Parsing the LCFG Language

Stefani Genkova

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

The LCFG is a system for automatically installing and managing the configuration of large numbers of machines. The system makes use of its own declarative language, which is used to specify the required configuration of the machines being managed. The drawbacks of the current implementation are the informal syntax of the LCFG language and the use of an external tool (the C preprocessor, CPP) to preprocess the configuration files.

This report presents the creation of an explicit grammar for LCFG and the design, implementation, and testing of a parser for the language. The proposed grammar combines the basic LCFG syntax with a subset of CPP directives. Specifically, this report uses the ANTLR tool to generate a parser from the grammar specification. The parser is further integrated into a language application that processes the parse tree in order to gather configuration parameters and generate XML profiles. The proposed solution allows an LCFG source to be parsed in a single step without the need of the C preprocessor. The report concludes with a summary of the work done and proposes potential further extensions.

Table of Contents

1 Introduction **4**
1.1 Motivation 4
1.2 Project goal 5
1.3 Summary of results 5
1.4 Structure of the report 6

2 Background **7**
2.1 LCFG architecture 7
2.2 LCFG configuration files 8
2.2.1 Source files 8
2.2.2 Header files 8
2.2.3 Schema files 8
2.2.4 Package list files 9
2.3 LCFG Language 9
2.3.1 Resources 9
2.3.2 Mutation 10
2.3.3 Tag lists 10
2.3.4 References 11
2.3.5 Spanning maps 11
2.4 The C preprocessor 11

3 Unified LCFG grammar **13**
3.1 Analysis of LCFG configuration files 14
3.1.1 Inclusion of header files 14
3.1.2 Object-like macros 14
3.1.3 Function-like macros 15
3.1.4 Conditionals 15
3.1.5 Errors and warning 15
3.1.6 Comments 15
3.1.7 Unnecessary CPP features 16
3.2 Unified LCFG language concepts 16
3.3 Unified LCFG grammar rules 17
3.3.1 Statements 19
3.3.2 Resources 19
3.3.3 Variables 20
3.3.4 Functions 20

3.3.5 Procedures . . . 21
3.3.6 Conditionals . . . 21
3.3.7 Expressions . . . 22

4 LCFG parser 23
4.1 Tools . . . 23
4.2 Lexer rules specification . . . 24
4.3 Parser rules specification . . . 26
4.4 LCFG language application . . . 27
4.4.1 Object classes . . . 28
4.4.2 Visitor functions . . . 31
4.4.3 Macro processor . . . 33
4.4.4 Profile generator . . . 38
4.4.5 Usage . . . 39

5 Evaluation 41
5.1 Basic tests . . . 41
5.1.1 Resource assignment test . . . 41
5.1.2 Resource mutation test . . . 42
5.1.3 Variable assignment test . . . 43
5.1.4 Conditionals test . . . 43
5.1.5 Functions test . . . 45
5.1.6 Procedure call test . . . 46
5.1.7 References test . . . 47
5.1.8 Extended mutation operators test . . . 48
5.2 System tests . . . 49

6 Conclusion 54

Bibliography 56

Appendices 58

A LCFG grammar specification 59
A.1 Lexer rules . . . 59
A.2 Parser rules . . . 62
A.3 `MacroProcessor` rules . . . 64

B Evaluation results 66
B.1 Profiles comparison report (summary) . . . 66
B.2 Profiles comparison report (detailed) . . . 69

Chapter 1

Introduction

1.1 Motivation

Managing the configuration of computers in sites consisting of a large number of diverse hosts is recognised as being a challenging task for system administrators. The factors that complicate the system configuration process of large sites are identified in [10]: relationships between components on different machines, frequent configuration changes, diversity ranging from laptops to database servers, coordination between administrators, deployment of complex distributed applications, reconfiguration of autonomic systems without manual intervention, usability of configuration tools, security issues, etc. To address the system configuration problems and to eliminate the mistakes due to the manual configuration process, a number of automated solutions have been developed. Examples of such automated configuration tools include Cfengine [13], Nix [14] and LCFG [8].

LCFG (**L**ocal **C**on**Fi**G**uration system) is an automatic configuration and installation system originally developed by the Department of Computer Science at the University of Edinburgh around 1993. The system has been very successful in handling the department's network of several hundred Unix machines. Later expansions in the local use of LCFG introduced a major update to the original version known as LCFGng, which is now being used more widely outside of the University of Edinburgh [12]. LCFG makes use of its own language, which is used to specify the required configuration of the machines being managed. Every machine that should be configured has its own source file. A group of machines may share common sets of configuration parameters via header files organised into hierarchies.

