DOI: 10.2298/CSIS130120071K

Context Parsing (Not Only) of the Object-File-Format Description Language

Jakub Křoustek and Dušan Kolář

Faculty of Information Technology, IT4Innovations Centre of Excellence
Brno University of Technology, Božetěchova 1/2, 612 66 Brno, Czech Republic
{ikroustek,kolar}@fit.vutbr.cz

Abstract. The very first step of each tool such as linker, disassembler, or debugger is parsing of an input executable or object file. These files are stored in one of the existing object file formats (OFF). Retargetable tools are not limited to any particular target platform and they have to deal with handling of several OFFs. Handling of these formats is similar to parsing of computer languages — both of them have a predefined structure and a list of allowed constructions. However, OFF constructions are heavily mutually interconnected and they create context-sensitive units. In present, there is no generic system, which can be used for OFF description and its effective parsing.
In this paper, we propose a formal language that can be used for OFF description. Furthermore, we present a design of a context parser of this language that is based on the formal models. The major advance of this solution is an ability to describe context-sensitive properties on the level of the language itself. This concept is planned to be used in the existing retargetable decompiler developed within the Lissom project. In this project, the language and its parser will be used for an object file parsing and its automatic conversion into the internal uniform file format. It is important to say that the concept of this parser can be utilized within other programming languages.

Keywords: object file format, context parsing, scattered context grammar, priority function, attributed grammar, decompilation, Lissom, ELF

1. Introduction

Reverse compiler (i.e. decompiler) is yet another tool that takes executable files on its input. Its purpose is to translate this input into a high level language (HLL) representation, such as a C code. This tool can be used for source code reconstruction, binary code migration, malware analysis, etc. Retargetable decompilation is a more difficult task because it must handle all the platform, operating system, and programming language specific features.

Platform-specific decompilation is a well-described discipline, e.g. see [5,7,40]. On the other hand, retargetable (i.e. platform-independent) decompilation is still a quite unexplored area, despite the first attempts done decades ago. However, several steps of retargetable decompilation have been already done, such as

uniform extraction of instruction semantics, machine-code decoding, reverse compilation into HLL, etc. See [43,44] for details.

However, there is still one non-covered phase of retargetable decompilation — the handling and conversion of platform-dependent object file formats (OFFs). This is the preliminary step of decompilation. Within this step, an input file is being analyzed, validated, and converted into the internal, uniform representation. This conversion transforms all the necessary information (e.g. machine code, data, symbols) into the internal structures. It might be possible to use one of the existing single-purpose converters or write a new one from a scratch. However, none of these is a truly generic solution.

This problem can be divided into two tasks. (1) Description of a target OFF using a specific description language. (2) Automatic generation of an OFF converter based on this description. Afterwards, the converter can be used for conversion of applications stored in a particular OFF into an internal code representation used by a decompiler.

Structure of most OFF is relatively complicated (e.g. Windows PE, UNIX ELF) because its elements are mutually interconnected and the structure is heavily influenced by content of these elements. We can say that these elements create a context-sensitive behavior. This is a problem for design of such OFF description language because the theory of computer-language compilation settled down on the concept of context-free parsing for most of the existing languages during the last sixty years.

Within the context-free parsing concept, syntax of programs is usually processed using automatically generated context-free parsers. Parser generators like YACC, Bison, or ANTLR are able to create a skeleton of target language-specific parser. However, this skeleton has to be enriched of hand-written HLL code implementing semantics checking (so called semantic actions). This concept is prone to errors and each change of the target language needs reimplementation of the parser (at least its semantic actions).

In this paper, we present a new formal language for the description of OFFs that is capable to describe context-sensitive elements. We propose a context parser of this language that is based on the newly created formal models (attributed scattered context grammar with priority function, etc.).

The concept is planned to be used in the existing retargetable decompiler developed within the Lissom project [23]. In this project, the OFF language is used for object-file handling and its automatic conversion into the internal Common-Object-File-Format (COFF)-based file format, which is processed by the decompiler afterwards, see [45] for details. Moreover, the language is designed to be general enough for usage in other retargetable tools (e.g. loaders, disassemblers, debuggers) and the context parser can be used for parsing of different programming languages, not just OFF description language.

The paper is organized as follows. Section 2 introduces some preliminaries. Section 3 briefly characterizes common OFFs. Then, we discuss existing conversion techniques and applications in Section 4. The Lissom project is briefly described in Section 5. Our language for OFF description is presented together

with an example of its usage in the subsequent Section 6. Within this section, we also depict several context-sensitive features of this language. In Section 7, we present a concept of the context parser as well as the definition of the new formal models that the parser is based on. We also give a short overview on the current state of the parser's implementation together with experimental results within the same section. Finally, discussion of future research closes the paper in Section 8.

2. Preliminaries and Definitions

We assume a reader is familiar with the formal language theory (for further reference, see for example [26]).

Definition 1. A *phrase-structure grammar* is a quadruple

$$G = (V, T, P, S),$$

where

- V is a *total alphabet*;
- $T \subset V$ is a finite set of *terminal symbols* (*terminals*);
- $S \in V - T$ is the *start symbol* of G;
- P is a finite *set of productions* $p = x \to y$, $x \in V^*(V-T)V^*$, $y \in V^*$.

The symbols in $V-T$ are referred to as *nonterminal symbols* (*nonterminals*). We set lhs$(p) = x$ and rhs$(p) = y$, which represents the *left-hand side* and the *right-hand side* of the production p, respectively.

Definition 2. A *context-sensitive grammar* (CSG) is a phrase-structure grammar

$$G = (V, T, P, S),$$

such that every production $p = x \to y \in P$ satisfies $|x| \leq |y|$.

Definition 3. A *context-free grammar* (CFG) is a phrase-structure grammar

$$G = (V, T, P, S),$$

such that every production $p = x \to y \in P$ satisfies $A \to x$, where $A \in V-T$ and $x \in V^*$.

Definition 4. A *scattered context grammar* (SCG, see [11]) is a quadruple,

$$G = (V, T, P, S),$$

where

- V is a total alphabet;
- $T \subset V$ is a finite set of terminals;

- $S \in V - T$ is the start symbol;
- P is a finite set of productions of the form

$$(A_1, \ldots, A_n) \rightarrow (x_1, \ldots, x_n),$$

where $A_i \in V - T$, $x_i \in V^*$ for all $i : 1 \leq i \leq n$.

Definition 5. A *propagating scattered context grammar* (PSCG) is a SCG

$$G = (V, T, P, S),$$

in which every $(A_1, \ldots, A_n) \rightarrow (x_1, \ldots, x_n) \in P$ satisfies $x_i \in V^+$ for all $i : 1 \leq i \leq n$.

Definition 6. Let $G = (V, T, P, S)$ be a (propagating) SCG. If

$$y = u_1 A_1 u_2 \ldots u_n A_n u_{n+1},$$
$$z = u_1 x_1 u_2 \ldots u_n x_n u_{n+1},$$

and $y, z \in V^*$, $p = (A_1, \ldots, A_n) \rightarrow (x_1, \ldots, x_n) \in P$, then y *directly derives* z in the SCG G according to the production p,

$$y \Rightarrow_G z \; [p] \text{ (or simply } y \Rightarrow_G z\text{)}.$$

Let $\Rightarrow_G^+$ and $\Rightarrow_G^*$ denote the *transitive* and the *reflexive-transitive closure* of $\Rightarrow_G$, respectively. To express that G makes the *derivation* from u to v by using the sequence of productions $p_1, p_2, \ldots, p_n \in P$, we write $u \Rightarrow_G^* v \; [p_1 p_2 \ldots p_n]$ (or $u \Rightarrow_G^+ v \; [p_1 p_2 \ldots p_n]$ to emphasize that the sequence is non-empty). We abbreviate $\Rightarrow_G$ to $\Rightarrow$ when it is clear which grammar we are referring to. This definition also holds for other SCG-based grammars listed below.

Now we are able to define scattered context grammars regulated by priority functions, see [21] for details of their properties.

