SRL – A Simple Retargetable Loader*

David Ung Cristina Cifuentes
Centre for Software Maintenance
School of Information Technology
University of Queensland
{davidu,cristina}@it.uq.edu.au

Abstract

A loader is a systems program used by an operating system (OS) to load a binary executable file onto memory to execute it. The internal format of a binary executable file is called the binary-file format (BFF); this format is dependent on the OS and the particular computer architecture it runs on.

Traditionally, when developing machine-code manipulation tools such as binary translators and disassemblers, developers need to write a decoder for each type of binary executable file they want to manipulate, i.e. for n different binary executables, they need to write n different loaders. With the advent of binary translation technology and the increased number of machines and operating systems, a retargetable loader (RL) would eliminate the effort required in creating different loaders; if only one such environment existed.

SRL, a simple retargetable loader, is a first attempt at developing an RL framework by means of a simple BFF grammar. Three different environments, (x86,DOS,EXE), (x86,Windows,NE) and (Sparc,Solaris,ELF), were used as the basis for the development and testing of SRL. The three environments give a good coverage of different BFFs currently in use by OSs for RISC and CISC machines.

1. Introduction

Machine-code manipulation tools such as binary translators, disassemblers, decompilers, binary debuggers, and tracers or profilers, are systems programming tools that interact with the computer's operating system (OS) to either translate low-level machine instructions into a higher level of abstraction (e.g. in the case of disassemblers, decompilers and debuggers), or to another low-level of abstraction (e.g. in the case of binary translators). The first step in developing any machine-code manipulation tool is to understand the source binary-file format (BFF) of the executable program[1]; i.e. the type of information it stores, how it is stored, and how it can be accessed or retrieved. Users of computer programs are normally not concerned with any of these issues since the OS automatically handles the *loading* of the program into memory.

When a program is to be executed in memory, the OS first extracts all necessary information from the executable file's header and carries out any necessary actions (e.g. relocation) before putting it into memory. In other words, the OS decodes the executable file into an understandable representation in memory and passes control to the program for its execution. This process is often described as the *loading* of a program.

The general object code decoding abstraction for the purposes of creating machine-code manipulation tools is shown in Figure 1, where we describe the environment of a binary executable by means of a triplet (M,OS,BFF). M denotes a machine or computer architecture, OS denotes the operating system used by that machine, and BFF denotes the binary-file format used by that operating system. In Figure 1, a source object file (M1,OS1,BFF1) is fed into a loader which decodes file-related information. The program's text is then decoded based on M1's machine instruction set onto an intermediate representation that is suitable for the particular machine-code manipulation tool. For example, a binary translator will need to translate the intermediate representation onto another target object code program for a different environment (M2,OS2,BFF2); a disassembler will need to produce an assembly code program; and a decompiler will need to produce a high-level language program. Each of these tools will require a different level of abstraction in their intermediate representation.

*This work is partially supported by Sun Microsystems Laboratories and the Australian Research Council under grant No. A49702762.

[1]Different names are used in the literature to refer to executable programs: binary programs, object code, binary executables, or executable programs. All of these names are used as synonyms in this paper, and refer to the end product of the compilation and linkage process.

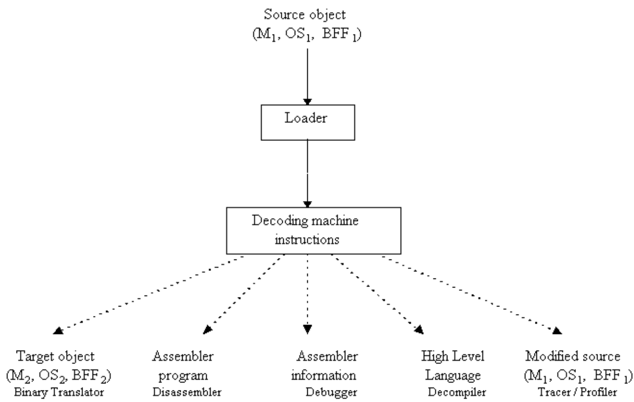

Figure 1. Object code decoding abstraction

Binary translators are of particular importance in this study as they interact with BFFs twice; once during the decoding of the source eecutable program, and once during the encoding of the new target executable program. Existing binary translators such as Digital's VEST and mx [13], Freeport Express [5] and FX!32 [15]; Sun's Wabi [14]; Apple's MAE [4] and AT&T's Flashport [1] support one or two different platforms only. We are developing a retargetable binary translation framework that supports a larger set of platforms.

