CLOPS: A DSL for Command Line Options

Mikoláš Janota, Fintan Fairmichael, Viliam Holub,
Radu Grigore, Dermot Cochran, and Joseph R. Kiniry

School of Computer Science and Informatics
and
The Complex and Adaptive Systems Laboratory (CASL)
University College Dublin, Ireland

Abstract. Programmers often write custom parsers for the command line input of their programs. They do so, in part, because they believe that both their program's parameterization and their option formats are simple. But as the program evolves, so does the parameterization and the available options. Gradually, option parsing, data structure complexity, and maintenance of related program documentation becomes unwieldy. This article introduces a novel DSL called CLOPS that lets a programmer specify command line options and their complex inter-dependencies in a declarative fashion. The DSL is supported by a tool that generates the following features to support command line option processing: (1) data structures to represent option values, (2) a command line parser that performs validity checks, and (3) command line documentation. We have exercised CLOPS by specifying the options of a small set of programs like `ls`, `gzip`, and `svn` which have complex command line interfaces. These examples are provided with the Open Source release of the CLOPS system.

1 Introduction

A boiling frog is a well known phenomenon in software development. Just as a frog meets its fate in water that is slowly heated, so do software projects become unmanageable as their complexity gradually increases.

Processing of command line options is not an exception. The first few versions of a program have small number of options that are easy to process and do not have many dependencies between them.

No wonder then, that many programmers, with famous last words "this is easy" on their lips, opt for custom code for option processing. As the number of the program's options and their inter-dependencies grows, option processing code gets bloated and diverges from its documentation.

This article suggests an alternative by providing a DSL that enables programmers to explicitly, declaratively capture options and their inter-dependencies. Furthermore, a tool is provided that produces documentation, code for command line processing, and data structures. The data structures produced represent the values of the options and are automatically populated during parsing.

The observations described above shaped the language proposed in this article. On the one hand, simple option sets must be easy to write when the program is in its early stages. But on the other hand, a set of options must be easy to evolve and even complicated option sets must be supported.

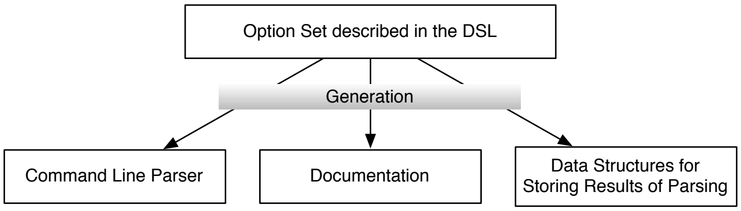
Option Set described in the DSL
Generation
Command Line Parser
Documentation
Data Structures for Storing Results of Parsing

Fig. 1. Assets generated by the CLOPS system.

This article contains several core contributions:

- We identify the main concepts seen in command line processing and describe their interdependencies (Section 4).
- We propose a novel DSL called CLOPS for describing command line option interfaces. Its syntax and semantics are described in Section 5.
- We provide a tool that reads in a description in this language and generates multiple artifacts (Figure 1):
  - a Java implementation of the command line parser,
  - documentation, and
  - data structures that are used to store the result of parsing.

  Section 6 briefly discusses the implementation and explains how new constructs are added to the DSL.
- We demonstrate the utility of the language on a set of examples (Section 6.3).

Section 8 is an overview of other approaches tackling command line processing and Section 9 provides pointers for further research and improvements. The following section sets up the ground for further discussion.

2 Background

Command line utilities are widespread in the software community and hence there is no need to go into great detail about them. Nevertheless, several properties of command line options that will be important for later discussion must be reviewed.

A *program* is run from a command line with a list of *options* that affect its behavior. For instance, the program `ls` lists files in a multi-column format if run with the option `-C` and lists files one-per-line if run with the `-1` option.

This means that the invoked program has to process the options provided on the command line and store the result of that analysis into its state in some fashion.

Obviously, the behaviors of these particular options, `-C` and `-1`, cannot be put into effect at the same time. We say that there is a *dependency* between these two options.

As a consequence, `ls -C -1` results in the same behavior as if only `-1` was provided and vice versa, according to the principle *last one wins*. For such dependency we say that the options *override* one another. There are several kinds of dependencies with varying resolution strategies.

Another common pattern seen in command line arguments is option *arguments*. To give an example, the program `tail` prints the last x lines of a given input and is typically invoked with `tail -n x filename`. This means that option processing needs to make sure that `x` is indeed a number, and produce an error otherwise, and convert its textual representation into a numeric one. Often, an argument has a *default value*. If `tail` is invoked without the `-n` option, for instance, `x` is assumed to have value 10.

The set of options supported by a certain program represent its (textual) user interface. Hence, it is important for a program's options to be well-documented as a program's users cannot be expected to read its source code. In UNIX-like systems documentation is maintained in the database of *man(ual) pages*. As stated before, one of the objectives of this work is to keep a program's documentation consistent with its behavior.

3 CLOPS Exposure

Let us imagine we are writing a program which encrypts a given file. As we want to avoid problems with weirdly-named files (such as starting with the hyphen), we introduce one-argument options `-i filename` and `-o filename` to enable the user to specify the input and the output file, respectively.

```
NAME:: encrypt
ARGS::
  input:{"-i"}:{ file }:[ mustexist, canbedir="false"]
  output:{"-o"}:{ file }:[ canbedir="false"]

FORMAT::
      input output;
```

Fig. 2. A simple interface of an encrypting program.