Definition 7. A *(propagating) scattered context grammar with priority*, abbreviated as ((P)SCGP), is a quintuple

$$G = (V, T, P, S, \pi),$$

where (V, T, P, S) is a (propagating) scattered context grammar and π is a *priority function*

$$\pi : P \rightarrow \mathbb{N}.$$

Definition 8. Let $G = (V, T, P, S, \pi)$ be a (P)SCGP. We say that y directly derives z in (P)SCG G according to the production p, $y \Rightarrow_G z \; [p]$ (or simply $y \Rightarrow_G z$), if and only if:

- $y = u_1 A_1 u_2 \ldots u_n \; A_n u_{n+1} \in V^*$,
- $z = u_1 x_1 u_2 \ldots u_n x_n u_{n+1} \in V^*$,
- $p = (A_1, \ldots, A_n) \rightarrow (x_1, \ldots, x_n) \in P$, and

- there is no $p' = (A'_1, \ldots, A'_n) \rightarrow (x'_1, \ldots, x'_n) \in P$, such that:
  1. $y = u'_1 A'_1 u'_2 \ldots u'_n A'_n u'_{n+1} \in V^*$, and
  2. $\pi(p') > \pi(p)$.

Definition 9. A *(propagating) scattered context language with priority* is language generated by a (propagating) scattered context grammar with priority. The family of (propagating) scattered context languages with priority is denoted by $\mathcal{L}(\text{(P)SCP})$. In [21], it has been proved that

$$\mathcal{L}(CS) = \mathcal{L}(PSCP) \subset \mathcal{L}(RE) = \mathcal{L}(SCP),$$

where RE stands for the set of all recursively enumerable languages.

3. Object File Formats

The term *object file format* refers to a format of an executable code, library code, or object code that has not been linked yet. In the following text, we focus mainly on the executable code. A generic OFF usually consists of the following parts [22]:

- *Header* – contains essential information about the file (e.g. its identification, size, section pointers);
- *Object code* – i.e. sections containing machine code and application data;
- *Relocations* – "*Relocation is the process of assigning load addresses to the various parts of the program, adjusting the code and data in the program to reflect the assigned addresses*" [22]. We can find a wide range of relocation types for each target architecture. Some relocations can be resolved during compilation by linker; while the other ones has to be resolved by loader before program's execution;
- *Symbols* – symbols are usually stored in tables and they characterize its local, imported, and exported symbols (variables, functions, etc.);
- *Debugging information* – generated by compilers for debug support. There exist several debugging information standards [20]. The presence of the debugging information is optional.

Unfortunately, there is no such generic format and each platform (i.e. a combination of an operating system and a processor architecture) has its own format, or a derivative of an existing one. In present, we can find two major OFFs — UNIX ELF [39] and Windows PE [29], see Fig. 1. However, other formats are on arise (e.g. Apple Mach-O), see [19]. In the Lissom project [23], a COFF-based file format is used for internal code representation. The overview of other common formats can be found in [19].

The **UNIX ELF** [39] file format is a standard on all UNIX-like systems. It is independent on a particular target architecture (e.g. Intel IA-32, SPARC, ARM). The leading part of the ELF file is a header with all the essential information. It also points to the program and section header tables. These tables contain information about particular segments and sections, respectively (e.g. their sizes,

offsets within the file). Each section can store different content (e.g. code, data, symbol, hash tables); furthermore, one or more sections may form a segment.

From the linker point of view, an ELF file consists of a group of sections defined in a section-header table. Contrariwise, loader handles the ELF file as a group of segments defined in a program-header table, see 1. The very important characteristic of this format is its flexibility. Only the header has a fixed offset within the file, all other elements are optional, as well as their offsets within the file. Therefore, all elements are scattered throughout the file, and the size or content of padding is unspecified.

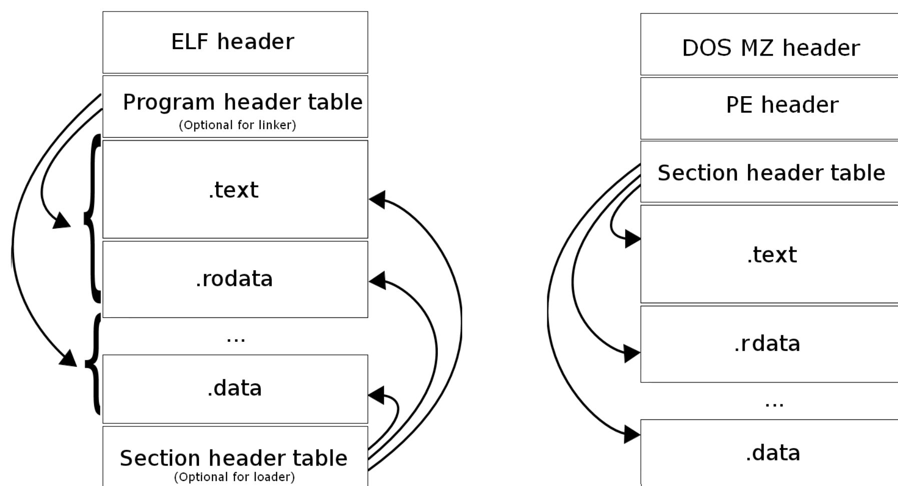

Fig. 1. ELF (on left) and Windows PE (on right) file formats.

The **Microsoft's Windows Portable Executable** (WinPE) [29] format also supports all the three file types — object files, executables, and libraries. Windows PE can be used on all Windows-based systems on architectures Intel IA-32, IA-64, x86-64, ARM, and others. The structure of the PE format is based on the COFF format [10]. It consists of a number of headers and sections that tell the loader how to map the file into memory. Each section has its own header and often a specific purpose, for illustration see 1. For example, the `.text` section holds the program code; `.data` sections hold global variables, `.edata` and `.idata` sections contain exported and imported symbols, etc.

The **E32Image** format is used on the Symbian operation systems, usually used in smartphones [31]. It was developed by Symbian Ltd., which currently belongs to Nokia. E32Image is used only on the ARM architecture.

E32Image is a proprietary format, and its specification has never been publicly published; therefore, all the necessary information was gathered using reverse engineering. This format was originally based on Windows PE, but since Symbian version 9.1 (in 2005), its authors switched to an ELF-like format. The

E32Image file is created from an existing executable PE or ELF file by a special post-linker. The main idea of this format is to provide basic file format structure with low-memory overhead. The differences over the mentioned formats include that the same type sections are merged and might be compressed, information about the target architecture (e.g. word size) is not explicitly encoded, and that unnecessary strings (e.g. symbol names) might be removed.

The **Mach-O** object file format [3] is used in operation systems Darwin, NeXTSTEP, Mac OS X, or iOS from Apple Inc. It is made up of three parts — Mach-O header, followed by a series of load commands, and one or more segments, each of them containing up to 255 sections. Mach-O supports Intel IA-32, x86-64, PowerPC, and PowerPC64 as the target architectures.

A special feature of this format is its support of multi-architecture binaries, where multiple Mach-O files can be combined in a single multi-architecture file. Such binary file contains code for multiple instruction set architectures.

4. Related Work

We can find several projects focused on parsing and binary conversion of OFFs. They are used mostly in reverse engineering or for code migration between particular platforms. The largest group consists of hand-coded tools that are focused on binary conversion between two particular OFFs.

A typical example is the Macintosh Application Environment project [2], which supports execution of native Apple Macintosh applications on UNIX based workstations. AT&T's FreePort Express [6] is another binary translator, which permits conversion of SunOS and Solaris executables into Digital UNIX executables. Wabi allows conversion of executables from Windows 3.x to Solaris [12].

Another important project is the Binary File Descriptor library (**BFD**) [4]. BFD was developed by the Cygnus Support company, and currently forms a part of the GNU Binutils package. It supports unified, canonical format for manipulation tens of OFFs (e.g. ELF, PE, COFF). BFD is used as a front-end of many existing projects, however, it is not a retargetable solution because support of each new OFF must be hand-coded. Furthermore, due to BFD's complexity, the interconnection of the target application and BFD is often difficult. Details about a successful BFD-based solution can be found in [19].

The last group of projects uses their own grammar-based systems for a formal description of binary formats. The architecture description language (ADL) **SLED** [32], developed within the New Jersey Machine Code Toolkit [33], is supposed to describe the instruction sets of target processor architectures, i.e. syntax and binary encoding of each instruction. Such description can be used for the automatic generation of retargetable linker [9], debugger [34], or other tools. However, this language does not support description of OFFs. We can find the same limitation in all common ADLs, see [25] for more details.

