | Horizontal ;

getStringId : StringToken LParen Number RParen ;

vfrStatementImage
    : vfrImageTag SemiColon
    ;

vfrImageTag
    : Image Equals ImageToken LParen Number RParen
    ;

vfrStatementLocked
    : vfrLockedTag SemiColon
    ;

vfrLockedTag : Locked ;
```

```
vfrStatementRules
    :  Rule  StringIdentifier  Comma  vfrStatementExpression  EndRule
SemiColon
    ;


vfrStatementModal : Modal SemiColon ;

vfrStatementExtension
    : Guidop
    Guid Equals guidDefinition
    (
    Comma Datatype Equals
    (
    UINT64 ( LBracket Number RBracket )?
    | UINT32 ( LBracket Number RBracket )?
    | UINT16 ( LBracket Number RBracket )?
    | UINT8 ( LBracket Number RBracket )?
    | BOOLEAN ( LBracket Number RBracket )?
    | EfiStringId ( LBracket Number RBracket )?
    | EfiHiiDate ( LBracket Number RBracket )?
    | EfiHiiTime ( LBracket Number RBracket )?
    | EfiHiiRef ( LBracket Number RBracket )?
    | StringIdentifier ( LBracket Number RBracket )?
    )
    vfrExtensionData
    )?
    (
    Comma ( vfrStatementExtension )*
    EndGuidop
    )?
    SemiColon
    ;

vfrExtensionData
    :
    (
        Comma Data ( LBracket Number RBracket )?
        (
            Period StringIdentifier ( LBracket Number RBracket )?
        )*
        Equals Number
    )*
    ;

vfrStatementBanner
```

```
    : Banner ( Comma )?
    Title Equals getStringId Comma
    (
        (
            Line Number Comma
            Align ( Left | Center | Right ) SemiColon
        )
        |
        ( Timeout Equals Number SemiColon )
    )
    ;

vfrStatementConditional
    : vfrStatementDisableIfStat
    | vfrStatementSuppressIfStat
    | vfrStatementGrayOutIfStat
    ;

vfrStatementDisableIfStat
    : DisableIf vfrStatementExpression Comma
    ( vfrStatementStatList )*
    EndIf SemiColon
    ;

vfrStatementSuppressIfStat
    : SuppressIf vfrStatementExpression Comma
    ( vfrStatementStatList )*
    EndIf SemiColon
    ;

vfrStatementGrayOutIfStat
    : GrayoutIf vfrStatementExpression Comma
    ( vfrStatementStatList )*
    EndIf SemiColon
    ;

vfrStatementStatList
    : vfrStatementStat
    | vfrStatementQuestions
    | vfrStatementConditional
    | vfrStatementLabel
    | vfrStatementExtension
    ;

vfrStatementLabel : Label Number SemiColon ;
```

```
vfrStatementStatTag : vfrImageTag | vfrLockedTag ;

vfrStatementQuestions
    : vfrStatementBooleanType
    | vfrStatementDate
    | vfrStatementNumericType
    | vfrStatementStringType
    | vfrStatementOrderedList
    | vfrStatementTime
    ;

vfrStatementTime :
    Time
    (
        (
        vfrQuestionHeader Comma
        ( Flags Equals vfrTimeFlags Comma )?
        vfrStatementQuestionOptionList
        )
    |
    (
        Hour  Varid  Equals  StringIdentifier  Period  StringIdentifier
Comma
        Prompt Equals getStringId Comma
        Help Equals getStringId Comma
        minMaxTimeStepDefault
        Minute  Varid  Equals  StringIdentifier  Period  StringIdentifier
Comma
        Prompt Equals getStringId Comma
        Help Equals getStringId Comma
        minMaxTimeStepDefault
        Second  Varid  Equals  StringIdentifier  Period  StringIdentifier
Comma
        Prompt Equals getStringId Comma
        Help Equals getStringId Comma
        minMaxTimeStepDefault
        ( vfrStatementInconsistentIf )*
    )
    )
    Endtime SemiColon
    ;

minMaxTimeStepDefault
    : Minimum Equals Number Comma
    Maximum Equals Number Comma
```

```
    ( Step Equals Number Comma )?
    ( Default Equals Number Comma )?
    ;

vfrTimeFlags
    : timeFlagsField ( OrBitwise timeFlagsField )*
    ;

timeFlagsField
    : Number
    | HourSuppress
    | MinuteSuppress
    | SecondSuppress
    | StorageNormal
    | StorageTime
    | StorageWakeup
    ;

vfrStatementOrderedList
    : Orderedlist
    vfrQuestionHeader Comma
    ( Maxcontainers Equals Number Comma)?
    ( Flags Equals vfrOrderedListFlags )?
    vfrStatementQuestionOptionList
    Endlist SemiColon
    ;

vfrOrderedListFlags
    : orderedlistFlagsField ( OrBitwise orderedlistFlagsField )*
    ;

orderedlistFlagsField
    :
    ( Number
    | Unique
    | Noempty
    )
    | questionheaderFlagsField
    ;

vfrStatementStringType
    : vfrStatementString
    | vfrStatementPassword
    ;

vfrStatementPassword
```

```
    : Password
    vfrQuestionHeader
    ( Flags Equals vfrPasswordFlagsField Comma )?
    ( Key Equals Number Comma )?
    MinSize Equals Number Comma
    MaxSize Equals Number Comma
    vfrStatementQuestionOptionList
    Endpassword SemiColon
    ;

vfrPasswordFlagsField     :     passwordFlagsField     (     OrBitwise
passwordFlagsField )* ;

passwordFlagsField
    : Number
    | questionheaderFlagsField
    ;