Figure 2 is how we write it in CLOPS. The list under **ARGS** provides the possible command line options identified as *input* and *output*. For both of the options it specified that they accept file-names as values. Additionally, the file given as *input* must exist and none of the files can be a directory.

The **FORMAT** is a regular expression that tells us that both *input* and *output* must be specified and *input* comes first.

The parser generated according to this CLOPS description will accept command lines such as `-i foo -o bar`, provided that the file `foo` exists.

After a while we get bored with this restrictive form and we decide to enable the user not to specify the output file as the program will derive the name from the name of the input file. Additionally, our colleague who has been using our tool is annoyed by all the debugging messages that are meaningless to him and we introduce the quiet mode. Since we have our first end-user, we add textual descriptions so we can generate simple documentation.

```
NAME:: encrypt1

ARGS::
  input:{"-i"}:{ file }:[ mustexist, canbedir="false"]:"The␣input␣file."
  output:{"-o"}:{ file }:[ canbedir="false"]:"The␣output␣file."
  quiet:{"-q"}:"Turn␣the␣quiet␣mode␣on."

FORMAT::
    quiet? input output?;
```

Note that for *quiet* we have not specified any type, which is the same as writing *quiet*:{`"-q"`}:{**boolean**}.

4 Concepts

The previous section highlighted several common patterns that are seen in command line interfaces. Namely, we have different *types* of options, options *depend* on one another, and the *order* of options on the command line is important.

What follows is a concept analysis of this domain. Each of these concepts is later reflected in the CLOPS DSL.

Options. The command line interface of a particular program focuses on a set of options. Each option has a *type* such as string, number, or file. For parsing purposes, each option is associated with a set of strings (possibly infinite) that determines how the option is specified on the command line. For instance, for the command `head`, any string of the form `-n NUMBER` specifies the number-of-lines option. For a given option, we will call these strings *match strings*. Last but not least, an option has a *documentation string* for end-user documentation purposes.

Values. Parsing a command line results in an internal representation of that particular command line. We will refer to this result as *option values*. As the program executes, these values control its behavior. Each option can be *set* to a value that must agree with the option's type. For example, the value of the number-of-lines option seen above, is an integer.

Format. The ordering of options provided on the command line is often important. Different orderings may have different meanings to the program, and sometimes a different ordering of the same options is not valid. The *format* specifies all valid sequences of options.

We should note that the format not only determines whether a sequence of options is valid, but plays an important role in parsing. For instance, in the case of the version control system `svn`, one can write `svn add add`, where the first occurrence of the string `add` means "add a file to the repository" and the second occurrence means that the file to be added is called `add`. The parser is able to make this distinction as `svn`'s format specifies that a command must precede a filename and only one command use is permitted.

Validity. Only certain combinations of option values are *valid*. A function that determines whether a given combination of the values is valid or not is called a *validity function*. Validity functions are used to enforce constraints on options, for instance ensuring that two mutually exclusive options are not both enabled. Whenever an end-user specifies option values that fail validity testing, the command line parser produces an appropriate error message.

Rules. A change in the value of one option may affect values of other options. Such relationships are represented using *rules* that are triggered during parsing to change the value of other options. More specifically, a rule consists of: a *trigger*, telling us at which point of the format it may fire; a *guard*, a condition for firing; and an *effect*, the action executed when the guard holds. Rules are used to realize, among other features, overriding. For instance, in `ls`, the option `-1` triggers a rule that sets `-C` to *false*, and vice versa.

Command line string. An option is associated with a set of strings that state how that particular option is specified on the command line. However, each command line specifies *multiple* options.

In practice, a command line input is represented as a list of strings. A special character (denoted here by ␣) is appended to each of these strings and then the strings are concatenated to produce the *command line string*. This string will be used as the input to the parser.

Parsing partitions the given command line string into substrings, each corresponding to a different option. For instance, `ls` run with the command line string `-S␣--format=long␣` sorts files by size (`-S␣`) and prints them in the long format (`--format=long␣`).

Usually options come on the command line separately, therefore the match strings typically contain the delimiter character. For instance, the `ls` command's `--format` option *matches* on `--format=FORMATSPECIFIER␣`, where the `FORMATSPECIFIER` is one of `across`, `commas`, `horizontal`, `long`, `single-column`, `verbose`, and `vertical`.

In some programs, the individual match strings do not have to be delimited. For instance, `ps xa` is equivalent to `ps x a`. This means that the option `x` matches on the character `x` *optionally* followed by ␣. These patterns motivated us to represent a command line input as a single string as we can treat such options uniformly.