We can find two formalisms called **BFF grammar** (Binary File Format Grammar). The first one (**DWG BFF**) [8] was originally designed for description of non-executable file formats. More precisely, it was designed and tested only on

the AutoCAD DWG format. This grammar is a state-of-the-art concept, which has never been implemented, nor used in any tool. The grammar is limited to the DWG format, but it can be be possibly used on other OFFs. Its author claims (see [8]) that the grammar is in LL(1) form and it can be parsed by the recursive descent approach.

The DWG BFF grammar inspired project UQBT [42] and its **SRL BFF** grammar [41]. Despite the limitations of the original DWG BFF grammar, it was simplified and used within this project for generation of the Simple Retargetable Loader (SRL). Although, it is claimed that this concept can be used for automatic generation of other retargetable tools, the grammar constructions are limited only for a simple loader. According to [41], the SRL was tested as an ELF loader for existing decompiler `dcc` [5].

Both BFF grammars have several limitations. For example, they are unable to properly model optional elements of OFFs, such as the missing section header table in ELF; relocation information is not taken into account, etc. The most significant drawback of both grammars is the lack of semantic actions [1] (e.g. semantic checks, validation of OFF content, user-defined actions), see Figure 2. Therefore, the grammars are only capable to describe syntactical structure of the input OFFs, but modeling of context-sensitive properties is left on the user.

```
DEFINITION FORMAT
    header
    program_header_table
    sections
    section_header_table
END FORMAT

DEFINITION header
    h_ident SIZE 16
    h_type SIZE 16
    h_machine SIZE 16
    h_version SIZE 32
    ...
END header
```

Fig. 2. Example of the BFF grammar description of the ELF format [42].

In conclusion, none of the previously mentioned concepts can be used for effective handling of OFFs for retargetable decompilation.

5. Lissom Project's Retargetable Decompiler

The Lissom project's [23] retargetable decompiler aims to be independent on any particular target architecture, operating system, or OFF. It consists of two main parts—the preprocessing part and the decompilation core, see Figure 3. Its detailed description can be found in [45,43]. The decompilation process consists of the following phases.

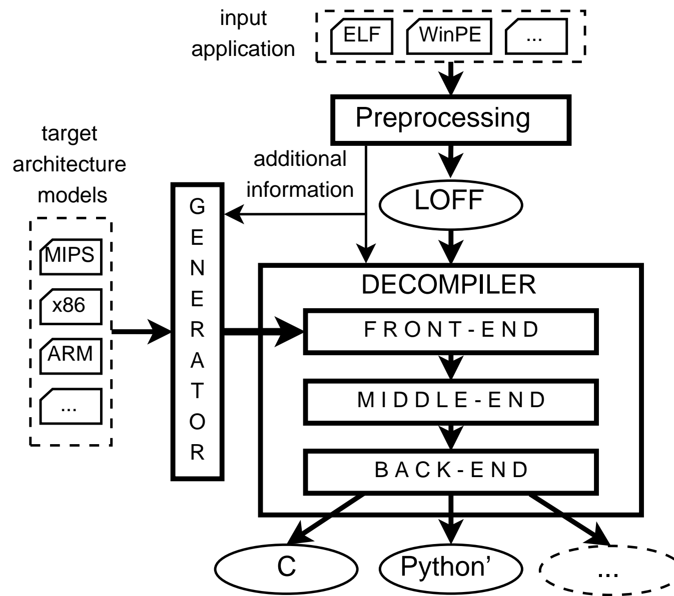

Fig. 3. The concept of the Lissom project's retargetable decompiler.

(1) At first, the input binary executable file is transformed using a plugin-based *binary file converter* from a particular OFF (e.g., Windows PE, ELF) into its own internal object-file-format called **LOFF** (Lissom Object File Format) [16]. LOFF was designed in reference to independence on any particular architecture, universality, and to be well readable. Therefore, it is possible to describe architectures with different types of endianity, byte sizes, instruction lengths, or instruction alignments. It is also possible to store executable, object, or library code within the LOFF format.

The LOFF structure is similar to the COFF format. Basically, it has one header, followed by section headers, sections, and symbolic information (symbols, relocations, and debug information). The section's content is characterized by section flags. The format of LOFF is textual; therefore, it is possible to study its content without any additional tools, see Figure 4. LOFF is used by a complete set of retargetable tools that are automatically generated in this project (e.g. retargetable disassembler, simulator, and decompiler).

```
AgT62kG9y7   // Magic string
32           // Word bit-size
4            // Bytes per word
0            // Byte order, 0-little, 1-big
X            // Flags (eXecutable, etc.)
1            // Is the entry point set?
143654972    // Byte address of the entry point
30           // Section count
1            // Symbol table count
...          // Information about sections
.text        // Section header name
0            // Section byte alignment
1            // Is address absolute?
143654972    // Section address
T            // Section flags (Text, Data, BSS, etc.)
10536        // Section data size in bytes
0            // Count of relocations
20           // First line of section data
0            // First line of relocation data
             // Section data follows
00111111110000001111110000000101 // .text section data
00000000000000000000000000000000
```

Fig. 4. Simplified example of the LOFF format (several attributes are not listed).

In present, the conversion plugin from each supported OFF into LOFF is hand written; thus, the converter is not truly retargetable yet. This is the reason why we need a fully-automatic retargetable solution, such as presented in this paper.

(2) Afterwards, the LOFF file is processed in the *front-end* part which is partially automatically generated based on the description of target architecture (e.g. MIPS, ARM, Intel x86). The architecture description language ISAC [25], developed also within the Lissom project, is used for this purpose. This decompilation phase is responsible for decoding of machine-code instructions, their static analysis, and detection of HLL constructions (e.g. loops, functions). The resulting code is emitted as LLVM IR [24], which is used as an internal code representation of decompiled applications in the remaining decompilation phases.

(3) Afterwards, this program representation is optimized in a *middle-end* using many optimization passes.

(4) Finally, the program intermediate representation is emitted as the target HLL in a *back-end*. Currently, the C language and a Python-like language are used for this purpose and the decompiler supports decompilation of MIPS, ARM, and x86 executables. Both middle-end and back-end are built on the top of the LLVM Compiler System [38].

6. Context-Sensitive Description of Object File Formats

In the previous section, we described the current state of the OFF conversion tool used within the Lissom project. This plugin-based concept has been already implemented and it is used in practice. Its main drawback is the implementation complexity of each newly created plugin. Such plugin has to be written manually either on the top of some existing library or written entirely from scratch.

The complexity of existing parsing-libraries differs dramatically based on the target file format and supported features of library, see Table 1. For example, the BFD library supports multiple file formats (more than 50), but it contains more than half million code lines and its maintenance and extensibility is questionable. On the other hand, there exist lightweight libraries, like ELFIO, containing only few thousand code lines, but they lack any advanced functionality, such as processing of the parsed files.

Table 1. Complexity of several existing OFF parsing libraries.

Parsing library	Lines of code (LoC)
Binary File Descriptor library (BFD)	615,856
PeLib	12,220
LibELF	10,930
pyelftools	10,582
ELFIO	3,068

The existing plugins used within the Lissom project have different complexity too, see Table 2. The former three plugins in the table use the third party libraries; therefore, they are relatively small. On the other hand, the E32Image and Android DEX plugins are build from scratch and they are larger than the others.

Table 2. Complexity of Lissom project OFF-conversion plugins.

Conversion plugin	Lines of code (LoC)
WinPE	2,831
ELF	2,154
Mach-O	2,227
E32Image	9,419
Android DEX	10,582

According to our experience, the manual implementation of conversion plugins is slow (in matter of implementation) and prone to errors. This approach

is also complicated whenever implementing a non-common OFF (e.g. bFLT, XCOFF, OMF) because there are no suitable existing parsing libraries.

In order to achieve a true decompilation retargetability, we should apply the concept similar to one used for description of target processor architectures and automatic generation of the front-end part. This can be done in two steps. (1) Develop a specific language for OFF description. (2) Create an automatic generator of OFF-handling tools (i.e. OFF parsing and conversion) based on the description of the target OFF. Once this concept is adopted, the description complexity of the new OFFs should significantly decrease (i.e. the user will need to describe OFF using several hundred lines without using any external libraries).