1.1. Retargetable loading

Traditionally, when developing a machine-code manipulation tool, developers need to write a decoder for every BFF they want to manipulate. For example, if we want to write a disassembler for an Intel x86 machine running DOS and using the EXE binary file format [6, 12], we write a loader for the EXE format and a decoder of machine instructions for (x86,DOS). If we then decide to write another disassembler for the Windows New Executable (NE) BFF [6, 12], we need to write another loader for NE and a modified machine instruction decoder for (x86,Windows) as the interface to information on the BFF is different. So, if we have n different (M,OS,BFF) tuples, we will need to write n different loaders.

A systems programmer developing a machine-code manipulation tool and wanting to test the new tool on A different machine architectures, using B different operating systems and C different BFFs, will need to write AxBxC different loaders. However, the process carried out by all loaders is similar despite the fact that the (M,OS,BFF) tuples are different. Ideally, developers would like to write as minimum code as possible in order to cater for different (M,OS,BFF) tuples. This approach is possible with the development of reusable components that are automatically generated from specifications; i.e. a retargetable loader (RL) framework.

The input to the RL framework is a BFF specification and a binary executable. The BFF specification is an unambiguous description of a binary-file format. The output of the RL is a high-level language interface for the loading of the program.

The rest of this paper is structured in the following way. Section 2 discusses previous works related to the design of a retargetable loader. Section 3 describes the development of an RL using specifications; structure of binary-file formats (BFFs) are discussed in Section 4. The BFF properties and the grammar used by the Simple Retargetable Loader (SRL) are the main topic of Section 5. Section 6 describes how we tested the SRL and the results accomplished. A summary and conclusion follow the paper.

2. Previous work

A few methods and tools are currently available for capturing binary information stored in executable programs. Overall, there are 3 different approaches that can be used in developing a loader (or any other system tool) [16]:

1. hand-craft the code,
2. use library routines to assist in the writing of the code, or
3. use specifications for automatic generation of the code.

The first approach is the easiest and quickest to implement, although sometimes tedious to test. This simplicity advantage is only good for creating loaders that are limited to the knowledge of one BFF. As described earlier, one would need to hand craft n different loaders for n different (M,OS,BFF) system tuples, and hence this approach is unsuitable for retargetability purposes.

Approaches 2 and 3 can be time-consuming to implement at first (i.e. developing the library or the generator of code based on specifications), however, once this time investment has been made, the production of other loaders is quick and simple. An overhead in learning the tool at hand will always be needed though. Either approach provides support for the creation of an RL framework. A widely used example of the second approach is provided by the Binary-File Descriptor (BFD) library [2], which uses an extensive structure to represent details within a binary file; routines for new BFFs are added to the library incrementally. An example of the third approach was attempted in the DWG (AutoCAD) work [7], which uses a specification language to describe the contents of the file. Both these methods are reviewed in the coming subsections.

With the recent trend towards Java and portable software, the use of machine-independent representations to store the final "executable" program is bound to grow. Java "executable" programs are binary representations of bytecodes

for an abstract stack machine. This machine is implemented by the Java Virtual Machine (VM) [11]. Java "executables" also contain other information required by the VM for execution of the program.

The creation of a library interface for loading such bytecode programs, or the specification of such binary format, are possible. However, the loading of the program as such will be governed by rules in the VM rather than traditional Von Neumann machine rules (i.e. execute machine instructions sequentially from the given entry point, following the flow of control of the program). In the work reported in this paper, we have concentrated on the traditional, in use, CISC and RISC machines.

2.1. Binary-file descriptor library

GNU's Binary-File Descriptor (BFD) Library [2] is a package containing common routines that applications can use regardless of their underlying binary-file format. The BFD library divides each specified BFF into the front-end and the back-end. The front-end interfaces between the user and the BFD, while the back-end provides a set of calls which the BFD front-end can use to decode and manage the object file. To support a new BFF, the programmer needs to create a new BFD back-end and add it to the library.

BFD has its own binary representation for internal processing known as the canonical object file format. When an object file (M,OS,BFF) is opened, the front-end BFD routines automatically determine the format of the input object-file. A descriptor is built in memory with information about which routines are to be used to access elements of the object file's data structure. When the program wants information about the object files, the BFD reads from different sections of the file and processes them. Each BFD back-end will have routines to convert section representations of the object file to BFD's internal canonical object-file format.