vfrStatementString
    : String
    vfrQuestionHeader Comma
    ( Flags Equals vfrStringFlagsField Comma )?
    ( Key Equals Number Comma )?
    MinSize Equals Number Comma
    MaxSize Equals Number Comma
    vfrStatementQuestionOptionList
    EndString SemiColon
    ;

vfrStringFlagsField : stringFlagsField ( OrBitwise stringFlagsField )*
;

stringFlagsField
    :
    ( Number
    | MultiLine
    )
    | questionheaderFlagsField
    ;

vfrStatementNumericType
    : vfrStatementNumeric
    | vfrStatementOneOf
    ;

vfrStatementOneOf
```

```
    : Oneof
    vfrQuestionHeader
    ( Flags Equals vfrOneofFlagsField Comma )?
    ( vfrSetMinMaxStep )
    vfrStatementQuestionOptionList
    EndOneof SemiColon
    ;

vfrOneofFlagsField
    : numericFlagsField ( OrBitwise numericFlagsField )*
    ;

vfrStatementNumeric
    : Numeric
    vfrQuestionHeader
    ( Flags Equals vfrNumericFlags Comma )?
    ( Key Equals Number Comma )?
    vfrSetMinMaxStep
    vfrStatementQuestionOptionList
    Endnumeric SemiColon
    ;

vfrSetMinMaxStep
: Minimum Equals Number Comma
Maximum Equals Number Comma
( Step Equals Number Comma )?
;

vfrNumericFlags
    : numericFlagsField ( OrBitwise numericFlagsField )*
    ;

numericFlagsField
    :
    ( Number
    | NumSize1
    | NumSize2
    | NumSize4
    | NumSize8
    | DispIntDec
    | DispUintDec
    | DispUintHex
    )
    | questionheaderFlagsField
    ;
```

```
vfrStatementDate
    : Date
    (
        (
        vfrQuestionHeader
        ( Flags Equals vfrDateFlags Comma )?
        vfrStatementQuestionOptionList
        )
    |
        (
        Year Varid Equals StringIdentifier Period StringIdentifier
Comma
        Prompt Equals getStringId Comma
        Help Equals getStringId Comma
        minMaxDateStepDefault
        Month Varid Equals StringIdentifier Period StringIdentifier
Comma
        Prompt Equals getStringId Comma
        Help Equals getStringId Comma
        minMaxDateStepDefault
        Day Varid equalTerm StringIdentifier Period StringIdentifier
Comma
        Prompt Equals getStringId Comma
        Help Equals getStringId Comma
        minMaxDateStepDefault
        ( vfrStatementInconsistentIf )*
        )
    )
    EndDate SemiColon
    ;

vfrStatementInconsistentIf
    : InconsistentIf
    Prompt Equals getStringId Comma
    vfrStatementExpression
    EndIf
    ;

vfrStatementQuestionOptionList
    : (
    vfrStatementQuestionTag
    | vfrStatementQuestionOptionTag
    )*
    ;

vfrStatementQuestionOptionTag
```

```
    : vfrStatementSuppressIfQuest
    | vfrStatementValue
    | vfrStatementDefault
    | vfrStatementOptions
    | vfrStatementRead
    | vfrStatementWrite
    ;

vfrStatementOptions : vfrStatementOneOfOption ;

vfrStatementOneOfOption
    : Option
    Text Equals getStringId Comma
    Value Equals vfrConstantValueField Comma
    Flags Equals vfrOneOfOptionFlags
    ( Comma vfrImageTag )*
    SemiColon
    ;

vfrOneOfOptionFlags     :     oneofoptionFlagsField     (     OrBitwise
oneofoptionFlagsField )* ;

oneofoptionFlagsField
    : Number
    | OptionDefault
    | OptionDefaultMfg
    | Interactive
    | ResetRequired
    | Default
    ;

vfrStatementRead : Read vfrStatementExpression SemiColon ;
vfrStatementWrite : Write vfrStatementExpression SemiColon ;

vfrStatementSuppressIfQuest
    : SuppressIf vfrStatementExpression SemiColon
    vfrStatementQuestionOptionList
    EndIf
    ;

vfrStatementQuestionTag
    : vfrStatementStatTag Comma
    | vfrStatementInconsistentIf
    | vfrStatementNoSubmitIf
    | vfrStatementDisableIfQuest | vfrStatementRefresh
    | vfrStatementVarstoreDevice
```

```
    | vfrStatementExtension
    | vfrStatementRefreshEvent
    ;

vfrStatementVarstoreDevice : VarStoreDevice Equals getStringId Comma ;

vfrStatementRefreshEvent : RefreshGuid Equals guidDefinition Comma ;

vfrStatementRefresh
    : Refresh Interval Equals Number
    ;

vfrStatementDisableIfQuest
    : DisableIf  vfrStatementExpression SemiColon
    vfrStatementQuestionOptionList
    EndIf
    ;

vfrStatementNoSubmitIf
    : NoSubmitIf
    Prompt Equals getStringId Comma
    vfrStatementExpression
    EndIf
    ;

vfrDateFlags : dateFlagsField ( OrBitwise dateFlagsField )* ;

dateFlagsField
    : Number
    | YearSuppress
    | MonthSuppress
    | DaySuppress
    | StorageNormal
    | StorageTime
    | StorageWakeup
    ;

vfrQuestionHeader
    : ( Name Equals StringIdentifier Comma )?
    ( Varid Equals vfrStorageVarId Comma )?
    ( QuestionId Equals Number Comma )?
    vfrStatementHeader
    ;

vfrStatementHeader
    : Prompt Equals getStringId Comma
```

```
    Help Equals getStringId Comma
    ;

minMaxDateStepDefault
    : Minimum Equals Number Comma
    Maximum Equals Number Comma
    ( Step Equals Number Comma )?
    ( Default Equals Number Comma)?
    ;