In the rest of this section, we introduce the OFF description language. This language should be able to describe structure of each particular OFF as well as its context-sensitive aspects. We specify this language by using a grammar denoting this language and we add its brief description. At the end of this section, an example of ELF description is presented.

6.1. Grammar of the OFF Description Language

In programming language terminology, the grammars (see Section 2) are used for describing syntax of programming languages. In other words, a grammar creates a core of a programming-language parser. Such parser handles input programs (written in this language) using the grammar productions (rules). Parser can be used within compilers, verification software, or just for syntax checking. Within the classical compilation concept (see [1]), grammar serves only for description of syntax. The programming language semantics have to be described manually (e.g. HLL code realizing analysis of parsed code, semantic actions coupled with grammar productions).

In our case, we also use grammar to represent some kind of code — OFF structure. Each particular grammar description specifies one OFF; using this description, we are able to automatically generate the parser of this OFF. This parser will be used as a core of OFF converter to LOFF format. Moreover, our grammar is more advanced and it can also describe context-sensitive properties as well as semantic actions on the level of the grammar itself (this is another difference to existing OFF grammars described in Section 4 that are based on classical context-free grammars as defined in Definition 3). Formal definition and parsing of this grammar are described in the following section.

The language is designed for a description of the common OFFs (and hopefully the future ones, too). Executable or object file on parser's input are viewed as a binary stream. Its parsing is done via interconnected analyzers that invoke each other whenever it is necessary. Analyzers are also able to seek to the desired file offset within the stream. The language is not limited to any particular OFF construction, and it is capable to describe optional or scattered parts of the OFFs.

Modified Extended Backus-Naur Form (EBNF) is used for grammar's syntax description. Terminal symbols are typeset in **`boldface`**. Symbol $\sim$ is used for

concatenation. Sequences (i.e. zero or more repetitions) are denoted by {}; optional constructions (i.e. zero or one occurrence) are denoted by []; finally, selections (i.e. a choice between more constructions) are denoted by |. The grammar is depicted in Figure 5. For clarity, only the most important productions are specified.

```
start        -> root analyzer_def { parser_def } { production }
analyzer_def -> analyzer id ( [ offset [ , offset] ] ) { {
                statement ; } }
statement    -> element [ { semantic_actions } ]
             -> analyzer_id { [ times ] } [ { semantic_actions } ]
element      -> type id_attribute
             -> type [ value ]
analyzer_id  -> id_attribute ( [ offset [ , offset] ] )
type         -> ( int | uint ) ~ bitwidth_size { [ array_size ] }
attribute    -> [ < id { , id } ] > ]
production   -> ( id_attribute { , id_attribute } ) ->
                ( [ id'_attribute ] { , [ id'_attribute ] } ) [ priority ]
```

Fig. 5. Grammar of the OFF description language.

start is the start symbol of the grammar (see Definition 1). The keyword **root** denotes the starting analyzer, which is executed at the beginning of parsing. Each analyzer can be controlled by the begin and end offset. In that case, analyzer executes its job from the beginning offset and it must finish analysis before the stop offset, otherwise it will end as a parsing error. Analyzers read desired number of bits from an input stream, see Figure 6 for illustration.

The number of bits is specified by element with different sizes (specified by type). Elements are continual sequences of bits in an input stream. The value of an element can be skipped (i.e. so-called “don't care” value), enforced (i.e. analyzer ends with error if there is an unexpected value on input), or checked by analyzer, see Figure 7.

Elements and analyzers may contain a list of attributes. Attributes contain information about properties such as element's value or type. They can be used either in semantic actions (e.g. checking of element) or in context productions. In general, attributes are used for re-referencing previously parsed parts, such as information from OFF header. This context behavior is not common in classical programming language grammars. Therefore, it is possible to use both synthesized and inherited attributes from previously parsed elements within the semantic actions, see [1] for details.

Checking of elements is done either via semantic_actions, which are statements of the ANSI C code. Semantic actions can be used either for element checking as well as for interaction with retargetable tools. In our case, they are used mainly for direct LOFF generation. For illustration see Figure 8.

```
// Root (starting) parser of a particular file format XY
root analyzer XY_OFF_parser ()
{
    /* It invokes an analyzer of file-header. The header
       is located on the first 64 bytes. */
    header_parser(0, 512);
    // ...
}

// Parser of header - limited by offset range
analyzer header_parser (start_offset, end_offset)
{
    // ...
}

// Parser not limited by any offset range
analyzer another_parser ()
{
    // ...
}
```

Fig. 6. Example of analyzers definition.

```
analyzer header_parser (start_offset, end_offset)
{
    uint8 'X';  // Two magic bytes - enforced values
    uint8 'Y';
    int16;      // Don't care value (e.g. OFF version)
    // ...
}
```

Fig. 7. Example of statement types that are usable within analyzers.

Parsing can be also controlled via context `productions`; they are formatted as scattered context grammar productions (see Defintion 4); therefore, the number of items within brackets must be the same on both sides (ε-rules are allowed). The nonterminals $\texttt{id}_{\texttt{attribute}}$ stand for `element` or `analyzer_id` and they are rewritten according to the right-hand-side of those productions. Attributes are also taken into account during derivation. Finally, it is possible to describe `priority` of each production. A higher value means a higher priority. This is handy whenever we need to perform some actions before any other production (e.g. detecting a fault OFF structure as soon as possible). Details about parsing of these productions are described in the following section.

Finally, analyzers are interconnected via the `analyzer_call` statement. Analyzer can be invoked multiple times using `times`, this is useful for descrip-

```
analyzer header_parser (start_offset, end_offset)
{
    // ...
    int16 architecture <value>
    {
        if (architecture.value != 1)
        {   // Unsupported target architecture type
            parse_error();
        }
        else
        {   // C code producing a part of OFF conversion
            converter->setArchitectureType(architecture.value);
        }
    };
    // ...
}
```

Fig. 8. Usage of attributes and semantic actions within analyzers.

tion of repeating parts (e.g. table items). Analyzer invocation can also be done within the semantic actions by a call to the function with analyzer's name. Therefore, it is possible to conditionally invoke different analyzers based-on an actual context, see Figure 9.

```
root analyzer XY_OFF_parser ()
{
    // Invocation with offset range
    header_parser(0, 512);
    // Invocation without offset range
    another_parser();
    // Invocation 10 times
    another_parser() [10];
    // ...
}
```

Fig. 9. Example of different types of analyzer call.

6.2. Example of Usage

We can illustrate usage of the previously defined language on the 32-bit ELF format. A snippet of this description is depicted on Figure 10. The following description is used for its conversion to the Lissom LOFF format. At first, the

header is analyzed by invocation of `elf_header` analyzer. This analyzer starts at zero offset and analyzes all its elements and makes necessary checks.

It also converts basic information (e.g. entry point, endianness) to the LOFF format. The **`value`** attribute is used in several elements for referencing from other elements. At the end of the `elf_header` analyzer, we can see conditional invocation of section-header-table analyzer. It will be executed only if the table is present. We can also see that the analyzer `elf_sht` is invoked together with specification of its beginning and ending offset gathered from previous attributes. This corresponds to the structure depicted in Figure 1.

The last construction depicted on this example is a context production. It controls that executable files do not contain static relocations (e.g. static relocation `R_386_PC32`). It is marked with priority higher than other productions; therefore, it will be checked at first. Whenever the preliminary part is satisfied (e.g. executable file is not properly linked), it blocks parsing by nonterminal `error`, which leads to parsing error.

7. Context Parsing

In this section, we present a concept of the context parser that can be used for parsing the previously described OFF language. The major difference to other existing languages is its support of describing context-sensitive relations. However, parsing of these constructions is non-trivial because there is no suitable formalism capable of describing such grammar in present.

The idea of context parser is not entirely new and we can find several attempts to create a parser for context-sensitive language (Definition 2) in past, see [37,35,1]. These attempts were only partially successful. They were either focused on a very specific aspects of some domain-specific language, or they were not based on formal models; therefore, it was hard to prove such concepts.

Today's traditional techniques perform context analysis via semantic actions written in the host language accompanying usually context-free grammar of a suitable form (see [1]). The other possibility is to use some context-free parser based on any available technique and then to perform analysis of a data structure created as an output of the parser (usually some tree-like structure or some kind of byte-code [30]).