The BFD library is a good example of using library routines to develop an RL. Unfortunately, the library itself is very large; the number of functions offered in the front-end are exceptionally many. The BFD front-end was designed in mind to allow programmers to be able to retrieve all types of information about any BFF; at least the existing ones at the time. Due to its generality and bulkiness, it is difficult to use without spending a big overhead on learning how to use it. Perhaps because it is too general, it often contain more information than is needed for system applications.

2.2. DWG-based grammar

An initial attempt at developing a BFF grammar was done by Faase using the AutoCAD's DWG format [7]. In this grammar, terminal symbols consist of a number of bytes and are the fundamental set of base types found in most programming languages: char, int, long, float and double. A binary object file is viewed as a stream of bytes. The grammar supports different byte ordering for integers, and different formats for floating point numbers for various machines. For example, the definition of a word representation in a little-endian machine is given by:

```
type word :=
     byte : first,
     byte : second
     return ((word)first | ((word)second << 8).
```

In a little-endian machine, the lower order byte is stored before the higher order byte. New types are defined as a series of bytes following this rule:

```
type_def_rule :=
    "typedef" data_type basic_type_name ":="
    ("byte" ":" byte_name) LIST
    "return" expr ".".
```

AutoCAD's file format contains a header, several sections and blocks of information related to 2D or 3D drawings. This general file structure resembles that of a simple BFF. However, the information stored in the sections is very different as binary executables contain relocation information, symbol table information, dynamic linking information, and more. Also, since the DWG format is used as the basis for development, the resulting grammar is biased towards DWG. A complete specification for the DWG format can be found in [7].

3. Developing a retargetable loader via specifications

The use of specifications in the development of software engineering tools is not new. Parser and compiler generator tools based on specifications, such as lex [10], yacc [9] and Eli [8], have been around for a while and have proven to be very useful. The input to these generator tools is often a specification, usually in the form of a context free grammar commonly found in specifying the syntax of programming languages. This concept can also be applied to binary objects where each of the BFFs can be specified in a grammar that can be understood by the RL. The resulting object file specification needs to contain information about the structure of the object file and how various sections can be accessed. Each specification acts as a framework for all object files in the group, i.e. a template for all objects belonging to the environment (M,OS,BFF). In programming language terms, the BFF template resembles the variable types while each of the object files is an instance of this type.

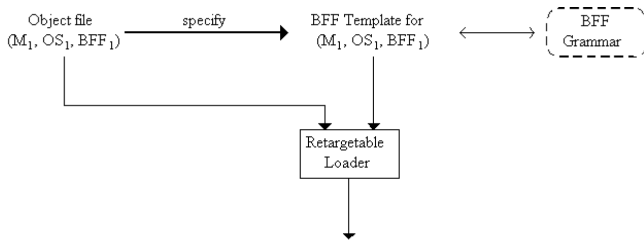

Figure 2. Developing an RL via specification

Figure 2 describes the RL approach; an object file (M1,OS1,BFF1) has a specification template according to the syntax of the BFF grammar. A retargetable loader would use the template information as the basis for processing the object file (M1,OS1,BFF1).

4. Binary-file formats

The general structure of a BFF can be seen to be made up by the following abstraction:

- A header containing general information about the program and information needed to access various parts of the file.
- A number of sections holding code and data (raw data).
- Relocation table(s).
- Symbol table information.

Most BFFs can be mapped to the general model in Figure 3. Information regarding the location of sections, symbol tables, etc are usually identified within the file header. Nevertheless, some BFFs do not distinguish between these structures; in the DOS EXE format, the file header contains information about the relocation table, but there is no information about where the symbol table is stored (if any), and where data is; there is only one section that embodies all code, data and symbol table information. In all cases though, the program's header will contain enough information to determine the entry point (i.e. the start of the program's code) in the file.