The LCFG language has evolved over the years. Because of permanent experimental additions and the need for backward compatibility with legacy systems part of the current LCFG language has an informal syntax. The LCFG server uses an ad-hoc parser and relies on the C preprocessor to include header files and to handle conditionals, macro definitions, and macro expansions. Since the C preprocessor is designed to process C source code, which does not have the same syntax as LCFG source files, some character strings are mistakenly interpreted by the preprocessor

[6]. Problems with LCFG are mainly caused by comment characters and string quoting used in source files. More specifically, single apostrophe could be interpreted as the beginning of a constant string; and an error will be issued if a closing single quote cannot be found. For example, the following LCFG declaration (taken from `/core/include/lcfg/defaults/syslog.h` file):

```
syslog.appselector_auth ifdef(`AUTHDEBUG', auth.debug,auth.info)
```

causes a missing terminating character warning when processed by CPP.

The above-mentioned drawbacks and the lack of clear grammar for the language make it impossible to produce other tools that process/analyse LCFG sources or to write alternative compiler implementations.

1.2 Project goal

The goal of this project is to create an explicit grammar for LCFG and implement a prototype parser. This should form a basis on which further tools for processing configuration descriptions can be built, and the language can be further extended efficiently. To achieve the goal, at the start of the project a set of tasks to be carried out was determined:

- Study the LCFG architecture and LCFG configuration language.
- Investigate the feasibility of parsing the LCFG source files in a single step without using the C preprocessor; propose an LCFG grammar.
- Choose and study an appropriate tool for generating lexers and parsers.
- Compose the lexer and parser rules for the LCFG grammar; generate a lexer and parser for the grammar.
- Extend the generated parser to create a language application that processes the parse tree in order to collate configuration parameters and generate XML profiles.
- Evaluate the parser against real system configurations.

1.3 Summary of results

The main results of the project can be summarised as follows:

- A unified grammar for LCFG language is proposed. This grammar combines the basic LCFG syntax with a subset of C preprocessor (CPP) directives that is sufficient for inclusion of header files, pre-definition of configuration parameters and conditional assignment of values to resources. A set of LCFG examples is prepared for testing and demonstrating of the grammar.

- An LCFG parser is implemented. The parser is developed with the aid of the ANTLR tool and further integrated into an LCFG language application, which collates all of the configuration information from the included header files and creates a single XML profile file for a given host.
- An approach for processing CPP/C files without the use of the C preprocessor is adapted and implemented to parse the combination of LCFG basic elements and CPP directives as a single language.
- A small set of additional operators for modifying resource values (mutation) is implemented and tested.
- The parser and the LCFG application are validated against the output of the production LCFG compiler using real configuration files.

1.4 Structure of the report

The remaining part of the report is structured in five chapters.

Chapter 2 provides a brief overview of the LCFG system and the syntax used to specify configuration parameters.

Chapter 3 explains the approach applied to address the processing of LCFG language elements and CPP directives as a single language and presents the proposed LCFG grammar.

Chapter 4 describes the design and implementation of the LCFG parser using the ANTLR tool and the proposed grammar.

Chapter 5 presents the results from the evaluation and testing of the implemented parser and LCFG language application.

Chapter 6 concludes the report with a summary of the work done and proposes potential further extensions.

Chapter 2

Background

2.1 LCFG architecture

As described above, the LCFG is a system for automatically installing and managing the configuration of large numbers of machines. The LCFG architecture is shown in Figure 2.1.