vfrStatementBooleanType
    : vfrStatementCheckBox
    | vfrStatementAction
    ;

vfrStatementCheckBox
    : Checkbox
    vfrQuestionHeader
    ( Flags Equals vfrCheckBoxFlags Comma )?
    ( Key Equals Number Comma )?
    vfrStatementQuestionOptionList
    EndCheckbox SemiColon
    ;

vfrCheckBoxFlags
    : checkboxFlagsField ( OrBitwise checkboxFlagsField )*
    ;

checkboxFlagsField
    :
    (
    Number
    | CheckboxDefault
    | CheckboxDefaultMfg
    )
    | questionheaderFlagsField
    ;

vfrStatementAction
    : Action
    vfrQuestionHeader Comma
    ( Flags Equals vfrActionFlags Comma )?
    Config Equals getStringId Comma
    vfrStatementQuestionTagList
    EndAction SemiColon
    ;
```

```
vfrStatementQuestionTagList : ( vfrStatementQuestionTag )* ;

vfrActionFlags
: actionFlagsField ( OrBitwise actionFlagsField )*
;

actionFlagsField
: Number
| questionheaderFlagsField
;

vfrStatementDefault
    : Default
    (
        (vfrStatementValue Comma | Equals vfrConstantValueField Comma)
        ( DefaultStore Equals StringIdentifier Comma )?
    )
    ;

vfrConstantValueField
    :
    ( Number
    | TRUE
    | FALSE
    | ONE
    | ONES
    | ZERO
    )
    | Number Colon Number Colon Number
    | Number Divide Number Divide Number
    | StringToken LParen Number RParen
    ;

vfrStatementValue
    : Value Equals vfrStatementExpression
    ;

vfrStatementExpression
    : andTerm (Or andTerm)*
    ;

andTerm
    : bitwiseorTerm (And bitwiseorTerm)*
    ;
```

```
bitwiseorTerm
    : bitwiseandTerm ( OrBitwise bitwiseandTerm )*
    ;

bitwiseandTerm
    : equalTerm ( AndBitwise equalTerm )*
    ;

equalTerm
    : compareTerm (EqualsEquals compareTerm | NotEquals compareTerm)*
    ;

compareTerm
    : shiftTerm
    (
    LessThan shiftTerm
    | LessThanEquals shiftTerm
    | GreaterThan shiftTerm
    | GreaterThanEquals shiftTerm
    )*
    ;

shiftTerm
    : addMinusTerm
    (
    ShiftLeft addMinusTerm
    | ShiftRight addMinusTerm
    )*
    ;

addMinusTerm
    : multdivmodTerm
    (
    Plus multdivmodTerm
    | Minus multdivmodTerm
    )*
    ;

multdivmodTerm
    : castTerm
    (
    Multiply castTerm
    | Divide castTerm
    | Modulus castTerm
    )*
    ;
```

```
castTerm
    : (
    LParen
    (
    BOOLEAN
    | UINT64
    | UINT32
    | UINT16
    | UINT8
    )
    RParen
    )*
    atomTerm
    ;

atomTerm
    : vfrExpressionCatenate
    | vfrExpressionMatch
    | vfrExpressionParen
    | vfrExpressionBuildInFunction
    | vfrExpressionConstant
    | vfrExpressionUnaryOp
    | vfrExpressionTernaryOp
    | vfrExpressionMap
    | ( Not atomTerm )
    ;

vfrExpressionCatenate
    :      Catenate      LParen      vfrStatementExpression      Comma
vfrStatementExpression RParen
    ;

vfrExpressionMatch
    : Match
    LParen vfrStatementExpression Comma vfrStatementExpression RParen
    ;

vfrExpressionParen
    : LParen vfrStatementExpression RParen
    ;

vfrExpressionBuildInFunction
    : dupExp
    | ideqvalExp
    | ideqidExp
```

```
    | ideqvallistExp
    | questionref1Exp
    | rulerefExp
    | stringref1Exp
    | pushthisExp
    | securityExp
    | getExp
    ;

dupExp : Dup ;

vfrQuestionDataFieldName : StringIdentifier;

ideqvalExp
    : Ideqval
    vfrQuestionDataFieldName EqualsEquals Number
    ;

ideqidExp
    : Ideqid
    vfrQuestionDataFieldName EqualsEquals vfrQuestionDataFieldName
    ;

ideqvallistExp
    : Ideqvallist
    vfrQuestionDataFieldName EqualsEquals ( Number )+
    ;

questionref1Exp
    : Questionref
    LParen StringIdentifier | Number RParen
    ;

rulerefExp
    : Ruleref LParen StringIdentifier RParen
    ;

stringref1Exp
    : Stringref LParen getStringId RParen
    ;

pushthisExp : Pushthis ;

securityExp : Security LParen guidDefinition RParen ;

getExp
```

```
    :   Get   LParen   vfrStorageVarId   OrBitwise?   Flags?   Equals?
vfrNumericFlags? RParen
    ;

questionheaderFlagsField
    : ReadOnly
    | Interactive
    | ResetRequired
    | OptionsOnly
    ;

vfrExpressionConstant
    : TRUE
    | FALSE
    | ONE
    | ONES
    | ZERO
    | UNDEFINED
    | VERSION
    | Number
    ;

vfrExpressionUnaryOp
    : lengthExp
    | bitwisenotExp
    | question23refExp
    | stringref2Exp
    | toboolExp
    | tostringExp
    | unintExp
    | toupperExp
    | tolwerExp
    | setExp
    ;

lengthExp
    : Length LParen vfrStatementExpression RParen
    ;

bitwisenotExp
    : NotBitwise LParen vfrStatementExpression RParen
    ;