The current development domain for our tools is based on the DOS EXE format [6, 12] Windows (16-bit) NE [6, 12] and Solaris ELF [14] BFF formats. These formats vary in their degree of complexity and information stored: the DOS EXE is very simple and limited in structure, whereas the Solaris ELF format is the most complex, while the Windows NE is somewhere in between. For example, for a simple "Hello world" program, using a DOS EXE file will contain a file header, relocation table and a single image for both code and data. The Windows NE version wil

File header
Relocation Table
Symbol Table
Section 1
Section 2
..
..
Section n

Figure 3. BFF abstraction

contain most DOS EXE information plus additional details such as the resource table, entry table, etc. The ELF format contains even more information about the file; sections within the object file hold information used in dynamic linking: code, data, relocation tables, symbol tables, dynamic linking information, etc. Typical "Hello world" binaries for (x86,DOS,EXE), (x86,Windows,NE) and (Sparc,Solaris,ELF) are 6432, 16384 and 5280 bytes long respetively. It can clearly be seen that although the latter two files are dynamically linked, their sizes are not necessarily smaller than the static (first) case. This is due to the small nature of the example program and the inclusion of the DOS EXE header information within the NE format.

5. BFF grammar

In this section we describe the BFF grammar developed based on the EXE, NE and ELF formats. We start by describing some of the properties of BFFs, followed by a description of the grammar itself.

5.1. BFF properties

In a binary file, some parts within the file are interrelated. Although the structure of the binary file does not change at run-time, it's file size, location of sections (or regions) and contents can vary significantly, sometimes having information at the end of the file refering to sections in the middle of the file. Because of this behaviour, the ability of the grammar to reference previously defined information is very important. The resulting grammar not only needs to be general, it must also be flexible to assist the final RL in re-referencing previously parsed information. Information that needs to be re-referenced is usually found in the file header of the binary object file. As the specification for a particular BFF is parsed, any reference to previously read information needs to be handled appropriately.

The ability for a programming language to re-use defined information later in the program can be quite restricted. Although user defined types can be referenced later when

declaring instances, they do not have a value. Macros in languages can be used (referenced) throughout the rest of the program after its definition, but their values are fixed and cannot be changed. In contrast, each binary file of a given BFF has its own set of records that identifies itself. The BFF specification captures the structure of these records, but not its information. The general structure of the BFF is known through the specification, however each instance of this BFF can only be understood by an RL during its parsing process at run-time. The specification defines the items in the file, but their run-time values give meaning to other definitions in the rest of the specification.

The following example will clarify the idea of re-referencing in a BFF specification: let us assume we have a "Hello World" program stored in a Windows NE BFF. The segment table for the Windows NE BFF consists of a fixed number of segment table entries. The exact number of entries is listed as one of the fields in the program's file header–NumSegEnt. To create a copy of the segment table for the "Hello World" program in memory, the RL must allocate the number of entries according to NumSegEnt. In the Windows NE specification, the definition of the file header and segment table could be as follows:

```
FileHeader : STRUCTURE {
  ..
  NumSegEnt : int;
  ..
}
SegmentTable : ARRAY NumSegEnt OF SegTableEnt;
```

In other words, the value of NumSegEnt is used to specify the size of the array SegmentTable. In traditional languages, the array size needs to be specified at compile time. However, in this language, the size of the segment table is only known at run-time, hence the array is allocated at run-time when the information becomes available. This behaviour is what we refer to as *re-referencing* of information. Most re-referenced information is located in the file header, but sometimes this is not the case. For example, to locate the segment table, the address where it can be located must be defined:

```
SegmentTable : ARRAY NumSegEnt OF SegTableEnt;
               ADDRESS NewHoff + SegToff;
```

The address is the addition of `SegToff`, found within the second header, and `NewHoff`, located in the first header.

5.2. BFF grammar for simple retargetable loader

SRL, a simple retargetable loader, was constructed to develop a tool that would support the developed BFF grammar, and generate C code for the loading of a binary file stored in the specified binary-file format. SRL requires a generic BFF grammar such that it can be easily extended if later found insufficient to describe other BFFs. Our focus has been mainly on three BFFs: DOS EXE, Windows (16 bit) NE and Solaris ELF formats. The difference in complexity between these BFF (DOS EXE–simple, Solaris ELF–very general and Windows NE–moderate) gives an indication of how well the grammar works. We present the abstract syntax of BFFG, SRL's BFF grammar. The grammar syntax is in extended BNF (EBNF) format. EBNF has the following language symbols:

- Sequences are denoted by {}. E.g. {rep} specifies zero or more repetitions of rep.
- Optional is denoted by []. E.g. [opt] specifies zero or one occurrences of opt.
- Selection is denoted by |. E.g. A | B | C specifies a choice between A, B and C.