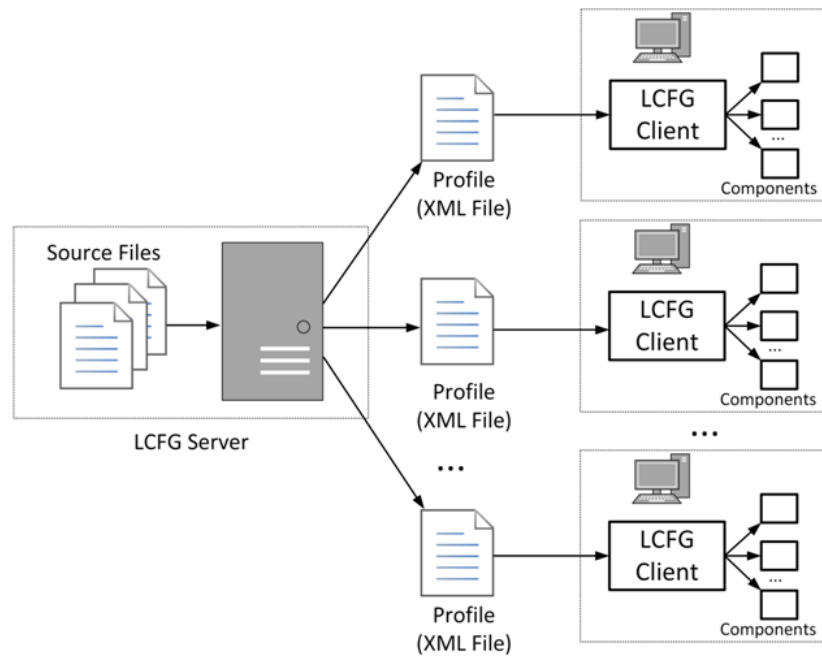

Figure 2.1: The LCFG architecture

There are three main parts to the LCFG software [11]:

- The LCFG server. The server collates all of the configuration information from the source files and creates a single profile file in XML format for each machine. The profile contains all of the configuration parameters (resources) for that machine. The profile file is usually published by the Web server to make the

profile available to the client. When a profile changes, the LCFG server sends a notification to the LCFG client.

- The LCFG client. Each machine runs its own copy of the client. The client polls the LCFG server at regular intervals. Upon receiving a change notification from the server, the client downloads a new profile for the machine and calls the corresponding scripts to apply the configuration.
- LCFG components. Each machine includes a collection of scripts called components. They are responsible for translating the LCFG resources into machine-specific configuration files. Each component is responsible for a self-contained subsystem. The component generates the configuration file and takes care of other low-level details such as restarting any associated daemons.

2.2 LCFG configuration files

The LCFG server maintains a central repository of configuration settings. All the configuration information is stored in plain-text files. There are four different types of configuration files: source files, header files, schema files, and package lists [11].

2.2.1 Source files

Each machine in the system has its own source file. This file contains the configuration description for the machine and may include one or more header files. In practice, common parameters for a group of machines are described in header files. Thus, the source files contain only those configuration parameters that are unique to that particular machine. In addition, a source file may edit or override the values of common parameters previously defined in header files.

2.2.2 Header files

Header files provide a way for supporting inheritance between configuration objects. They contain common sets of configuration parameters, which can be included by the source files or other header files. Header files have the extension `.h`.

2.2.3 Schema files

Each component has its own schema file. It defines the resources the component uses, the default values for the resources, and possibly some validation or type information for them. The default values are used in the profile if no explicit value is provided by the source files. Each resource in a schema file may have its own meta-resource (beginning with `'@'`), which specifies types and validation code for resource values, as well as templates for building nested lists of records. Similarly to source files, schema

files may include declarations from header files or other schema files. Schema files have the extension `.def`.

2.2.4 Package list files

Package lists specify the software packages to be installed on a machine. The LCFG server passes this list to the client, where it can be processed by a component. The component updates the installed packages on the machine so that they conform to the supplied list. The package list files usually have the extension `.rpms`.

2.3 LCFG Language

The LCFG language is a declarative configuration language. The language specifies what the configuration of a system should look like rather than describing the steps that need to be taken to make changes happen.

The source files and header files contain resource values and various directives for manipulating them [9].

2.3.1 Resources

The configuration parameters are described by key/value pairs called *resources*. The syntax for defining a resource is:

```
component.attribute value
```

The resource name consists of a *component* name and an *attribute* name separated by a dot. The *value* is an arbitrary string which is separated from the resource name by whitespace, for example:

```
dhclient.mac     00:16:3E:a3:0d:18
sysinfo.owner    inf.ed.ac.uk
inv.model        Apple MacBook Pro
```

Once a resource value is assigned, it can only be changed using a *mutation*. If no value is supplied for a resource, the default value from the components schema file is used.

A special component named `profile` must be included in each source file. Its attributes are interpreted as directives to the LCFG compiler. For example, the resource `profie.components` is used to list the components to be included in the generated profile file.

2.3.2 Mutation

A special notation, referred to as *mutation*, is provided to prevent resource values from being overridden by accident. The mutation feature is mainly used to edit or override the defaults for resource values specified in header files. To indicate a mutation, the resource is prefixed with an exclamation mark (`'!'`).