question23refExp
    : Questionrefval
    LParen
```

```
    (
    DevicePath Equals StringToken BackSlash LParen S Number BackSlash
RParen Comma
    )?
    (
    Uuid Equals guidDefinition Comma
    )?
    vfrStatementExpression
    RParen
    ;

stringref2Exp
    : Stringrefval LParen vfrStatementExpression RParen
    ;

toboolExp
    : Boolval LParen vfrStatementExpression RParen
    ;

tostringExp
    : Stringval Format? Equals? Number? Comma?
    LParen vfrStatementExpression RParen
    ;

unintExp
    : Unintval LParen vfrStatementExpression RParen
    ;

toupperExp
    : Toupper LParen vfrStatementExpression RParen
    ;

tolwerExp
    : Tolower LParen vfrStatementExpression RParen
    ;

setExp
    : Set
    LParen
    vfrStorageVarId OrBitwise? Flags? Equals? vfrNumericFlags? Comma
    vfrStatementExpression
    RParen
    ;

vfrStorageVarId
    : ( StringIdentifier LBracket Number RBracket )
```

```
    |
    (
    StringIdentifier
        (
        Period StringIdentifier ( LBracket Number RBracket )?
        )*
    )
    ;

vfrExpressionTernaryOp
    : conditionalExp
    | findExp
    | midExp
    | tokenExp
    | spanExp
    ;

conditionalExp
    : Cond
    LParen
    vfrStatementExpression
    QuestionMark
    vfrStatementExpression
    Colon
    vfrStatementExpression
    RParen
    ;

findExp
    : Find
    LParen
    findFormat ( OrBitwise findFormat )*
    Comma
    vfrStatementExpression
    Comma
    vfrStatementExpression
    Comma
    vfrStatementExpression
    RParen
    ;

findFormat
    : Sensitive
    | Insensitive
    ;
```

```
midExp
    : Mid
    LParen
    vfrStatementExpression
    Comma
    vfrStatementExpression
    Comma
    vfrStatementExpression
    RParen
    ;

tokenExp
    : Token
    LParen
    vfrStatementExpression
    Comma
    vfrStatementExpression
    Comma
    vfrStatementExpression
    RParen
    ;

spanExp
    : Span
    LParen
    Flags Equals spanFlags ( OrBitwise spanFlags )*
    Comma
    vfrStatementExpression
    Comma
    vfrStatementExpression
    Comma
    vfrStatementExpression
    RParen
    ;

spanFlags
    : Number
    | LastNonMatch
    | FirstNonMatch
    ;

vfrExpressionMap
    : Map
    LParen
    vfrStatementExpression
    Colon
```

```
    (
    vfrStatementExpression
    Comma
    vfrStatementExpression
    Comma
    )*
    RParen
    ;

guidSubDefinition
    : Hex2 Comma Hex2 Comma Hex2 Comma Hex2 Comma Hex2 Comma
    Hex2 Comma Hex2 Comma Hex2
    ;

guidDefinition
    : LBrace
    Hex8 Comma Hex4 Comma Hex4 Comma
    (
    LBrace guidSubDefinition RBrace
    | guidSubDefinition
    )
    RBrace
    ;

// lexer rules
Form : 'form' ;
EndForm : 'endform' ;
FormId : 'formid' ;
Title : 'title' ;
Rule : 'rule' ;
EndRule : 'endrule' ;
Catenate : 'catenate' ;
Match : 'match' ;
Dup : 'dup' ;
Ideqval : 'ideqval' ;
Ideqid : 'ideqid' ;
Ideqvallist : 'ideqvallist' ;
Questionref : 'questionref' ;
Ruleref : 'ruleref' ;
Stringref : 'stringref' ;
Pushthis : 'pushthis' ;
Security : 'security' ;
Get : 'get' ;
Flags : 'flags' ;
Length : 'length' ;
Questionrefval : 'questionrefval' ;
```

```
StringToken : 'STRING_TOKEN' ;
Stringrefval : 'stringrefval' ;
Boolval : 'boolval' ;
Stringval : 'stringval' ;
Unintval : 'unintval' ;
Toupper : 'toupper' ;
Tolower : 'tolower' ;
Set : 'set' ;
Cond : 'cond' ;
Find : 'find' ;
Sensitive : 'SENSITIVE' ;
Insensitive : 'INSENSITIVE' ;
Mid : 'mid' ;
Token : 'token' ;
Span : 'span' ;
Map : 'map' ;
FirstNonMatch : 'FIRST_NON_MATCH' ;
LastNonMatch : 'LAST_NON_MATCH' ;
Image : 'image' ;
ImageToken : 'IMAGE_TOKEN' ;
Locked : 'locked' ;
Format : 'format' ;
Uuid : 'Uuid' ;
DevicePath : 'DevicePath' ;
S : 'S:' ;
Default : 'default' ;
DefaultStore : 'defaultstore' | 'defaultStore' ;
Value : 'value' ;
Year : 'year' ;
Help : 'help' ;
Month : 'month' ;
Prompt : 'prompt' ;
Varid : 'varid' ;
EndDate : 'enddate' ;
Minimum : 'minimum' ;
Maximum : 'maximum' ;
Step : 'step' ;
Date : 'date' ;
Name : 'name' ;
QuestionId : 'questionid' ;
EndIf : 'endif' ;
InconsistentIf : 'inconsistentif' ;
SuppressIf : 'suppressif' ;
Read : 'read' ;
Write : 'write' ;
Option : 'option' ;
```