$$
\begin{aligned}
\text{description} &\rightarrow \text{options format fly}^{?}\ \text{override}^{?}\ \text{validity}^{?}\\
\text{options} &\rightarrow \textbf{OPTIONS::}\ \text{opt}^{+}\\
\text{format} &\rightarrow \textbf{FORMAT::}\ \text{formatRegex}\\
\text{fly} &\rightarrow \textbf{FLY::}\ \text{flyRule}^{+}\\
\text{override} &\rightarrow \textbf{OVERRIDE::}\ \text{overrideRule}^{+}\\
\text{validity} &\rightarrow \textbf{VALIDITY::}\ \text{validityRule}^{+}\\
\text{opt} &\rightarrow \text{optName optAliases optType}^{?}\ \text{optProps}^{?}\ \text{optDoc}^{?}\\
\text{formatRegex} &\rightarrow \text{optName} \mid \text{formatRegex}\ (\boldsymbol{\star} \mid \textbf{+} \mid \textbf{?}) \mid \text{formatRegex}\ \textbf{OR}\ \text{formatRegex}\\
\text{flyRule} &\rightarrow \text{optName}\ (\textbf{:}\ \text{guard})^{?}\ \boldsymbol{\longrightarrow}\ \text{assignment}^{+}\\
\text{overrideRule} &\rightarrow \text{guard}\ \boldsymbol{\longrightarrow}\ \text{assignment}^{+}\\
\text{validityRule} &\rightarrow \text{guard}\ (\textbf{:}\ \text{errorMsg})^{?}\\
\text{optName} &\rightarrow \textit{id}\\
\text{optAliases} &\rightarrow \textbf{:\{}\ \textit{regex}\ (\textbf{,}\ \textit{regex})^{?}\ \textbf{\}}\\
\text{optType} &\rightarrow \textbf{:\{}\ \textit{id}\ \textbf{\}}\\
\text{optProps} &\rightarrow \textbf{:[}\ \text{propName} = \textit{string}\ (\textbf{,}\ \text{propName} = \textit{string})^{*}\ \textbf{]}\\
\text{optDoc} &\rightarrow \textit{string}\\
\text{guard} &\rightarrow \textit{expr}\\
\text{assignment} &\rightarrow \text{optName}\ \textbf{:=}\ \textit{expr}\\
\text{errorMsg} &\rightarrow \textit{expr}\\
\text{propName} &\rightarrow \textit{id}
\end{aligned}
$$

Fig. 3. Syntax of the CLOPS DSL.

5 DSL

We define the CLOPS domain specific language by giving its syntax and its semantics.

5.1 Syntax

Figure 3 gives an overview of the CLOPS syntax[1]. The meta-symbols used in the right hand side of productions (*, $^+$, $^?$, |) have the usual meaning (zero or more times, one or more times, optional, alternative). Parentheses are also meta-symbols. (We omit the literal round parentheses from the grammar for clarity.) Terminals are typeset in **bold**. Nonterminals are typeset in roman if their productions are present, and are typeset in *italics* if their productions are missing.

There are four nonterminals not described in the grammar: *id*, *string*, *regex*, and *expr*. An *id* is a Java identifier[2]. A *string* is a Java string literal. A *regex* is a

[1] See also `http://radu.ucd.ie/temp/clops/grammar.html`

[2] The option type is allowed to contain `-`, unlike Java identifiers

Java string literal s that does not cause an exception when used in $\mathit{Pattern}.\mathit{compile}(s)$. An *expr* is a side-effect-free Java expression whose type is constrained, as discussed later.

A command line parser *description* in CLOPS consists of one or more *option* declarations, a *format* declaration, and, optionally, *rule* declarations. The informal meaning of each of these syntax sections is summarized through a series of examples below. A formal semantics is given in the next section.

Example 1: A Prototypical Option Declaration Here is an example of an option declaration:

```
tabsize:{"-T", "--tabsize"}
       :{int}
       :[minvalue="0" default="8"]
       :"assume given tab stops"
```

The *name* of the option is `tabsize`, it has *aliases* `"-T"` and `"--tabsize"`, its *type* is `int`, and its *documentation string* is `"assume given tab stops"`. Values are given to the option *properties* `minvalue` and `default`. The type `int` is **not** a Java type, but is an *option type*. The built-in types include `string`, `file`, and `boolean`, which is the default option type. The complete set of built-in option types is given later in Table 1, and Section 6 discusses how the type set is extended.

The *match strings* of the option `tabsize` are the words of the regular language specified by the expression `((-T)|(--tabsize))[=␣]([^␣]*)+␣`. The prefix `((-T)|(--tabsize))` is built from the aliases, while the suffix `[=␣]([^␣]*)+␣` is automatically defined by the option type. The suffix is overridden for any option type using the property `suffixregexp`.

For another example, consider again the command `head`. This command permits the user to write `-n NUMBER` but also a shorter version `-NUMBER`.

To specify this shorthand syntax, the suffix would be `(\\d)+␣` (non-zero number of digits followed by the delimiter), and the prefix `-` (a hyphen).

Example 2: Format Strings using Options A convention many UNIX tools follow is to consider everything on a command line that does begin with a hyphen character (`-`) as *not* being a filename. All file names that do start with a hyphen must appear after the special option `--`. This behavior is captured in the format string:

```
(Flag OR File)* (HyphenHyphen HyphenFile*)?
```

Here `Flag`, `File`, `HyphenHyphen`, and `HyphenFile` are option names. Section 6 introduces *option groups*, which make formats easier to write.

Example 3: Fly Rules *Fly rules* are used, among other purposes, to obtain the 'last wins' behavior discussed earlier with regards to the `ls` command. Two fly rules that describe this interface are:

```
HyphenOne -> HyphenC := false
HyphenC -> HyphenOne := false
```

Validity rules, such as the following, are used if an error is preferable:

```
HyphenOne? && HyphenC? -> "Both -1 and -C were given."
```

For fly rules, a trigger is specified via an option name on the left of the arrow. For override rules, the trigger is described similarly. All validity functions are triggered when parsing finishes.