A mixture of several descriptive means (grammar together with host language or another combination) bound by explicit data structures stored in trees, attributes, code, or their mixture is not suitable if an analyzer is to be described formally. Moreover, a change to the input language syntax usually dramatically affects other parts of a parser.

Therefore, in this section, we define two new formal models that are based on scattered context grammars—*attributed scattered context grammars* and *attributed scattered context grammars with priority function*. These grammars can be effectively used for formal description of context-sensitive relations in a particular language. Furthermore, we modify the existing regulated pushdown au-

```
root analyzer ELF32 () {
    elf_header(0) { check_header(); }; // parse ELF header
}
analyzer elf_header (start_offset) {
    uint8 [16] e_ident { /* Check of the "ELF Identification"
                            field */ };
    uint16 e_type <value> {
        if (e_type.value > 4)
            parse_error(); // Unsupported ELF file type
    };
    uint16;             // e_machine - a don't care value
    uint32 1;           // e_version needs to be '1'
    uint32 e_entry <value> { // Direct generation of LOFF
        LOFF->setEntryPoint(e_entry.value);
    };
    // ...
                        // Section header table's offset (SHT)
    uint32 e_shoff <value>;
    // ...
    // Size of entrie in SHT and number of elements in SHT
    uint16 e_shentsize;
    uint32 e_shnum <value> {
        if (e_shoff.value != 0) // Analyze SHT
            elf_sht(e_shoff,
                    e_shoff + e_shnum.value * e_shentsize);
    };
    // ...
}
analyzer elf_sht (start_offset, end_offset) {
    // ...
                        // Analysis of Section Header Table
}
analyzer elf_section (start_offset, end_offset) {
    // ...
                        // Analysis of each particular section
}

// Productions describing context behavior
// Simplified control of appearance of static relocations
// within executable files
(elf_header<is_executable>, elf_relocation<is_static>}) ->
    (error, error) [999] // High-priority production
// Other productions
```

Fig. 10. A code snippet of an ELF description using OFF language.

tomata (see [17]) for parsing these grammars. Finally, we give a brief overview of a context parser construction.

7.1. Attributed Scattered Context Grammars

In this subsection, we define two new formalisms that are based on scattered context grammars. We assume a reader is familiar with the attributed grammars (for further details see [36,30,1].

Definition 10. A *voidy n-tuple* over domain D is the tuple

$$< d_1, \dots, d_n > \in D^n,$$

where D^n stands for $D_1 \times D_2 \times \dots \times D_n$ and $n \in \mathbb{N}$; if $n = 0$ then the tuple is void and we write $<>$ or simply we do not write anything if it is clear from the context.

Definition 11. *Variable voidy Cartesian product* $\cup D$ over domain D is defined as

$$\cup D = \cup_{i=0}^{n} D^i,$$

where D^n stands for $D_1 \times D_2 \times \dots \times D_n$ and $n \in \mathbb{N}$; $D^0 = \{<>\}$.

Definition 12. An *attributed scattered context grammar* (aSCG) is a seven-tuple,

$$G = (V, T, P, S, D, R, \rho),$$

where

- V is a total alphabet;
- $T \subset V$ is a finite set of terminals;
- $S \in V - T$ is the start symbol;
- P is a finite set of productions of the form

$$(A^1_{w_1}, \dots, A^n_{w_n}) \rightarrow (x^1_{@}, \dots, x^n_{@}),$$

  where $A^i \in V - T$, $w_i = \rho(A^i)$, $x^i_{@} \in V^*$ for all $i : 1 \leq i \leq n$ and all symbols in $x^i_{@}$ have their corresponding voidy tuple of attributes;
- D is the domain of attributes;
- R is the naming of attributes representing any value from D;
- ρ is a mapping $\rho : V \rightarrow \cup R$, where $\cup R$ is the variable voidy Cartesian product.

Definition 13. An *attributed propagating scattered context grammar* (aPSCG) is an aSCG

$$G = (V, T, P, S, D, R, \rho),$$

in which every $(A^1_{w_1}, \dots, A^n_{w_n}) \rightarrow (x^1_{@}, \dots, x^n_{@}) \in P$ satisfies $x^i_{@} \in V^+$ for all $i : 1 \leq i \leq n$.

Notation of attribute use is the following: we write

$$A_{<a_1,\dots,a_n>} \text{ if } \rho(A) = < a_1, \dots, a_n >$$

for any n, or simply A_w if attribute names are not in our focus; we write $A_{<>}$ if we want to stress that void attribute tuple is assigned to the symbol, or we write just A for the sake of simplicity. If there is a string of symbols $x = A_1 \dots A_n$ and for every $A_i, i \in \{1 \dots n\}$ there is w_i such that $\rho(A_i) = w_i$ we write $x_@$ to stress that every symbol of x has its voidy tuple of attributes.

Definition 14. Let $G = (V, T, P, S, D, R, \rho)$ be a (propagating) aSCG. If

$$y = u_1 A^1_{w_1} u_2 \dots u_n A^n_{w_n} u_{n+1},$$
$$z = u_1 x^1_@ u_2 \dots u_n x^n_@ u_{n+1},$$

and $y, z \in V^*$, $p = (A^1_{w_1}, \dots, A^n_{w_n}) \to (x^1_@, \dots, x^n_@) \in P$, and if every attribute occurring in $A^1_{w_1}, \dots, A^n_{w_n}$ and $x^1_@, \dots, x^n_@$ has a value from D defined and every occurrence of some attribute $a \in R$ in $A^1_{w_1}, \dots, A^n_{w_n}$ and $x^1_@, \dots, x^n_@$ carries the same value from D then y directly derives z in the (propagating) aSCG G according to the production p,

$$y \Rightarrow_G z\ [p] \text{ (or simply } y \Rightarrow_G z\text{)}.$$

A language generated by (propagating) aSCG is defined the same way as for (propagating) SCG. Similarly, family of (propagating) attributed scattered context languages is defined as $\mathcal{L}$(a(P)SC).

To give a light insight and motivation on usage of attributed grammar, we present a small example. Let us take into account the language $a^n b^n c^n$ for $n \geq 1$. This is truly a context-sensitive language (see [28]). Using SCG[1], we can describe the language by grammar:

$$G_1 = (\{S, X, C, a, b, c\}, \{a, b, c\}, P, S),$$

with P containing

$$\begin{aligned} P = \{ \quad & (S) \to (XC), & [p_1] \\ & (X, C) \to (aXb, cC), & [p_2] \\ & (X, C) \to (ab, c)\} & [p_3] \end{aligned}$$

As an example of derivation by using this grammar

$$\begin{aligned} S &\Rightarrow XC & [p_1] \\ &\Rightarrow aXbcC & [p_2] \\ &\Rightarrow aaXbbccC & [p_2] \\ &\Rightarrow aaabbbccc & [p_3] \end{aligned}$$

From a formal point of view, the presented grammar represents a perfect description of a given language. From a practical point of view, this kind of description is too specific. Let us assume a modification of this language: $z^n u^n v^n$ for $n \geq 1$. This language has the same structure as the previous one except the

[1] This grammar is actually a propagating SCG because it does not contain any erasing rules.

different terminal names. However, this small difference means that the original grammar has to be significantly rewritten.

In particular, terminals a, b, and c are too specific in the original grammar. In a fact, each of them can be considered as an identifier, which was named e.g. $'a'$. On the other hand, such an identifier is bound to some particular value ($'a'$) that can be described by a value of attribute bound to particular (otherwise anonymous) terminal.

Thus, we introduce attributes to keep fully formal view and obtaining expressive power and variability. Therefore, we define the attributed SCG

$$G_2 = (V, T, P, S, D, R, \rho).$$

Now we can modify our example in such a way: we add a domain of attributes D of all textual strings (string is written in quotes, e.g. $'z'$), we add a naming $R = \{q, w, e\}$, we define mapping ρ such a way, so that:

$$\begin{aligned} \rho(S) &= <> \\ \rho(X) &= < q, w > \\ \rho(C) &= < e > \\ \rho(a) &= < q > \\ \rho(b) &= < w > \\ \rho(c) &= < e > \end{aligned}$$

and present an aSCG grammar productions (as a modification of the previous SCG):

$$\begin{aligned} (S) &\rightarrow (X_{<'a','b'>} C_{<'c'>}) \\ (X_{<q,w>}, C_{<e>}) &\rightarrow (a_{<q>} X_{<q,w>} b_{<w>}, c_{<e>} C_{<e>}) \\ (X_{<q,w>}, C_{<e>}) &\rightarrow (a_{<q>} b_{<w>}, c_{<e>}) \end{aligned}$$

A modification of the presented aSCG allows to change terminals with redefinition of just a single grammar production, in particular, attribute values, as the production remains the same, as such. To get the second mentioned language, we have to change just the first production to $(S) \rightarrow (X_{<'z','u'>} C_{<'v'>})$ and the rest remains the same.