```
Text : 'text' ;
NoSubmitIf : 'nosubmitif' ;
DisableIf : 'disableif' ;
Refresh : 'refresh' ;
Interval : 'interval' ;
VarStoreDevice : 'varstoredevice' ;
RefreshGuid : 'refreshguid' ;
Checkbox : 'checkbox' ;
EndCheckbox : 'endcheckbox' ;
Key : 'key' ;
Action : 'action' ;
EndAction : 'endaction' ;
Config : 'config' ;
Subtitle : 'subtitle' ;
Horizontal : 'HORIZONTAL' ;
ResetButton : 'resetbutton' ;
EndResetButton : 'endresetbutton' ;
Goto : 'goto' ;
FormsetGuid : 'formsetguid' ;
Question : 'question' ;
Numeric : 'numeric' ;
Endnumeric : 'endnumeric' ;
Oneof : 'oneof' ;
EndOneof : 'endoneof' ;
Day : 'day' ;
Hour : 'hour' ;
Minute : 'minute' ;
Second : 'second' ;
Time : 'time' ;
Endtime : 'endtime' ;
MultiLine : 'MULTI_LINE' ;
String : 'string' ;
EndString : 'endstring' ;
MinSize : 'minsize' ;
MaxSize : 'maxsize' ;
Password : 'password' ;
Endpassword : 'endpassword' ;
Orderedlist : 'orderedlist' ;
Endlist : 'endlist' ;
Maxcontainers : 'maxcontainers' ;
Unique : 'UNIQUE' ;
Noempty : 'NOEMPTY' ;
GrayoutIf : 'grayoutif' ;
Label : 'label' ;
Banner : 'banner' ;
Line : 'line' ;
```

```
Align : 'align' ;
Left : 'left' ;
Right : 'right' ;
Center : 'center' ;
Timeout : 'timeout' ;
Guidop : 'guidop' ;
EndGuidop : 'endguidop' ;
Guid : 'guid' ;
Data : 'data' ;
Datatype : 'datatype' ;
Modal : 'modal' ;

// expressions
Or : 'OR' ;
And : 'AND' ;
Not : 'NOT' ;

// punctuation
Equals : '=' ;
EqualsEquals : '==' ;
NotEquals : '!=' ;
Comma : ',' ;
SemiColon : ';' ;
OrBitwise : '|' ;
AndBitwise : '&' ;
NotBitwise : '~' ;
LessThan : '<' ;
LessThanEquals : '<=' ;
GreaterThan : '>' ;
GreaterThanEquals : '>=' ;
ShiftLeft : '<<' ;
ShiftRight : '>>' ;
Plus : '+' ;
Minus : '-' ;
Multiply : '*' ;
Divide : '/' ;
Modulus : '%' ;
LParen : '(' ;
RParen : ')' ;
LBrace : '{' ;
RBrace : '}' ;
QuestionMark : '?' ;
Colon : ':' ;
Period : '.' ;
LBracket : '[' ;
RBracket : ']' ;
```

```
BackSlash : '\\' ;

// types
BOOLEAN : 'BOOLEAN' ;
UINT64 : 'UINT64' ;
UINT32 : 'UINT32' ;
UINT16 : 'UINT16' ;
UINT8 : 'UINT8' ;

// constants
TRUE : 'TRUE' ;
FALSE : 'FALSE' ;
ONE : 'ONE' ;
ONES : 'ONES' ;
ZERO : 'ZERO' ;
UNDEFINED : 'UNDEFINED' ;
VERSION : 'VERSION' ;

// number and string
Number
    : ('0x'[0-9A-Fa-f]+) | [0-9]+
    ;
StringIdentifier
    : [A-Za-z_] [A-Za-z_0-9]*
    ;
Hex8 : '0x'[0-9A-fa-f] ;
Hex4 : '0x'[0-9A-fa-f] ;
Hex2 : '0x'[0-9A-fa-f] ;

// flags
NumSize1 : 'NUMERIC_SIZE_1' ;
NumSize2 : 'NUMERIC_SIZE_2' ;
NumSize4 : 'NUMERIC_SIZE_4' ;
NumSize8 : 'NUMERIC_SIZE_8' ;
DispIntDec : 'DISPLAY_INT_DEC' ;
DispUintDec : 'DISPLAY_UINT_DEC' ;
DispUintHex : 'DISPLAY_UINT_HEX' ;
ReadOnly : 'READ_ONLY' ;
Interactive : 'INTERACTIVE' ;
ResetRequired : 'RESET_REQUIRED' ;
OptionsOnly : 'OPTIONS_ONLY' ;
YearSuppress : 'YEAR_SUPPRESS' ;
MonthSuppress : 'MONTH_SUPPRESS' ;
DaySuppress : 'DAY_SUPPRESS' ;
StorageNormal : 'STORAGE_NORMAL' ;
StorageTime : 'STORAGE_TIME' ;
```

```
StorageWakeup : 'STORAGE_WAKEUP' ;
OptionDefault : 'OPTION_DEFAULT' ;
OptionDefaultMfg : 'OPTION_DEFAULT_MFG' ;
CheckboxDefault : 'CHECKBOX_DEFAULT' ;
CheckboxDefaultMfg : 'CHECKBOX_DEFAULT_MFG' ;
HourSuppress : 'HOUR_SUPPRESS' ;
MinuteSuppress : 'MINUTE_SUPPRESS' ;
SecondSuppress : 'SECOND_SUPPRESS' ;
EfiStringId : 'EFI_STRING_ID' ;
EfiHiiDate : 'EFI_HII_DATE' ;
EfiHiiTime : 'EFI_HII_TIME' ;
EfiHiiRef : 'EFI_HII_REF' ;

// whitespace
Whitespace
    :   [ \t]+
        -> skip
    ;

Newline
    :   (   '\r' '\n'?
        |   '\n'
        )
        -> skip
    ;

BlockComment
    :   '/*' .*? '*/'
        -> skip
    ;