The fly rule examples above do not specify a guard, so the guard defaults to **true**. The validity rule example contains an explicit guard and illustrates some convenient syntactic sugar. The API generated for each option consists of a method to query its value and a method to query if it was set at all. The option name used in an expression is desugared into a call to get the option's value, while the option name followed by a question mark is desugared into a call to check if the option was set. As expected, the (Java) type of the guard must be **boolean**. In the future we plan to allow JML [12] expressions to be used as guards.

The actions of fly rules are assignments. The left hand side of an assignment names an option, while the right hand side contains an expression. The expression must have a type convertible (according to Java rules) to the type of the option's value. Override rules behave the same fashion, but validity rules are different. As seen in the validity rule example, the action of the rule is not written as an assignment, but is instead implicit. In this case, the assigned option has the type `string-list`, and is used to collect error messages.

The validity function is specified through validity rules. These rules are triggered after parsing has finished and, if their guard evaluates to **true**, the string on the righthand side is produced as an error.

5.2 Semantics

As previously mentioned, a CLOPS description is used to generate several artifacts, such as documentation and source code. The command line parser for the described options is the crux of these artifacts.

The mapping of the CLOPS description to a generated parser defines the DSL's semantics. The goal of this section is to provide an operational semantics of the command line parser for a given CLOPS description.

To be able to discuss a CLOPS description formally, we will assume that we have several mathematical objects that correspond to it.

Preliminaries Assume the existence of a finite state automaton $\mathcal{A}$. Each transition of $\mathcal{A}$ is labeled with an option o and a set of rules R. The notation $s \to^{o,R} s'$ denotes that there is a transition from state s to state s' in $\mathcal{A}$, labeled with the option o and the set of rules R.

Legal sequences of options, as given by the CLOPS description, correspond to paths in the automaton that start in the starting state and end in an accepting state. The automaton is built in the usual way from the regular expression specified via the CLOPS format section.

A *rule* is a triple (g, o, e) containing a guard, an option, and a side-effect-free value expression. The intuition behind the rules is that all the rules labeling a transition are triggered whenever that transition is taken. When a rule (g, o, e) is triggered, its guard g is evaluated. If the guard evaluates to *true*, then the expression e is evaluated and the result is assigned to the value of the option o. We say that such a rule has *fired*.

The function *parse* computes a value for a given option from an input string. Hence, its type is Option × String → Value. The manner in which a String is converted to a Value is defined by the Option in question.

The function *match* tries to find a matching string of a given option among the prefixes of the command line string. Its type is Options×String → Maybe String (using the functional languages convention). If the match is successful it returns the prefix on which the option matches, otherwise it returns **null**.

The *option store* is a partial function V from options to their values. The function is partial as when an option has not been set, it has no value.

Details Let us now proceed with the definition of the command line parser based on these mathematical objects (the automaton $\mathcal{A}$ and the functions *parse* and *match*).

The parsing of the command line string is a traversal of the automaton. At each stage of the parse the available transitions from the current automaton state represent the options that are valid for use at this point in the command line string. Each available option is checked to see if it matches at the current position of the command line string. If no option matches, we produce an error. If an option matches we take the transition for that option, fire the rules that label the same transition if their guards evaluate to true, and move forward in the command line string by the match length.

The rest of the section defines this traversal formally.

Some new notation is convenient. The notation $V[x \mapsto v]$ stands for a function that returns $V(y)$ if $x \neq y$ and v otherwise. For the side-effect-free expression e defined on the option values V, the notation $e(V)$ represents the value to which e evaluates after substituting values for options.

The operational semantics of the command line parser is given as a transition function on the parser's states. A *state* of the parser is either an *error state* or a triple $(c,\ V,\ s)$, a command line string (see Section 4), a function from options to their values, and a state of the automaton $\mathcal{A}$, respectively.

For a state $(c,\ V,\ s)$ we define the following two auxiliary sets:

1. Let *Legal* be the set of options labeling a transition from s in $\mathcal{A}$

$$\mathit{Legal}(s) \equiv \left\{ o \mid (\exists s')(s \rightarrow^{o,R} s') \right\}$$

2. Let *Matched* be a subset of *Legal* comprising options that match on c.

$$\mathit{Matched}(s, c) \equiv \{ o \in \mathit{Legal}(s) \mid \mathit{match}(o, c) \neq \mathbf{null} \}$$

The parser transition goes from the state $(c,\ V,\ s)$ to the state $(c',\ V',\ s')$ if *Matched* contains *exactly one* option o and $s \rightarrow^{o,R} s'$. The command line c' is obtained from c by stripping the prefix $Match(o)$.

The values of the options are updated by computing the function *Parse* for o and by triggering, in parallel, the rules labeling the transition being taken. This is formally written as follows.

Let $V'' = V[o \mapsto parse(match(o, c))]$. Then V' is defined as

$$V'(p) = \begin{cases} e(V) & \text{if there is a single rule } (g, p, e) \in R \text{ and } g(V'') \text{ holds, or} \\ V''(p) & \text{there is no such a rule.} \end{cases}$$

This covers the case when $|\text{Matched}| = 1$ and there are no conflicting rules. However, if $|\text{Matched}| = 1$, and there are at least two rules whose guards evaluate to **true** that assign different values to the same option, then $(c,\ V,\ s)$ goes to the error state.

If the set Matched is empty, there is no transition of $\mathcal{A}$ to be taken. If it contains *more* than one option, the parsing is ambiguous. Thus, if $|\text{Matched}| \neq 1$ we also go to the error state.