In the grammar, *non-terminals* appear in italics, terminals appear in normal fontface, "literal strings" appear with double quotes, and `examples` appear in courier. The start symbol for this grammar is *BFFspec*:

BFFspec => {*spec*}.
spec => *format-def defin* {*defin*} *load-info*
format-def => "DEFINITION" "FORMAT" *ident* {*ident*} "END" "FORMAT"
defin => "DEFINITION" *ident* "ADDRESS" *expression scope-def* "END" *ident*.
load-info => "FILEHEADER" *ident* "IMAGESIZE" *expression* "IMAGEADDRESS" *expression*
scope-def => *ident type-exp* {*ident type-exp*}
type-exp => "SIZE" *expression* | "ARRAY" *expression scope-def* "END" *ident*
expression => "(" *ident operator expression* ")" | *ident operator expression* | ϵ
operator => "+" | "-" | "*" | "/" | "^" | "%"
ident => "a".."z" | "A".."Z" {"a".."z" | "A".."Z" | "_"}

Hence, the body for any BFF specification is of the form:
spec => *format-def defin* {*defin*} *load-info*

5.2.1 format-def

format-def => "DEFINITION" "FORMAT" *ident* {*ident*} "END" "FORMAT"

The *format-def* non-terminal specifies the overall structure of the BFF. All *ident* names used for sections/divisions of the binary file defined as part of *format-def* must be defined later in the specification although the format need not include all parts of a BFF; i.e. some entries can act as space fillers that separate different parts of the file. An example definition of *format-def* for a simple BFF format is :

```
DEFINITION FORMAT
  file_header
  section
END FORMAT
```

In this example, `file_header` and `section` will need to be defined later on in the grammar. If we want to specify the DOS EXE file, then the relocation table would go between the `file_header` and `section`. However, if we are not concerned with its details, it can be omitted from the definition. The organisation of identifiers is not forced; it merely indicates the relative ordering of divisions. The above definition does not suggest that `section` starts at the end of `file_header`; in fact, `section` could be placed before `file_header`. The syntax of *format-def* does not place any ordering restrictions on it. But for clarity and ease of understanding, the user should arrange the definitions in a well-formed manner so that it reflects the actual file's structure.

5.2.2 defin

Each declared identifier *ident* in a *format-def* is defined using the following *defin* rule:

defin => "DEFINITION" *ident*
"ADDRESS" *expression*
scope-def
"END" *ident*

Defin declares a new structure (section or block of the file). The *ident* that follows the keyword DEFINITION must be previously declared in the grammar. The start location of this new structure (relative to the start of the file) is specified by the expression after the keyword ADDRESS; e.g. the definition of the `file_header` might look like this:

```
DEFINITION file_header ADDRESS 0
  h_sigLo SIZE 8
  h_sigHi SIZE 8
  h_lastPageSize SIZE 16
  ..
END file_header
```

The above definition indicates that the `file_header` starts at the beginning of the file (i.e. offset 0). All declarations that follow the ADDRESS belong to this definition; in the above case the `file_header`. The entries `h_sigLo`, `h_sigHi` and `h_lastPageSize` all belong to the same scope level and have a parent named `file_header`. This concept is equivalent to the definition of a structural type in most programming languages.

5.2.3 Loading-info

load-info => "FILEHEADER" *ident*
"IMAGESIZE" *expression*
"IMAGEADDRESS" *expression*

Load-info holds the fundamental information about the object file for loading to occur. It is crucial for any BFF specification to provide its loading information. There is no order on the occurrence of the three constructs, as long as all three exist in the specification. The FILEHEADER construct identifies the first region of the object that must be loaded into memory. This region is often the file header as it contain critical information about the locations of other regions and some house keeping information. The IMAGESIZE specifies the load image size; the size is often calculated based on the information obtained in the file header. The IMAGEADDRESS specifies the start address (relative to the beginning of the file) where the image is stored.

5.2.4 scope-def

scope-def => *ident type-exp* {*ident type-exp*}

Scope-def captures all information belonging to the same scope level within one structure. Its properties are analogous to the structural types in the language C. An *ident* name and type information defines each field within the scope.