To extend expressive power and bring necessary pragmatic features for practical exploitation of a(P)SCG in context analysis/parsing, we extend a(P)SCG to priority attributed scattered context grammars.

Definition 15. A *(propagating) attributed scattered context grammar with priority*
(a(P)SCGP) is an eight-tuple

$$G = (V, T, P, S, D, R, \rho, \pi),$$

where (V, T, P, S, D, R, ρ) is a (propagating) attributed scattered context grammar and π is a *function*

$$\pi : P \rightarrow \mathbb{N}.$$

Definition 16. Let $G = (V, T, P, S, D, R, \rho, \pi)$ be an a(P)SCGP. We say that y directly derives z in a(P)SCG G according to the production p, $y \Rightarrow_G z\ [p]$ (or simply $y \Rightarrow_G z$), if and only if:

- $y = u_1 A^1_{w_1} u_2 \dots u_n\ A^n_{w_n} u_{n+1} \in V^*$,
- $z = u_1 x^1_{@} u_2 \dots u_n x^n_{@} u_{n+1} \in V^*$,
- $p = (A^1_{w_1}, \dots, A^n_{w_n}) \rightarrow (x^1_{@}, \dots, x^n_{@}) \in P$,
- there is no $p' = (A'^1_{w_1}, \dots, A'^n_{w_n}) \rightarrow (x'^1_{@}, \dots, x'^n_{@}) \in P$, such that:
  1. $y = u'_1 A'^1_{w_1} u'_2 \dots u'_n A'^n_{w_n} u'_{n+1} \in V^*$, and
  2. $\pi(p') > \pi(p)$;
- and conditions of Definition 14 for attributes must hold.

Language generated by a(P)SCGP is defined similarly as for a(P)SCG.

To give an order of rules when several options could be used, we use priority. For demonstration, we define the attributed SCG with priority

$$G_3 = (V, T, P, S, D, R, \rho, \pi).$$

The grammar is the same as G_2 up to priority mapping, which is defined as:

$$\begin{aligned} \pi((S) \rightarrow (X_{<'a','b'>} C_{<'c'>})) &= 1 \\ \pi((X_{<q,w>}, C_{<e>}) \rightarrow (a_{<q>} X_{<q,w>} b_{<w>}, c_{<e>} C_{<e>})) &= 1 \\ \pi((X_{<q,w>}, C_{<e>}) \rightarrow (a_{<q>} b_{<w>}, c_{<e>})) &= 1 \end{aligned}$$

Then, the sentence $a_{<'a'>} a_{<'a'>} b_{<'b'>} b_{<'b'>} c_{<'c'>} c_{<'c'>}$ is obtained by the following derivation:

$$\begin{aligned} S &\Rightarrow X_{<'a','b'>} C_{<'c'>} && [p_1] \\ &\Rightarrow a_{<'a'>} X_{<'a','b'>} b_{<'b'>} c_{<'c'>} C_{<'c'>} && [p_2] \\ &\Rightarrow a_{<'a'>} a_{<'a'>} b_{<'b'>} b_{<'b'>} c_{<'c'>} c_{<'c'>} && [p_3] \end{aligned}$$

that represents the string $aabbcc$.

7.2. Regulated Pushdown Automata

As has been illustrated above, the a(P)SCGP can be easily used for description of context-sensitive languages. However, we still need a formal model for parsing such description. For this reason, we use a *Regulated Pushdown Automata*.

In [15], it is presented, how regulated pushdown automata can be exploited for building context parsers derived from scattered context grammars of particular features. Basic concept of regulated pushdown automata can be found in [17,27] — papers especially present definition and expressive power of various versions of regulated pushdown automata.

Consider a pushdown automaton (PDA)

$$M = (Q, \Sigma, \Omega, R, s, S, F),$$

where

- Q is a *finite set of states*;
- Σ is an *input alphabet*;
- Ω is a *pushdown alphabet*;
- R is a *set of productions* of the form

$$Apa \rightarrow wqb,$$

  where $A \in \Omega$, $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Omega^*$ and $b \in \{a, \varepsilon\}$ (if $b \neq \varepsilon$ then the production "tests" the value under the reading head, the head is not shifted, the symbol is not read);
- $s \in Q$ is the *start state*;
- $S \in \Omega$ is the *start symbol*;
- $F \subseteq Q$ is a *set of final states*.
- Without a loss of generality, we require that Q, Σ, and Ω are pairwise disjoint.

Now, consider a control language, Ξ (formally defined below), over M's productions. Informally, with Ξ, M accepts a word, x, if and only if Ξ contains a control word according to which M makes a sequence of moves so it reaches a final configuration after reading x.

Let Ψ be an alphabet of *production labels* such that $card(\Psi) = card(R)$, and ψ be a bijection from R to Ψ. For simplicity, to express that ψ maps a production, $Apa \rightarrow wq \in R$, to ρ, where $\rho \in \Psi$, this paper writes $\rho.Apa \rightarrow wq \in R$; in other words, $\rho.Apa \rightarrow wq$ means $\psi(Apa \rightarrow wq) = \rho$.

Definition 17. A *configuration* of M, χ, is any word from $\Omega^* Q \Sigma^*$. For every $x \in \Omega^*$, $y \in \Sigma^*$, and $\rho.Apa \rightarrow wq \in R$, M makes a move from configuration $xApay$ to configuration $xwqy$ according to ρ, written as $xApay \vdash xwqy\ [\rho]$.

Let χ be any configuration of M. M makes *zero moves from χ to χ according to ε*, symbolically written as $\chi \vdash^0 \chi\ [\varepsilon]$. Let there exist a sequence of configurations $\chi_0, \chi_1, \ldots, \chi_n$ for some $n \geq 1$ such that $\chi_{i-1} \vdash \chi_i\ [\rho_i]$, where $\rho_i \in \Psi$, for $i = 1, \ldots, n$, then M makes *n moves from χ_0 to χ_n according to $\rho_1 \ldots \rho_n$*, symbolically written as $\chi_0 \vdash^n \chi_n\ [\rho_1 \ldots \rho_n]$.

Definition 18. Let Ξ be a *control language* over Ψ; that is, $\Xi \subseteq \Psi^*$. With Ξ, M defines the following three types of accepted languages:

$L(M, \Xi, 1)$—*the language accepted by final state*
$L(M, \Xi, 2)$—*the language accepted by empty pushdown*
$L(M, \Xi, 3)$—*the language accepted by final state and empty pushdown*

defined as follows. Let $\chi \in \Omega^* Q \Sigma^*$. If $\chi \in \Omega^* F$, $\chi \in Q$, $\chi \in F$, then χ is a *1-final configuration*, *2-final configuration*, *3-final configuration*, respectively. For $i = 1, 2, 3$, define $L(M, \Xi, i)$ as $L(M, \Xi, i) = \{w \mid w \in \Sigma^*,$ and $Ssw \Rightarrow^* \chi\ [\sigma]$ in M for an i*-final configuration*, $\chi,$ and $\sigma \in \Xi\}$.

Definition 19. *Regulated pushdown automata* (RPDA). For any family of languages, X, set $RPDA(X, i) = \{L \mid L = L(M, \Xi, i)$, where M is a PDA and $\Xi \in X$, where $i = 1, 2, 3\}$.

Namely, pushdown automata regulated by linear languages have the same power as Turing machine

$$RE = RPDA(LIN, 1) = RPDA(LIN, 2) = RPDA(LIN, 3),$$

where RE stands for the set of all recursively enumerable languages and LIN stands for the set of all linear languages [26] — proof can be found in [17].

Thus, such automata are powerful enough for analysis of context languages. Nevertheless, we need a deterministic version of such automata. Their detailed description and a way, how the automaton can be built from a SCG of certain features, can be found in [15].