LineComment
    :   '//' ~[\r\n]*
        -> skip
    ;
```

Appendix E. VFR Application

```
package com.williamleara.utexas.mastersreport;

import org.antlr.v4.runtime.ANTLRInputStream;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
import org.antlr.v4.runtime.ParserRuleContext;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.InputStream;
import java.io.PrintWriter;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.StringTokenizer;

public class Vfr {

    public static class VfrToHTML extends vfrBaseListener {

        // name of the files that will hold the VFR preview
        static String fileNameMain = "vfr_preview.html";
        static String fileNameHead = "head.html";
        static String fileNameBody = "body.html";

        // set of strings associated with the VFR
        static String strFile = "strings.txt";

        // database of identifier to string mappings
        Map<String, String> strDatabase = new HashMap<String,
String>();

        // stack of properties
        Deque<String> propertyStack = new ArrayDeque<String>();

        // stack of strings that go with each property
        Deque<String> stringTokenStack = new ArrayDeque<String>();
```

```
        boolean inVfrStatementHeader = false;
        boolean gotTitle = false;
        int columnCount = 0;

        public VfrToHTML() {

            // remove output files from any previous runs
            List<File> filesToDelete = new ArrayList<File>();
            filesToDelete.add(new File(fileNameMain));
            filesToDelete.add(new File(fileNameHead));
            filesToDelete.add(new File(fileNameBody));

            for (File f : filesToDelete) {
                if (f.exists())
                  f.delete();
            }

            // populate string values from strings file
            File f = new File(strFile);
            BufferedReader reader = null;

            try {
                reader = new BufferedReader(new FileReader(f));
                String stringMapping = null;
                while ((stringMapping = reader.readLine()) != null) {
                    StringTokenizer          st          =          new
StringTokenizer(stringMapping, ",");
                    String key = st.nextToken();
                    String value = st.nextToken();
                    strDatabase.put(key, value);
                }
            } catch (Exception ex) {
                System.out.println(ex);
            }
        }

        // create header upon entering the form
        public                                                  void
enterVfrFormDefinition(vfrParser.VfrFormDefinitionContext ctx) {
            emitHeader();
        }

        // create footer as we leave the form
        public                                                  void
exitVfrFormDefinition(vfrParser.VfrFormDefinitionContext ctx) {
```

```
            emitFooter();
        }

        /* upon entering a VFR statement, we can ascertain whether or
not
         * "prompt" and/or "help" properties exist.  So, look for them
         * here and push them on the stack.
         */
        public                                                  void
enterVfrStatementHeader(vfrParser.VfrStatementHeaderContext ctx) {
            inVfrStatementHeader = true;   // ensure we don't push/pop
outside checkboxes
            for (int i=0; i<ctx.getChildCount(); i++) {
                if (ctx.getChild(i).getText().equals("help"))
                    propertyStack.push("help");
                if (ctx.getChild(i).getText().equals("prompt"))
                    propertyStack.push("prompt");
            }
        }

        public                                                  void
exitVfrStatementHeader(vfrParser.VfrStatementHeaderContext ctx) {
            // ensure we don't push/pop outside checkboxes
            inVfrStatementHeader = false;
        }

        /* leaving GetStringId we know the value of the string,
        so push it onto the stack. */
        public void exitGetStringId(vfrParser.GetStringIdContext ctx) {

            if (!gotTitle) {
                PrintWriter writer = null;
                try {
                    writer  =  new  PrintWriter(new  BufferedWriter(new
FileWriter(fileNameHead, true)));
                } catch (Exception ex) {
                    System.out.println(ex);
                }
                String title = ctx.getText();
                title = title.replace("STRING_TOKEN", "");
                title = title.replace("(", "");
                title = title.replace(")", "");
                writer.write("<h3>"    +    strDatabase.get(title)    +
"</h3>");
                writer.close();
                gotTitle = true;
```

```
            }

            if (inVfrStatementHeader == true) {
                String stringId = ctx.getText();
                stringId = stringId.replace("STRING_TOKEN", "");
                stringId = stringId.replace("(", "");
                stringId = stringId.replace(")", "");
                stringTokenStack.push(strDatabase.get(stringId));
            }
        }

        public                                                  void
exitVfrStatementCheckBox(vfrParser.VfrStatementCheckBoxContext ctx) {
            Map<String,     String>     checkboxProperties     =     new
HashMap<String, String>();
            PrintWriter writer = null;

            // populate the HashMap with the properties discovered
during tree traversal
            while              (!propertyStack.isEmpty()              &&
!stringTokenStack.isEmpty())
                checkboxProperties.put(propertyStack.pop(),
stringTokenStack.pop());

            // update the header part of the HTML page
            try {
                writer   =   new   PrintWriter(new   BufferedWriter(new
FileWriter(fileNameHead, true)));
            } catch (Exception ex) {
                System.out.println(ex);
            }

            // use % to write the checkboxes in two columns
            if ((columnCount % 2) == 1)
                writer.write("<input                    type=\"checkbox\"
checked=\"checked\" name=\"vfr\" value=\"value\">" +
                        checkboxProperties.get("prompt") + "<br>");
            else
                writer.write("<input                    type=\"checkbox\"
checked=\"checked\" name=\"vfr\" value=\"value\">" +
                        checkboxProperties.get("prompt"));
            writer.close();
            columnCount++;

            // update the body part of the HTML page
            try {
```

```
                writer = new PrintWriter(new BufferedWriter(new
FileWriter(fileNameBody, true)));
            } catch (Exception ex) {
                System.out.println(ex);
            }
            writer.write("<p>" + checkboxProperties.get("help") +
"</p>");
            writer.close();
        }

        void emitHeader() {
            PrintWriter mainWriter = null;
            PrintWriter headWriter = null;
            PrintWriter bodyWriter = null;