5.2.5 type-exp

type-exp => "SIZE" *expression* |
"ARRAY" *expression scope-def*
"END" *ident*

Type-exp defines the type for the identifier *ident* in bits. An identifier can be either a single element of a particular size or a group that is specified by the ARRAY construct. The *expression* after the keyword ARRAY identifies the number of elements in the ARRAY. Declarations within the ARRAY definition are bounded to the same scope, with array identifier being their parent. For example, the definition of the segment table in the Windows NE format follows:

```
DEFINITION seg_table ADDRESS (sh_segToff + sho_off)
  seg_table_ent ARRAY sh_segTent
    ste_logSectoff SIZE 16
    ste_size SIZE 16
    ste_flag SIZE 16
    ste_minsize SIZE 16
  END seg_table_ent
END seg_table
```

In the Windows NE format, the segment table is defined to be an array of structures named `seg_table_ent`. The number of array elements is `sh_segTent`, which must have been parsed earlier in the specification and its value is used at run-time. `ste_logSectoff`, `ste_size`, `ste_flag` and `ste_minsize` are the components (or fields) of one element of `seg_table_ent`.

6. Experimentation

SRL, the Simple Retargetable Loader, is an attempt to demonstrate the benefit of using an RL to build a machine-code manipulation tool. It is a scaled downed version of an RL and is implemented in C. The SRL is limited in a way by its simple grammar which contains a small number of constructs. As described in the previous section, the BFFG for the SRL was constructed using three different base environments: (x86, DOS, EXE), (x86, Windows, NE) and (Sparc, Solaris, ELF). The ELF (on a RISC architecture) being the most complex BFF of the three, the EXE (CISC) being the simplest, and the NE (CISC) somewhere in between. These three formats allowed the develoopment of a generic BFF grammar for the SRL.

When implementing an RL, one must consider to what extent does this RL take part in the decoding of the binary file. Does it decode the whole file and rewrite it to another representation or does it simply load the whole file to memory? How much detail is interpreted by the RL? In the case of SRL, the primary function was to produce a high-level language interface to represent the structures of the binary file (i.e. a header file in C), and the loading of the program's image (i.e. a C file).

The input to the SRL is the BFF specification: a binary description of the object file for an environment (M,OS,BFF) written in SRL's syntax grammar. Figure 4 is a description of what the SRL produces. The object structures are the type definitions for various regions of the binary executable file. The loading routine contains initialized information for the object structures and loading of the object image to memory. The object structure and loading routine are implemented as .H and .C files respectively using the C language.

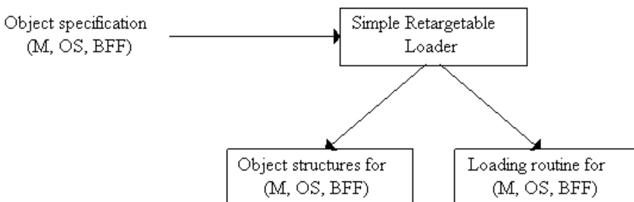

Figure 4. The Simple Retargetable Loader (SRL)

The specifications for (x86, DOS, EXE), (x86, Windows, NE) and (Sparc, Solaris, ELF) were used as inputs to the SRL and the set of corresponding .H and .C interface files were produced. The SRL output for the (x86,DOS,EXE) specification is listed in Figures 5 and 6. The data structure for manipulating the binary is generated in the .H file while the .C file provides a function `LoadImage()` which ini-

```
/* This file is generated by the BFF
generator using the grammar in
"dosexe.txt" */

 #ifndef _LOAD_H_
 #define _LOAD_H_

 #ifdef __MSDOS__
   #define INT  int
   #define LONG  unsigned long
 #else
   #define INT  short int
   #define LONG  unsigned int
 #endif __MSDOS__

 typedef unsigned char byte;
 typedef short int16;

 #define LH(p) ((int16)((byte*)(p))[0]+
 ((int16)((byte*)(p))[1]<<8))

 typedef struct {
   byte h_sigLo;
   byte h_sigHi;
   INT  h_lastPageSize;
   INT  h_numPages;
   INT  h_numReloc;
   INT  h_numParaHeader;
   INT  h_minAlloc;
   INT  h_maxAlloc;
   INT  h_initSS;
   INT  h_initSP;
   INT  h_checkSum;
   INT  h_initIP;
   INT  h_initCS;
   INT  h_relcTabOffset;
   INT  h_overlayNum;
 } headerT;

 typedef struct {
   headerT      *header;
   byte*        section;
   char*        filename;
   LONG         imagesize;
   byte*        image;
 } BFF;

 extern BFF* aBFF;

 LoadImage(char* filename);

 #endif _LOAD_H_
```

Figure 5. loader.H file generated by SRL

tializes the aBFF structure and finds the entry point to the program.