Definition 20. Let $M = (Q, \Sigma, \Omega, R, s, S, F)$ be a regulated pushdown automaton, with set of labels Ψ, bijection ψ from labels Ψ to productions R, and with control language Ξ. Such an RPDA is *deterministic* (DRPDA) if being in a state q, $q \in Q$, the appropriate action, which should be performed, can always be deterministically selected. This can only be due to the following two circumstances:

(1) For the given state, there is only one production $r \in R$ that is applicable in a given situation (state, symbol on the top of the pushdown or under the reading head) and, moreover, control language admits such a production.

(2) If there are more than one productions that are applicable in a given situation then the production can be deterministically denoted according to the actual context of sentential form of the control language applicable to the current state of operation performed by RPDA.

To give a rough idea from another viewpoint: in a center, there is non-deterministic pushdown automaton; all of its operations are encoded as a symbols of the control-language alphabet; successful operation of the PDA must be verified by the control language, which means that operation of PDA produces a string of symbols (step-by-step operations of the automaton are encoded to string of symbols) and if the string is a sentence of the control language then the operation of regulated PDA is successful; if PDA fails during its operation or the produced string is not in the control language then it means that analyzed input is not accepted.

7.3. Context Parser Construction

Relation between automata presented above and implementation is quite simple. We can build appropriate automaton from a given grammar (see [14,15,18]) automatically. Moreover, usage of Haskell programming language enables to build a kind of domain specific language on the top of Haskell. Thus, it is necessary to define the wanted grammar inside Haskell using supporting predefined constructs and the parser is done. Lexical analysis is done in the same way as in any other parser (i.e. definition of lexemes and their transformation to tokens, see [1]). Also manipulation of the output of the parser is done in a traditional way. The key feature is that just a simple modification of the grammar allows

to dramatically modify the parsed input. Thus, any change is much faster then using any other technique.

Furthermore, we can apply the same principles as in SCG parsing (see [15]):

– regulated PDA can be made deterministic;
– having SCG of suitable features (LL SCG, see [15]), we can algorithmically derive a deterministic regulated pushdown automaton, which accepts (decides) the language generated by the SCG.

To achieve full flexibility and big expressive power, so that changes in a language can be efficiently handled on the grammar level, we need to introduce attributes and priorities to LL SCG parsers.

Attributes Introduction of attributes is not difficult at all — grammar (omitting attributes) must satisfy the same conditions as LL SCG grammars, plus the following one — $\forall(A^1_{w_1}, \ldots, A^n_{w_n}) \to (x^1_@, \ldots, x^n_@) \in P$ it must hold:

– $\rho(A^1) =<>$ and
– let $x^1_@ = X^1_{v_1} \ldots X^m_{v_m}$ then $\forall X_i \in (V - T) : \rho(X_i) =<>, i \in \{1 \ldots m\}$

Priorities Fortunately, priorities are not a problem of construction parsing tables and automaton as such. They are problem of saving automaton configuration and its restoration — from a formal point of view.

The situation is such, priorities can cause that we have several grammar productions for expansion for the same automaton configuration (symbol under the reading head and top of the pushdown) — we say the productions are overlapping. In the traditional notion of deterministic PDA it means a conflict and no automaton can be built.

If we have priorities for grammar productions introduced then this situation is conflict if two or more such overlapping productions have the same priority assigned.

If the priorities for overlapping productions are different then we have to order such set of productions from the highest priority to the lowest one. When the automaton configuration gets to the point when some of these productions could be applied, the production with the highest priority is applied at first. If it fails then the original configuration is restored and the next production is applied and so on and so on, until some production succeeds. If none of the productions succeeds then the analysis fails with an error.

The problem is about storing the configuration and restoration, especially, how we can recognize that some production expansion fails. Fortunately, as it can be seen in [14,15,18], during expansion, when we search for suitable non-terminals on the pushdown, we use the control language to save the content of the pushdown that is popped out of the pushdown. Thus, when we reach bottom of the pushdown it means that the production cannot be expanded in the situation, so that we should apply another one. In such a situation the content of the pushdown is saved in the context of the control language and we can

restore it to its original content before trying to expand the production. It is used the same technique with a small difference, when right hand sides of the so far expanded part are not pushed to the pushdown, but the original content is.

7.4. Experimental Results

In present, the context parser of the OFF language is in the prototype phase. Therefore, we are unable to give any experimental results in a deeper detail yet. On the other hand, the concept of the parser can be described using the simple context-sensitive language.

Performance measurement of our approach is not easy. The reason is that there is no context parser based on grammar input available. General approaches are well known to be inefficient. Thus, it was quite difficult to find simple use case for comparison.

Our implementation language is Haskell due to ease of use for our purposes. Re-implementing our parser in C/C++ would be time consuming, so we decided to implement competitive parser of some suitable language in Haskell directly, without using our grammar based context parser.

We have decided to use parser of the aforementioned language $a^n b^n c^n$ based on the presented grammar, but without any attributes and priority — firstly, they are not necessary for such a simple case; secondly, it would be quite complicated to implement something similar in the other program for comparison.

The comparison is unfair for the SCG-based solution, though. We compare parser based on complex SCG with straightforward ”C-like” implementation of analysis of the language $a^n b^n c^n$. There are several reasons, why it is unfair:

– The grammar based parser uses stack to create contextual information and its consumption is proportional to input size.
– The ”C-like” implementation is very much Haskell syntax of C approach, on the other hand the grammar based parser is very much of the Haskell.
– ”C-like” implementation is constant space so it provokes for better performance.

In other words, we compared something incomparable, in a fact. The comparison of speed is depicted in Figure 11. The tests were limited on size due to stack utilization and application size limitation in Windows 32-bit application.

Surprisingly, the time complexity according to input size is almost the same. Thus, we can state that our approach is not only very efficient in change incorporation both on user and implementation side, but is is even quite efficient from the evaluation speed viewpoint.

The evaluation of this concept on a more complex examples (such as the OFF language) is marked as our future research but unavailable yet.

8. Conclusion and Future Work

This paper was focused on handling of OFFs and its usage in retargetable tools. Several existing solutions were presented, and their limitations were discussed.

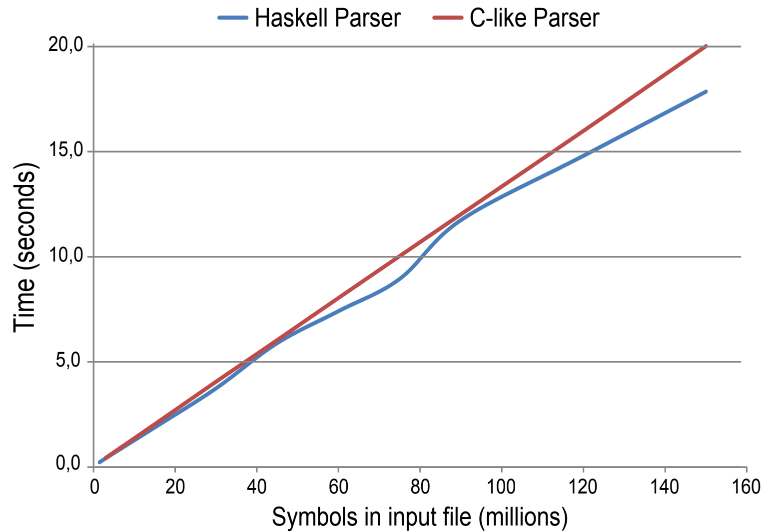

Fig. 11. Speed comparison between two parsers of the $a^n b^n c^n$ language.

The main contribution of this paper is a presentation of the language for OFF description and concept of its parser. The major advantage of this concept is its ability to describe and parse context-sensitive properties. The parser is based on the formal models that were designed for this purpose.

A prototype of this context parser is under development. The Haskell programming language is used for this purpose because it is well-suited for our needs (lazy evaluation [13], type inference, etc.). According to the preliminary experimental results, which were focused on simple languages like $a^n b^n c^n$, this approach is faster than other parsers of the same language.

The language can be used for OFF parsing and manipulation. Its main usage is within an existing retargetable decompiler, where it will be used for conversion from platform-dependent OFFs into an internal COFF-based file format. However, this is not a limitation because the language can be used in other retargetable tools, such as disassemblers, loaders, or debuggers.