The computation of the command line parser starts in a state with the command line string to be processed, an option value function not defined anywhere (not assigning any values), and the start state of the automaton.

Note: In Future Work we suggest how one could check *statically* that the size of the set Match is at most one.

6 Implementation

Source code and builds of the implementation are freely available for download on the tool's website[3]. There is also a user-oriented tutorial that details the first steps in running the tool and creating a CLOPS description. The tool has been implemented in Java and requires a Java Runtime Environment of at least version 5.0. Although we chose to write our implementation in Java, there is nothing to stop our approach from being implemented in other programming languages, as we do not rely on any language features that are unique to Java.

The overall functionality of the tool is as one might expect: it parses a specification provided in the CLOPS DSL and produces a command line parser with consistent documentation.

6.1 Option Types

Recall that when a user declares an option in the DSL, they must specify the type of the option. An option type determines the Java type used to represent the option's value and, consequently, the parsing mechanism used for the corresponding option. Table 1 lists the built-in option types.

[3] `http://radu.ucd.ie/temp/clops/tutorial.html`

In order to extend the CLOPS DSL, new option types may be implemented and registered for use. An *option factory* defines the known set of options. Thus, to extend the CLOPS system and provide additional types, a developer extends the default option factory to make the system aware of new developer-provided types. The implementation of new option types can extend the built-in types to reuse and refine their functionality, or one may be written from the ground up, so long as the *Option* interface is satisfied.

Name	Inherits from	Java type	Properties
basic	None	T	default[T], suffixregexp[string]
boolean	basic	*boolean*	allowarg[boolean]
counted-boolean	basic	*int*	countstart[int], countmax[int], warnonexceedingmax[boolean], erroronexceedingmax[boolean]
string	basic	*String*	
string-regexp	string	*String*	regexp[string]
string-enum	string	*String*	choices[string], casesensitive[boolean]
int	basic	int	minvalue[int], maxvalue[int]
float	basic	*float*	minvalue[float], maxvalue[float]
file	basic	*File*	canexist[boolean], mustexist[boolean], canbedir[boolean], mustbedir[boolean], allowdash[boolean]
list	basic	$List\langle T\rangle$	allowmultiple[boolean], splitter[string]
string-list	list	$List\langle String\rangle$	
file-list	list	$List\langle File\rangle$	canexist[boolean], mustexist[boolean], canbedir[boolean], mustbedir[boolean], allowdash[boolean]

Table 1. Built-in option types and properties. Abstract types are typeset in *italics*. T is the type of the concrete option's value.

While parsing each option declaration, the DSL parser queries the option factory for the corresponding option type. The option factory provides the necessary details of an option type: the class representing the option at runtime, as well as the Java type that is used for the value of the option.

The information provided by the option factory about each option's type is used in the code generation phase. An interface is generated that provides access to the value for each option. For instance, for a string option, a *getter* method with return type *String* is created. For each option there is also a method with return type *boolean* that determines whether an option has been set at all.

Each option type has a mechanism for indicating the properties that it accepts. When parsing a given input specification, we can check that the option will actually accept all provided properties. Assuming there are no problems, code is generated that sets the option properties to their specified values during the initialisation phase.

6.2 Option Groups

A tool's format string sometimes grows quite long and is consequently difficult to understand and maintain due to its sheer number of options, independent of their dependencies. For this reason, we allow the options to be grouped.

An option group is defined by specifying a unique identifier paired with a list of options and/or other groups that are contained in the group. Then, when the identifier for a group is used in a format string, its definition is recursively expanded as a set of alternatives. Of course, these are hierarchical groups, and thus must be acyclic. We have found that appropriate use of option groupings make format strings much more concise, understandable, and more easily maintained.

There are many differing styles of documentation used to provide information to the end-user on the command line options available for a given tool. For this reason we have leveraged a powerful and mature open-source templating library, the Apache Velocity Project's templating engine [14], to produce documentation from the information contained within a CLOPS description. We provide several built-in templates for documentation generation, and if a developer requires a different style for their documentation, they can modify an existing template, or create their own.

6.3 Experiments

In order to challenge CLOPS and to measure its effectiveness, several popular programs that have interesting option sets were described using the CLOPS system. Programs that are difficult or impossible to accurately describe using existing command line parsing tools without resorting to a large amount of custom parsing code were explicitly chosen.

Sources of information as to the semantics of a given tool's command line interface include manual pages, source code, command line help (e.g., *tool --help*), as well as trial-and-error with interesting option combinations.

We found describing these tools a useful exercise that led to many adjustments to our design and implementation. The fact that we were successfully able to describe their interfaces emphasizes the power and flexibility of our approach.

Additionally, even though all these programs are used by a large number of people, several inconsistencies were discovered in their documentation, and between their documentation and implementation.

Obviously, using the generated parser guarantees consistency between documentation and implementation. Moreover, the examples discussed in previous sections show that a systematic approach to recording options often highlights inconsistencies in their design, as well as their documentation.

6.4 ls

The program `ls` listing files found in the file-system is frequently used and many options influence which information is displayed and in what format. The `gnu` implementation of `ls` comes with 56 options.

The options worth noting are options determining the format of the output. There is one option `format`, with the enumeration domain, where each of the values from the enumeration corresponds to a boolean option. Snippet from the manual page follows.

```
--format=WORD  across  -x, commas -m, horizontal -x, long -l,
               single-column -1, verbose -l, vertical -C
```

This means that invoking `ls -C` should be the same as `ls --format=vertical`.