Surprisingly, the specification for the Windows NE managed to be larger than that for the ELF, although the ELF is presumed to be the most complex format of the three. Perhaps if the grammar had more constructs, then finer details could be captured. In that case, we would see more of the ELF structure. But is that part of the loader? Or does it depend on the needs of the manipulation tool? Does the loader need to examine and be able to identify and understand all the different regions of the object file? How much disposure should the loader know? Due to all these questions, a complete interface is still left as discussion and work in progress.

To examine the usability of the SRL's output, thus demonstrating the generality and usefulness of an RL; the generated loading files (.H and .C) produced from the (x86,DOS,EXE) specification were integrated with the DCC decompiler [3]. The hand-coded loading module for the Intel 286 DCC decompiler was replaced with the corresponding SRL outputs. With a few minor changes (calls to function names and variables used were changed), DCC was reconstructed using the generated loading files. The behaviour for the two versions of DCC was the same, hence demonstrating the correctness of the SRL output.

6.1. Interface

The interface routines produced by the SRL are very simple. The .C file merely provides a loading module for setting up an image in memory (see Figure 6). This is fine for a DOS EXE format as it is extremely simple, but for other types of BFFs, one would like to provide interface functions to access different regions of the binary file. For example, the Windows NE BFF has a number of tables – imported-name table, segment table, module reference table, etc. The structure of the segment table in the specification is:

```
DEFINITION seg_table ADDRESS (sh_segToff + sho_off)
  seg_table_ent ARRAY sh_segTent
  ste_logSectoff SIZE 16
  ste_size SIZE 16
  ste_flag SIZE 16
  ste_minsize SIZE 16
  END seg_table_ent
END seg_table
```

The SRL creates the structure for the segment table in the .H file and sets up a pointer that points to the beginning of the table in the image. There are no routines generated from the SRL for accessing this structure. If the programmer wants to access a particular entry in the table, then he/she must directly manipulate this structure by hand crafting that piece of code.

A desirable feature for an RL would be to automatically generate a set of interface routines, thus eliminating the need for the programmer to hand code routines to manipulate the structures directly.

6.2. Limitation of SRL's BFFG

The SRL's BFFG was designed to be as simple as possible. It merely provides a most basic framework model for creation of elementary loading routines. Most of the SRL functions deal with information about the file header and apply its definitions to the rest of the object file.

There are a number of areas that the SRL grammar does not include:

- Relocation information is not capture by the SRL but can be included easily by adding new constructs to the BFF grammar.
- System architectures such as big and little endian machine types are undefined. Such details need to be added to the grammar as well. The techniques used in describing the DWG format can be used to identify the byte ordering of the processor:

  ```
  type word :=
  byte : first,
  byte : second,
  return ((word)first | ((word)second << 8).
  ```

  Alternatively, a much simpler way would be just reserving the words big-endian or little-endian and let the SRL handle the byte ordering for these machines.

7. Summary and conclusions

There are essentially three basic approaches to provide loading of a binary object: handcraft the code, use library routines or use specifications. A retargetable loader can be built using library routines or specifications. Using library routines is simpler but can be difficult; attempts to use tools such as the BFD library are uninviting due to their complexity. Specifications are easily understood and are trouble free once they have been developed. It is an ideal method to develop a retargetable loader (RL) based on a binary-file format (BFF) grammar.