In the future research, we would like to use the context parser in other areas. For example it can be used for natural language processing, description of other binary file formats (i.e. not just OFF), or parsing of HLL programming languages, such C, where it will be able to automatically check consistency of declaration, definition, and usage of variables, see [28,41,37] for details.

Acknowledgments. This work was supported by the project TA ČR TA01010667 System for Support of Platform Independent Malware Analysis in Executable Files, BUT grant FEKT/FIT-J-13-2000 Validation of Executable Code for Industrial Automation Devices using Decompilation, BUT FIT grant FIT-S-11-2, by the project CEZ MSM0021630528

Security-Oriented Research in Information Technology, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Boston, 2nd edn. (2006)
2. Apple Inc.: Macintosh application environment. http://www.mae.apple.com (1994)
3. Apple Inc.: Mac OS X ABI Mach-O file format reference (2009)
4. Chamberlain, S.: The Binary File Descriptor Library. Iuniverse Inc. (2000)
5. Cifuentes, C.: Reverse Compilation Techniques. Ph.D. thesis, School of Computing Science, Queensland University of Technology, Brisbane, QLD, AU (1994)
6. Digital: Freeport express. http://www.novalink.com/freeport-express (1996)
7. Emmerik, M.J.V.: Static Single Assignment for Decompilation. Ph.D. thesis, University of Queensland, Brisbane, QLD, AU (2007)
8. Faase, F.: BFF: A grammar for binary file formats. http://www.iwriteiam.nl/Ha_BFF.html (2012)
9. Fernández, M.F.: Simple and effective link-time optimization of Modula-3 programs. SIGPLAN Not. 30(6), 103–115 (1995)
10. Gircys, G.R.: Understanding and Using COFF. O'Reilly & Associates, Inc., Sebastopol, US-CA (1988)
11. Greibach, S., Hopcroft, J.: Scattered context grammars. Journal of Computer and System Sciences 3(3), 233–247 (1969)
12. Hohensee, P., Myszewski, M., Reese, D.: Wabi cpu emulation. Hot Chips VIII (1996)
13. Jirák, O., Kolář, D.: Derivation in scattered context grammar via lazy function evaluation. In: 5th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09). OpenAccess Series in Informatics (OASIcs), vol. 13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, DE (2009)
14. Kolář, D.: Pushdown Automata: Another Extensions and Transformations. Habilitation thesis, Brno University of Technology, Faculty of Information Technology (2005)
15. Kolář, D.: Scattered context grammars parsers. In: 14th International Congress of Cybernetics and Systems of WOCS. pp. 491–500. Wroclaw University of Technology, PL, Wroclaw, PL (2008)
16. Kolář, D., Husár, A.: Output object file format for assembler and linker. Internal document, Brno University of Technology, Faculty of Information Technology, Brno, CZ (2012)
17. Kolář, D., Meduna, A.: Regulated pushdown automata. Acta Cybernetica 2000(4), 653–664 (2000)
18. Kolář, D., Meduna, A.: Regulated automata: From theory towards applications. In: 8th International Conference on Information Systems Implementation and Modelling (ISIM'05). pp. 34–48. MARQ, Ostrava, CZ (2005)
19. Křoustek, J., Matula, P., Ďurfina, L.: Generic plugin-based convertor of executable file formats and its usage in retargetable decompilation. In: 6th International Scientific and Technical Conference (CSIT'11). pp. 127–130. Ministry of Education, Science, Youth and Sports of Ukraine, Lviv Polytechnic National University, Institute of Computer Science and Information Technologies (2011)

20. Křoustek, J., Přikryl, Z., Kolář, D., Hruška, T.: Retargetable multi-level debugging in HW/SW codesign. In: 23rd International Conference on Microelectronics (ICM'11). p. 6. Institute of Electrical and Electronics Engineers (2011)
21. Křoustek, J., Židek, S., Kolář, D., Meduna, A.: Scattered context grammars with priority. International Journal of Advanced Research in Computer Science (IJARCS) 2(4), 1–6 (2011)
22. Levine, J.R.: Linkers and Loaders. Operating Systems, Morgan Kaufmann Publishers (2000)
23. Lissom: http://www.fit.vutbr.cz/research/groups/lissom/ (2013)
24. LLVM Assembly Language Reference Manual: http://llvm.org/docs/LangRef.html (2013)
25. Masařík, K.: System for Hardware-Software Co-Design. VUTIUM, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 1st edn. (2008)
26. Meduna, A.: Automata and Languages: Theory and Applications. Springer-Verlag, London, GB (2005)
27. Meduna, A., Kolář, D.: One-turn regulated pushdown automata and their reduction. Fundamenta Informaticae 2001, 1001–1007 (2001)
28. Meduna, A., Techet, J.: Scattered Context Grammars and their Applications. WIT Press, Southampton, GB (2010)
29. Microsoft Corporation: Microsoft portable executable and common object file format specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx (2013), version 8.3
30. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, US-CA (1997)
31. Nokia: E32Image. http://www.developer.nokia.com/Community/Wiki/E32Image (2012)
32. Ramsey, N., Fernández, M.: Specifying representations of machine instructions. ACM Transactions on Programming Languages and Systems 19(3), 492–524 (1997)
33. Ramsey, N., Fernandez, M.F.: The New Jersey Machine-Code Toolkit. In: USENIX Technical Conference. pp. 289–302 (1995)
34. Ramsey, N., Hanson, D.R.: A retargetable debugger. Tech. rep., Princeton University, Princeton, US-NJ (1992)
35. Roberts, D.M.: Earley parsing for context-sensitive grammars. http://danielmattosroberts.com/earley/context-sensitive-earley.pdf (2009)
36. Rodriguez-Cerezo, D., Cabezuelo, A.S., Sierra, J.L.: A systematic approach to the implementation of attribute grammars with conventional compiler construction tools. Computer Science and Information Systems (ComSIS) 9(3), 983–1017 (2012)
37. Rychnovský, L.: Parsing of context-sensitive languages. In: 2nd International Workshop on Formal Models (WFM'07). pp. 219–226. Opava, CZ (2007)
38. The LLVM Compiler Infrastructure: http://llvm.org/ (2013)
39. TIS Committee: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (1995), http://refspecs.freestandards.org/elf/elf.pdf
40. Troshina, K., Chernov, A., Derevenets, Y.: C decompilation: Is it possible? In: International Workshop on Program Understanding (IWPU'09). pp. 18–27 (2009)
41. Ung, D., Cifuentes, C.: SRL - a simple retargetable loader. In: The Australia Software Engineering Conference. pp. 60–69. IEEE Computer Society (1997)
42. UQBT - A Resourceable and Retargetable Binary Translator: http://itee.uq.edu.au/~cristina/uqbt.html (2012)

43. Ďurfina, L., Křoustek, J., Zemek, P., Kábele, B.: Detection and recovery of functions and their arguments in a retargetable decompiler. In: 19th Working Conference on Reverse Engineering (WCRE'12). pp. 51–60. IEEE Computer Society, Kingston, ON, CA (2012)
44. Ďurfina, L., Křoustek, J., Zemek, P., Kolář, D., Hruška, T., Masařík, K., Meduna, A.: Design of a retargetable decompiler for a static platform-independent malware analysis. In: 5th International Conference on Information Security and Assurance (ISA'11). Communications in Computer and Information Science, vol. 200, pp. 72–86. Springer-Verlag, Berlin, Heidelberg, DE (2011)
45. Ďurfina, L., Křoustek, J., Zemek, P., Kolář, D., Hruška, T., Masařík, K., Meduna, A.: Design of a retargetable decompiler for a static platform-independent malware analysis. International Journal of Security and Its Applications (IJSIA) 5(4), 91–106 (2011)

Jakub Křoustek is a Ph.D. student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received the MSc. degree from the same university in 2009. He is currently working on the Lissom research project as the leader of the generic decompiler and debugger development team. His current research interests include the reverse engineering, malware detection, and compiler design, with special focus on the code analysis and reverse translation.

Dušan Kolář went to Brno University of Technology, Czech Republic, where he studied computer science and cybernetics and obtained his degrees in 1994 and 1998. Since then, he has been working at the university, presently at the Faculty of Information Technology. His main research interests are formal languages and automata and formal models with focus on their exploitation in compilers and formal models transformation.

Received: January 20, 2013; Accepted: September 2, 2013.