This design is clearly a potential source of inconsistencies and we have found one. The option `format` can have only one value at a time, i.e., if defined on the command line multiple times, the last one wins. According to the POSIX standard, however, the option `-1` should be assumed whenever `-l` is given[4]. As the gnu implementation follows the standard, the call `ls -l -1` results in a different output than `ls --format=long --format=single-column`. The former corresponds to `ls -l` while the latter to `ls -1`. This means that the value `single-column` does not have the same effect as the option `-1` and the documentation is inaccurate.

Interestingly enough, the BSD implementation of `ls` that comes on the Mac OS X does not follow the POSIX standard as `ls -l -1` results to the equivalent of `ls -1`, i.e., the options `-l` and `-1` are regarded as distinct options which override one another.

We believe that situations similar to the `format` option arise often in command-line options: as the program evolves, new options are added (like `-1`) and at some point there is too many that deal with a similar issue that an enumeration option is introduced to unify them. If that is the case, maintaining backward compatibility is difficult.

We should note that the enumeration `format` does appear only in the gnu implementation.

How does CLOPS help? Rules let us specify the equality between options. The following CLOPS description snippet relates `format`'s value `commas` and the boolean option `commas`.

```
commas {commas}  -> format := {"commas"};
format {true} -> commas := {format=="commas"};
```

The first identifier is the triggering option followed by a condition and the action. The first rule specifies that if `commas` is set to `true`, the `format`'s value is set to the value `"commas"`. The second rule says that whenever the value of `format` changes, `commas` is set to `true` iff `format`'s value is `"commas"`.

[4] http://www.opengroup.org/onlinepubs/000095399/utilities/ls.html

Note that this specification is not quite symmetric, as it does not specify what should happen when `commas` is set to `false`. In the current solution we disallow the user to set the option explicitly to `false` as the following would result in an inconsistent state (`-m` is a identifier of the option `commas`).

```
--format=commas -m=false
```

Alternatively we could unset the option `format` if an option representing the current value is set to false. Not enabling boolean options to be set to *false* is seen in the gnu implementation.

The source code processing the options in the gnu implementation has 327 non-blank lines of code. The CLOPS description *including* the documentation has 255 non-blank lines.

7 Related Concepts

A CLOPS-generated parser performs two prominent operations: 1) Processes the given sequence of strings (the command line) and returns the set of option values. 2) Decides whether the combinations of option values is a valid one or not.

There is a variety of techniques aimed at describing sets of combinations of certain entities. A grammar describes a set of sequences of tokens; such sets are known as languages. A logic formula corresponds to a set of interpretations that satisfy the formula.

In CLOPS we can find reflections of both: the format string defines a regular language of options imposing restrictions on how options are sequenced on the command line; the validity function imposes restrictions on option values.

We should note, however, that grammars are typically used for quite a different purpose than the command line as they typically correspond to complex nested structures that drive compilers, interpreters, etc.

On the other hand, the elements on the command line live quite independently of one another and each correspond to a wish of the user running the program. Some elements of the command line provide necessary information for the program to carry out its job, such as `cp fileSrc fileDest`—the copy program hardly can do anything if it does not know which files to copy or where. Some elements of the command line trigger a certain behavior of the program. For instance, the program `ls` will happily run with or without the argument `-C` (column formatting) or the argument `-1` (one-per-line formatting). Each of these arguments corresponds to a *feature* of the program and the command line enables the user to express which features he requires.

In the program `ls`, the feature `-l` (long listing) is even so popular that users typically alias the invocation `ls -l` to `ll`. This provides us with an alternative view on the program: the original `ls` represents a family of programs and `ll` is a member of that family.

This brings us to another relevant domain *Software Product Lines (SPL)*, which studies families of programs [5]. In SPL an individual program is combination of certain features, and a family of programs is a set of feature combinations. The individual features and dependencies between them are captured in *feature model* [11].

In SPL terminology, the command line options represent variability of the program in question. In general, variability can be *bound* at different times, e.g., at *compile-time* or at *run-time* [**?**]. In our case, it is always bound at *run-time*. From the user perspective, however, there is no difference between an alias `ll` and a program called `ll` (the efficiency issue here is negligible).

Thus, a CLOPS description embeds a feature model of the program in question. And, deciding on which command line options should be supported corresponds to *feature oriented domain analysis* (FODA) [11].

This discussion brings us back to grammars, in SPL the relation between grammars and feature models is not new [2]. Grammars are particularly useful in approaches where the order of features is important [3].

8 Related Work

There is a rather large number of libraries for processing command line options. We identified three main groups of command line libraries.

The first group consists of libraries that follow the philosophy of the original Unix `getopt` function (Commons CLI [6] and JSAP [10]). These libraries usually provide only limited capabilities on top of basic `getopt`. Option values are queried via a *Map*-like interface and are usually treated as *String*s or a particular type explicitly predefined by a getter method name. Libraries from the first group are ignorant of dependencies between options—dependencies must be explicitly handled by the tool developer.

Libraries in the second group recognize options via annotations of variables or methods in the source code (JewelCLI [9] and Args4j [1]). This approach has several advantages to the `getopt`-like option specification. Types are extracted directly from the source code and therefore the parser automatically checks parameter syntax. Unfortunately, libraries in the second group also ignore option interdependencies.