There are a few differences between grammars used in programming languages and the grammar used for describing BFFs. The most significant difference is the ability of the BFF grammar to re-reference information that was previously defined. Previously defined information is critical in binary file processing: addresses and segment sizes are usually controlled by definitions found in the file header and their values are determined only at run-time.

```
/* This file is generated by the BFF generator
using the grammar in "dosexe.txt" */

 #include <stdio.h>
 #include <string.h>
 #include "loader.h"

 BFF    *aBFF;

 LoadImage(char* filename) {
   FILE *fp;
   LONG imageaddress;

   if ((fp=fopen(filename, "rb"))==NULL) {
     printf("cannot open file ");
     return 0;
   }
   aBFF = (BFF *)malloc(sizeof(BFF));
   aBFF->header = (headerT *)malloc(sizeof(headerT));
   if (fread(aBFF->header, sizeof(headerT), 1, fp) != 1) {
     printf("cannot read file ");
     return 0;
   }
   aBFF->imagesize = LH(&aBFF->header->h_numPages) * 512 - LH(&aBFF->header->h_numParaHeader)
     * 16 - (512 - LH(&aBFF->header->h_lastPageSize));

   aBFF->image = (byte *)malloc(aBFF->imagesize);
   fseek(fp, (Int)LH(&aBFF->header->h_numParaHeader) * 16, SEEK_SET);
   if (fread(aBFF->image, (size_t)aBFF->imagesize, 1, fp))!=1) {
     printf("error reading image ");
     return 0;
   }
   imageaddress = LH(&aBFF->header->h_numParaHeader) * 16;

   aBFF->section = aBFF->image + LH(&aBFF->header->h_initIP) + 16 - imageaddress;
   aBFF->filename = (char*) malloc(sizeof(char) *  (strlen(filename)+1));
   strcpy(aBFF->filename, filename);
   fclose(fp);

 }  /* LoadImage */
```

Figure 6. loader.C file generated by SRL

SRL, a simple retargetable loader, is a first attempt to develop an RL with a simple BFF grammar. To demonstrate how the SRL grammar works, specifications for (x86,DOS,EXE), (x86,Windows,NE) and (Sparc,Solaris,ELF) were created and used as input to the SRL. The SRL outputs a set of object structures (.H file) and loading routines (.C file) for each of the specifications. The SRL outputs for the (x86,DOS,EXE) specification file were incorporated into an existing binary-manipulation tool (the DCC decompiler) by replacing its loading modules with the SRL's loading output. The integration was successful and the program behaved indifferently as before.

A retargetable loader has a lot of potential. Being able to capture different binary-file structure is a big plus for software developers wanting to write machine-code manipulation tools. Its ability to express BFF structure and provide

an almost automatic way to generate loading information benefits particularly in the area of binary translation. There are still a lot of problems that are unsolved in this area. Issues like how much detail is exposed to the loader, and how to create a proficient and flexible RL still need to be addressed. The most efficient and ideal BFF grammar that contain the most general structures and constructs for all system environments has yet to be developed.

SRL and its BFFG are currently under development. SRL is part of an on going project on binary translation at the University of Queensland, Australia. For current information on the project refer to http://www.it.uq.edu.au/groups/csm/bintrans.html.

Acknowledgments

We would like to thank the anonymous referees for comments on improvements needed in this paper.

References

[1] AT&T. Flashport. http://www.att.com/ FlashPort/, 1994. AT&T Bell Labs.

[2] S. Chamberlain. *libbfd – The Binary File Descriptor Library*, first edition – BFD version 3.0 edition, Apr. 1991.

[3] C. Cifuentes and K. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995.

[4] A. Corporation. Macintosh application environment. http://www.mae.apple.com/, 1994.

[5] Digital. Freeport express. http://www.novalink.com/ freeport-express, 1995.

[6] R. Duncan. *The MSDOS Encyclopedia*, chapter 4, pages 107–147. Microsoft Press, 1988.

[7] F. Faase. DWG file format. http:// wwwis.cs.utwente.nl:8080/ faase/BFF/dwg_f.html, 1996.

[8] R. Gray, V. Heuring, S. Levi, A. Sloane, and W. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, Feb. 1992.

[9] S. Johnson. YACC - yet another compiler-compiler. Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[10] M. Lesk. LEX - a lexical analyzer generator. Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, Sept. 1996.

[12] Microsoft. Miscrosoft windows software development kit (SDK) for windows. Article ID: Q65260.

[13] R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, Feb. 1993.

[14] SunSoft. Wabi. http://www.sun.com/sunsoft/Products/PC-Integration-products/, 1994.

[15] T. Thompson. An Alpha in PC clothing. *Byte*, pages 195–196, Feb. 1996.

[16] W. Waite. Compiler construction: Craftsmanship or engineering? In T. Gyimóthy, editor, *Proceedings of the International Conference on Compiler Construction*, Lecture Notes in Computer Science 1060, pages 151–159, Linköping, Sweden, 24-26 April 1996. Springer Verlag.