            // write beginning contents of the main frameset
            try {
                mainWriter = new PrintWriter(new BufferedWriter(new
FileWriter(fileNameMain, true)));
            } catch (Exception ex) {
                System.out.println(ex);
            }
            mainWriter.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML
1.0 Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd\">");
            mainWriter.write("<html
xmlns=\"http://www.w3.org/1999/xhtml\">");
            mainWriter.write("<head>");
            mainWriter.write("</head>");
            mainWriter.write("<frameset rows=\"150,*\">");
            mainWriter.write("<frame src=\"head.html\"
scrolling=\"no\">");
            mainWriter.write("<frame src=\"body.html\"
scrolling=\"no\">");
            mainWriter.close();

            // write beginning contents of the header frame
            try {
                headWriter = new PrintWriter(new BufferedWriter(new
FileWriter(fileNameHead, true)));
            } catch (Exception ex) {
                System.out.println(ex);
            }
            headWriter.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML
1.0 Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd\">");
```

```
            headWriter.write("<html
xmlns=\"http://www.w3.org/1999/xhtml\">");
            headWriter.write("<body bgcolor=\"#dde3f1\">");
            headWriter.write("<form action=\"\">");
            headWriter.close();

            // write beginning contents of the body frame
            try {
                bodyWriter = new PrintWriter(new BufferedWriter(new
FileWriter(fileNameBody, true)));
            } catch (Exception ex) {
                System.out.println(ex);
            }
            bodyWriter.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML
1.0    Transitional//EN\"    \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd\">");
            bodyWriter.write("<html
xmlns=\"http://www.w3.org/1999/xhtml\">");
            bodyWriter.write("<body bgcolor=\"#dde3f1\">");
            bodyWriter.close();
        }

        void emitFooter() {
            PrintWriter mainWriter = null;
            PrintWriter headWriter = null;
            PrintWriter bodyWriter = null;

            // write final contents of the body frame
            try {
                bodyWriter = new PrintWriter(new BufferedWriter(new
FileWriter(fileNameBody, true)));
            } catch (Exception ex) {
                System.out.println(ex);
            }
            bodyWriter.write("</body>");
            bodyWriter.write("</html>");
            bodyWriter.close();

            // write final contents of the header frame
            try {
                headWriter = new PrintWriter(new BufferedWriter(new
FileWriter(fileNameHead, true)));
            } catch (Exception ex) {
                System.out.println(ex);
            }
            headWriter.write("</form>");
```

```
            headWriter.write("</body>");
            headWriter.write("</html>");
            headWriter.close();

            // write final contents of the main frameset
            try {
                mainWriter = new PrintWriter(new BufferedWriter(new
FileWriter(fileNameMain, true)));
            } catch (Exception ex) {
                System.out.println(ex);
            }
            mainWriter.write("</frameset>");
            mainWriter.write("</html>");
            mainWriter.close();
        }
    }

    public static void main(String[] args) throws Exception {

        // check for input file
        if (args.length <= 0) {
            System.out.println("ERROR: missing input file");
            System.exit(1);
        }

        // create file input stream
        String inputFile = null;
        inputFile = args[0];
        InputStream is = System.in;
        if ( inputFile != null )
            is = new FileInputStream(inputFile);

        // create lexer and parser
        vfrLexer lexer = new vfrLexer(new ANTLRInputStream(is));
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        vfrParser parser = new vfrParser(tokens);
        parser.setBuildParseTree(true);
        ParserRuleContext tree = parser.vfrFormDefinition();
        ParseTreeWalker walker = new ParseTreeWalker();

        // perform analysis
        VfrToHTML vfrHTML = new VfrToHTML();
        walker.walk(vfrHTML, tree);
    }
}
```

Appendix F. Auto-Generated HTML

vfr_preview.html:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html
xmlns="http://www.w3.org/1999/xhtml"><head></head><frameset
rows="150,*"><frame src="head.html" scrolling="no"><frame
src="body.html" scrolling="no"></frameset></html>
```

head.html:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html
xmlns="http://www.w3.org/1999/xhtml"><body bgcolor="#dde3f1"><form
action=""><h3> Wireless Device Enable</h3><input type="checkbox"
checked="checked" name="vfr" value="value"> WWAN<input type="checkbox"
checked="checked" name="vfr" value="value"> WLAN<br><input
type="checkbox" checked="checked" name="vfr" value="value">
Bluetooth</form></body></html>
```

body.html:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html
xmlns="http://www.w3.org/1999/xhtml"><body bgcolor="#dde3f1"><p> The
WWAN checkbox enables/disables the WWAN radio</p><p> The WLAN checkbox
enables/disables the WLAN radio</p><p> The Bluetooth checkbox
enables/disables the Bluetooth radio</p></body></html>
```

Glossary

ACPI – Advanced Control and Power Interface
ANTLR – ANother Tool for Language Recognition
BDS – Boot Device Selection phase
BIOS – Basic Input/Output System
BNF – Backus-Naur Form
CMOS – Complementary Metal Oxide Semiconductor
CP/M – Control Program/Monitor
DASH – Desktop and Mobile Architecture for System Hardware
DOS – Disk Operating System
DP – DisplayPort
DXE – Driver Execution Environment
EDK – EFI Development Kit
EFI – Extensible Firmware Interface
IBV – Independent BIOS Vendor
IDE – Integrated Development Environment
IFR – Internal Forms Representation
LED – Light Emitting Diode
MSR – Model Specific Register
NEM – No-Eviction Mode
NOR – NOT OR, a type of flash memory used by SPI parts
NVRAM – Non-Volatile Random Access Memory
PCI – Peripheral Component Interconnect
PEI – Pre-Extensible Firmware Interface Initialization
POST – Power On Self-Test
RAM – Random Access Memory
ROM – Read Only Memory
RT – Run-Time
SATA – Serial AT Attachment
SCSI – Small Computer System Interface
SEC – Security, the first phase of UEFI
SMBIOS – System Management BIOS
SMBus – System Management Bus
SPI – Serial Peripheral Interface
UEFI – Unified Extensible Firmware Interface
USB – Universal Serial Bus
VFR – Visual Forms Representation

Bibliography

[1] Phoenix Technologies, System BIOS for IBM PCs, Compatibles, and EISA Computers, 2nd Ed., Reading: Addison-Wesley, 1991.