The third group contains libraries that process options externally specified (JCommando [8]). An external specification is compiled into a library, much like in the CLOPS approach.

Table 2 summarizes and compares some of the best known of these libraries with CLOPS in several ways. Libraries are compared based upon their approach, features, and flexibility.

Help extraction is the ability to generate help documentation from the command line specification. *Incremental options* are options where multiple uses change the state of the option in some way, for example using `-v` more than once often increases the level of verbosity. *Relations among options* are mechanisms to specify dependencies between options (for instance, two options that cannot be used together, an option that can only be used in conjunction with another, etc.).

Table 2. Features provided by the most popular CLO libraries.

	CLOPS	JewelCLI	Commons CLI	Args4j	JCommando	JSAP	getopt
Version	1.0	0.54	1.1	2.0.1	1.01	2.1	
Specification	Separate DSL	Source Annotation	Source Builder p.	Source Annotation	Separate DSL/XML	Source Builder p.	Source Table
License	MIT	Apache 2.0	Apache 2.0	MIT	zlib/libpng	LGPL	GNU
Option name variations	Regexp	List	Short/Long	No	Short/Long	Short/Long	Short/Long
Alternatives to leading hyphen	Yes	No	No	Yes	Command only	No	No
Option-argument format (e.g. `-m X`, `-m=X`, `-mX`)	Yes	No	No	No	No	No	No
Dynamic option names (e.g. `gzip -1,-2,..,-9.`)	Yes	No	No	No	No	No	No
Optional option-arguments	Yes	No	No	No	No	No	Yes
Default values for options	Yes	Yes	No	Yes	No	Yes	No
Types of option-arguments	Basic, special	Java prim., class, list	String	Basic, special	Basic	Java prim., List	char*
Option-argument validation	Type, attr., regexp	Type, regexp	None	Type	Type, range	Type	None
Command line format	Yes	No	No	No	No	No	No
Multiple use of one option	Yes	Last one	First one	Last one	Last one	First one	Call-backs
Help extraction	Yes	No	Yes	Yes	Yes	Yes	No
Fly rules	Rules	No	No	Call-backs	Call-backs	No	Call-backs
Relations among options	Yes	No	No	No	Limited	No	No
Option and arg. separator	Arbitrary	No	Yes, `--`	No	No	No	Yes, `--`
Option concatenation (old UNIX option style, e.g. `tar xzf`)	Yes	No	No	No	No	No	Yes (hyphen required)
Incremental options	Yes	No	No	Call-backs	Call-backs	No	Call-backs

Some of the libraries allow *call-backs* during specific stages of a parse (e.g. after an option has been set). Thus, for some of the abilities detailed it is possible for the developer to write extra code to achieve the desired effect. In these cases the libraries do not facilitate the functionality, but do not prevent it either.

Most of the libraries support short and long variants of option aliases. Supporting a list of variants, rather than only a short/long pair, is rather exceptional (JewelCLI). Strict adherence to a finite enumeration of option-alias strings is a limitation in the flexibility of specifying option names. For example, the `tail` command prints a number of lines from the end of the input, where the exact number is specified as an option of the form `-NUMBER`. Even if a list of variants is allowed, specifying the aliases for this type of option would be too verbose to be an acceptable approach.

The leading hyphen is used almost exclusively as an option prefix. However, alternative prefixes are often attractive in some situations. For example, the `chmod` command uses both `-` and `+` as prefixes for options and effectively distinguishes when a particular feature is to be enabled or disabled. Another example is the tradition of some Windows command line tools to use slash (`/`) instead of hyphen (`-`).

Although it is often useful, the format of a command line is another feature omitted by most libraries. Many tools (including `svn`, `zip`, and `chown`) require a command name early in a command line in order to properly process later options. If there is no support for format within a library, as is nearly uniformly the case, such functionality must be hard-coded.

Dependencies among options are almost always ignored in current command line utilities and, therefore, manual testing the validity of option values is delegated to the user of the library. As mentioned in the Introduction, this leads to overly complicated and potentially buggy code.

As an exception, we have found the JCommando [8] library. JCommando introduces the notion of a *command*, which is similar to an option except that only one command can be set by a command line. The set method for the command is called last, after all option set methods. The fallback command ("commandless") is used if no others are set by the user.

A command can specify what options can/should have been set for a valid parse. This is done using a boolean expression over option identifiers. Each identifier will evaluate to **true** or **false** at runtime according to whether that option was set or not. The boolean expression for the provided command is evaluated when that command is set, and if it evaluates to false an error is produced. This is a much simpler form of validity check than we allow. In particular, there is no way to relate options to each other outside the context of a command, nor is there a way to use option values in the test.

Surprisingly, many libraries do not provide a mechanism to enable a developer to specify, for example, filenames that start with a hyphen as any string starting with a hyphen is assumed to be an option. UNIX `getopt` solves this problem using the special string `--`, which is modeled in a CLOPS specification by the simple aforementioned format expression (see Section 5.1).

Feature modeling [11] has a goal similar to that of our DSL. A *feature model* explicitly captures the variability and commonality of a program [5]. In fact, one can imagine a program as a family of programs whose members correspond to the possible configurations of the feature model, as expressed via command line options. Whereas feature models target various types of variabilities at design- and compile-time, command line options represent variability resolved at the runtime.