[2] T. Parr, The Definitive ANTLR4 Reference, Dallas: The Pragmatic Programmers, 2012.

[3] T. Shanley and D. Anderson, ISA System Architecture, Reading: Addison-Wesley, 1995.

[4] W. L. Rosch, The Winn Rosch Hardware Bible, 3rd Edition, Indianapolis: Sams Publishing, 1994.

[5] P. Dice, "Booting an Intel Architecture System," Dr. Dobbs Journal, 26 December 2011. [Online]. Available: http://www.drdobbs.com/parallel/booting-an-intel-architecture-system-par/232300699. [Accessed April 2014].

[6] Wikipedia, "Pentium FDIV Bug," [Online]. Available: https://en.wikipedia.org/wiki/Pentium_FDIV_bug. [Accessed April 2014].

[7] P. Norton, The Peter Norton Programmer's Guide to the IBM PC, Redmond: Microsoft Press, 1985.

[8] Phoenix Technologies, System BIOS for IBM PC/XT/AT Computers and Compatibles, Reading: Addison-Wesley, 1989.

[9] Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba, "Advanced Configuration and Power Interface Specification 5.0," www.acpi.info, 2011.

[10] DMTF, "SMBIOS," [Online]. Available: http://www.dmtf.org/standards/smbios. [Accessed April 2014].

[11] DMTF, "DASH," [Online]. Available: http://www.dmtf.org/standards/dash. [Accessed April 2014].

[12] UEFI, Inc., "Unified Extensible Firmware Interface Specification 2.3.1," www.uefi.org, 2012.

[13] V. Zimmer, M. Rothman and R. Hale, Beyond BIOS: Developing with the Unified Extensible Firmware Interface 2nd Edition, Intel Press, 2010.

[14] Wikipedia, "Execute in place," [Online]. Available: https://en.wikipedia.org/wiki/Execute_in_place. [Accessed April 2014].

[15] J. A. Farrell, "Compiler Basics," August 1995. [Online]. Available: http://www.cs.man.ac.uk/~pjj/farrell/compmain.html. [Accessed April 2014].

[16] T. Parr, Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages, Dallas: The Pragmatic Programmers, 2010.

[17] J. Levine, flex & bison, Cambridge: O'Reilly, 2009.

[18] A. Aziz, "Lexical Analysis Overview," EE382V Advanced Programming Tools, 2011.

[19] S. Harwell, "C.g4," https://github.com/antlr/grammars-v4/tree/master/c, 2013.

[20] Wikipedia, "Backus-Naur Form," [Online]. Available: https://en.wikipedia.org/wiki/Backus-Naur_form. [Accessed April 2014].

[21] A. Aziz, "Parsing Implementation," EE382V Advanced Programming Tools, 2011.

[22] A. Aziz, "Parsing Overview," EE382V Advanced Programming Tools, 2011.

[23] Wikipedia, "Syntax Analysis," [Online]. Available: https://en.wikipedia.org/wiki/Syntax_analysis. [Accessed April 2014].

[24] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Reading: Addison-Wesley, 1995.

[25] Wikipedia, "Halstead complexity measures," [Online]. Available: https://en.wikipedia.org/wiki/Halstead_complexity_measures. [Accessed April 2014].

[26] Intel Corporation, "Initializing the Boot Process," http://www.tianocore.org/.

[27] Wikipedia, "Lexical Analysis," [Online]. Available: https://en.wikipedia.org/wiki/Lexical_analysis. [Accessed April 2014].

[28] Wikipedia, "Lexical Grammar," [Online]. Available: https://en.wikipedia.org/wiki/Lexical_grammar. [Accessed April 2014].

[29] M. Fowler, Domain-Specific Languages, Reading: Addison-Wesley, 2011.

[30] J. Jex, "Flash Memory BIOS for PC and Notebook Computers," in *IEEE Conference Publications*, 1991.

[31] V. Bashun, A. Sergeev, V. Minchenkov and A. Yakovlev, "Too young to be secure: Analysis of UEFI threats and vulnerabilities," in *Open Innovations Association (FRUCT), 2013 14th Conference of*, 2013.

[32] L. Dailey Paulson, "New Technology Beefs Up BIOS," *Computer,* vol. 37, no. 5, 2004.

[33] Intel Corporation, "VFR Programming Language, Version 1.7," http://www.uefi.org, 2012.

[34] International Business Machines, Technical Reference, Personal Computer AT, International Business Machines, 1985.

Vita

William Daniel Leara graduated from Boston University with a Bachelor of Science in 1993. Since then, he has worked for Dell Inc. in a variety of product development roles, including Test Engineering, Software Development, and most recently BIOS Development. He is currently working on developing UEFI BIOS firmware for Dell's latest generation of desktop and portable computer systems. In the autumn of 2011, he entered the Graduate School at the University of Texas at Austin and enrolled in the School of Engineering's Software Engineering Master's Degree program.

Permanent email: williamleara@gmail.com

This report was typed by the author, William Daniel Leara.