9 Conclusions and Future Work

Many command line tools solve the command line parsing problem using custom code, sometimes relying on a little help from specialized libraries. "A DSL is viewed as the final and most mature phase of the evolution of object-oriented application frameworks," according to Deursen et al [13]. CLOPS aims to succeed and supersede existing libraries in this regard. Combining an analysis of the set of existing solutions with the identification and refinement of the domain-specific concepts of command line options, a minimized conceptual framework is defined. Consequently, the syntax of the DSL is concise and self-documenting, reflecting exactly this conceptual design, which lends itself to a gradual learning curve. Additionally, while the DSL is simple enough to formalize and is easy to use, it is also powerful enough to cover all command line conventions of which we are aware.

The implementation is done classically, by writing a compiler that generates Java source code. From our experience of reimplementing the interface of standard UNIX tools and implementing the interface of some other tools, CLOPS increases a programmer's productivity, though the quantification of such is the subject of further work.

Some aspects are painfully missing from the current implementation. One compelling aspect of many DSLs is that consistency checks are accomplished at a higher level of abstraction. One such consistency check for CLOPS is the ability to statically analyze a CLOPS description to guarantee that the generated command line parser compiles and does not fail in certain ways. For example, the CLOPS system might perform type-checking instead of deferring such to the Java compiler. Creators of other DSLs found that early type-checking is sometimes more precise typechecking [13]. Another potential static check is ensuring regular expressions used to match options that can legally appear in the same parsing state are disjoint. This would ensure that the *Matched* set contains at most one element. (Of course, the parser would still fail at runtime when *Matched* is empty.) Finally, we require, but do not check, that the expressions used in rules do not have side-effects. Existing research on containing side-effects in languages like Java will prove useful [7,4].

Other parts are missing, but are less troublesome. A hassle that accompanies code generation is a more complicated build process. This annoyance is sometimes avoided by generating the parser at runtime. Of course, such an architecture change may hurt performance, but, for many command line parsing tools, we expect performance to be less important than ease of use.

In various flavors of UNIX there are many implementations of common command line POSIX utilities such as `chmod` and `ls`. After describing a single command line interface for a given tool across a suite of implementations one might generate command line strings automatically for system-level testing. Such work has been done in a limited form already here at UCD for the OpenCVS project.

Finally, some of this work is applicable in the context of tools with graphical, rather than textual, interfaces. GUI programs often store their settings in a preferences store that is read upon start-up, and many such programs provide a 'wizard' (e.g., a preferences pane) to populate the settings file. While it requires is no stretch of the imagination to envisage the CLOPS framework generating the wizards and the parser for such GUI programs, important details need to be clarified before this step is made.

References

1. Args4j. http://args4j.dev.java.net/.
2. Don Batory. Feature models, grammars, and propositional formulas. In H. Obbink and K. Pohl, editors, *Proceedings of the 9th International Software Product Line Conference (SPLC '05)*, volume 3714 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2005.
3. Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1992.
4. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, 1998.
5. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison–Wesley Publishing Company, 2002.
6. The Apache Commons CLI library. http://commons.apache.org/cli/.
7. A. Darvas and P. Muller. Reasoning about method calls in JML specifications. In *Formal Techniques for Java-like Programs*, 2005.
8. JCommando: Java command-line parser. http://jcommando.sourceforge.net/.
9. JewelCLI. http://jewelcli.sourceforge.net/.
10. JSAP: the Java Simple Argument Parser. http://www.martiansoftware.com/jsap/.
11. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA), feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, November 1990.
12. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.
13. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.
14. The Apache Velocity Project. http://velocity.apache.org/.

A Examples

A.1 gzip

```
NAME::
 Gzip


ARGS::

ascii:{"-a","--ascii"}
stdout:{"-c","--stdout","--to-stdout"}
decomporess:{"-d","--decompress","--uncompress"}
force:{"-f","--force"}
help:{"-h","--help"}
list:{"-l","--list"}
license:{"-L","--license"}
noname:{"-n","--no-name"}
name:{"-N","--name"}
quiet:{"-q","--quiet"}
recursive:{"-r","--recursive"}
rsyncable:{"--rsyncable"}
test:{"-t","--test"}
verbose:{"-v","--verbose"}:{counted-boolean}:[countmax="3",warnonexceedingmax]
version:{"-V","--version"}
compressionlevel:{"-"}:{int}:[suffixregexp="([1-9])\\00",maxvalue="9",minvalue="1",default=
compressionlevelbest:{"--best"}
compressionlevelfast:{"--fast"}

suffix:{"-S","--suffix"}:{string}

FORMAT::

option* suffix?;

WHERE::

option:
ascii OR stdout OR decomporess OR force OR help OR
list OR license OR noname OR name OR quiet OR recursive
OR rsyncable OR test OR verbose OR version OR
compressionlevel OR compressionlevelbest OR
compressionlevelfast;

FLY::
// Compression level aliases
```

```
compressionlevelbest -> compressionlevel:={9}, compressionlevelfast:={false};
compressionlevelfast -> compressionlevel:={1}, compressionlevelbest:={false};

// List forces decompress and stdout
list -> decompress:={true};
list -> stdout:={true};

// Test forces decompress and stdout
test -> decompress:={true};
test -> stdout:={true};

// Quiet and verbose are reversed
quiet -> verbose:={0};
verbose -> quiet:={false};

OVERRIDES::
// Default compression level is 6
{$(compressionlevel?)} -> compressionlevel:={6};

VALIDITY::
```