The current implementation uses a number of special predefined functions that compute a new value for the resource from both the previous value and the function argument. These macros are defined in a header files named `mutate.h`. The following code demonstrates the application of some frequently used mutation functions:

```
/* Override the previous value of dns.zone_inf */
!dns.zone_inf       mSET(inf.ed.ac.uk)
/* Append an item to a (space-separated) list */
!mailcap.filetypes  mEXTRA(audio)
/* Append an item to a list if it is not already present */
!profile.components mADD(client)
/* Remove an item from a (space-separated) list */
!roles.netgroups    mREMOVE(critlevel_low)
/* Replace the item 6 in a list with item 6a 6b 6c */
!rsyslog.ruleblock_syslogRules  mREPLACE(6,6a 6b 6c)
```

2.3.3 Tag lists

Some resources define a list of items rather than a single value. The value of such a resource is a space-separated list of tags. The members of the list are identified by the tags appended to a common prefix in their attributes. For example, the following code declares a list structure with three tags: *lcfglib*, *lcfgdata* and *lcfgbin*:

```
sysinfo.paths   lcfglib lcfgdata lcfgbin
sysinfo.path_lcfglib    /usr/lib/lcfg
sysinfo.path_lcfgdata   /usr/lib/lcfg/conf
sysinfo.path_lcfgdata   /usr/bin
```

Each tag list resource has its meta-resource in the corresponding schema file, for example:

```
@paths path_$
```

The value appearing in the meta-resource is a template (or a list of templates) used by the compiler to identify possible list elements. The template itself defines a common prefix and has a `$` as a placeholder for each tag value. For example, given the template `path_$`, the compiler will create the following nested structure as a part of `sysinfo` component:

```
  paths
    lcfglib
      path = /usr/lib/lcfg
    lcfgdata
```

```
      path = /usr/lib/lcfg/conf
    lcfgbin
      path = /usr/bin
```

2.3.4 References

References provide a way to link a value of a resource with some resource from the same or another component. There are two kinds of references: *late references* and *early references*. An example of a late reference is shown in the following declaration:

```
network.hostname    <%profile.node%>
```

where the `network.hostname` resource will take the final value of `profile.node`. *Early references* assign the current value of a resource and are specified using a double percent sign, for example <`%%profile.node%%`>.

2.3.5 Spanning maps

While references enable one resource to refer to the value of some other resource of the same node, spanning maps provide a mechanism for referencing resource values from other nodes. Using this mechanism, a node may publish some of its resource values to a specific spanning map. After publishing, these resources become available to other nodes which subscribe to the spanning map.

Resources to be published/subscribed are specified using `publish` and `subscribe` directives in their corresponding meta-resources defined in schema files:

```
@serversmap %publish: dhcluster
@dhcpdservers %subscribe: servers
```

To avoid inconsistent configurations, the LCFG compiler defers the publication of profile files for subscriber nodes until all compilations have been completed.

2.4 The C preprocessor

As mentioned previously, the current implementation of the LCFG compiler passes the source files through the C preprocessor (CPP). The CPP is a macro processor that is used by the C compiler to transform the source code before compilation. Although C preprocessors vary in some details, the full set of CPP features is documented in section 6.10 of the C language ISO standard [17]. A number of options are provided that can alter the default C preprocessor output.

The CPP is intended to process only C/C++ programs. Although it can be used independently, a number of errors are reported to occur when processing files that do not obey C lexical rules [6].

The CPP directives are not part of the current LCFG syntax. To demonstrate a typical error that may occur due to preprocessing performed by CPP, let us consider the following LCFG code:

```
1 #define DOMAIN inf.ed.ac.uk
2 mail.address `student' <student@DOMAIN>
```

Processing line #2, the CPP ignores the opening (back) quote of `student` and treats the closing apostrophe as the beginning of a string. Assuming the `DOMAIN` is inside a string, the CPP does not perform its substitution with `inf.ed.ac.uk`. Hence, the resulting assignment would be:

```
mail.address = `student' <student@DOMAIN>
```

instead of:

```
mail.address = `student' <student@inf.ed.ac.uk>
```

The following chapter discusses the possibilities of processing the LCFG source files without the CPP. It proposes a unified LCFG grammar, which will be further used to compose a parser that processes the LCFG files without a separate CPP preprocessing step.

Chapter 3

Unified LCFG grammar

Parsing code which contains a combination of two different languages is usually not trivial. The task of parsing such code involves solving various context-sensitive problems. For example, identifiers in the first language can also be keywords in the second language; the same character sequence can be one token in the first language or multiple tokens in the second, etc.

To briefly demonstrate the challenges in parsing code containing both LCFG and CPP syntax, let us consider the example LCFG source code shown in Listing 3.1.

```
1  syslog.addselector  local.debug
2
3  #if defined _USE_NETINF_DNS_H
4  DNS_SLAVE_FWD_Z(creent,creent.org)
5  !dns.file_creent    mSET(zone.creent)
6  #else
7  #error _USE_NETINF_DNS_H not defined
8  #endif
9
10 #define FILE_SYMLINK(T,L,D) \
11 !file.files         mADD(T) ¢\
12 file.file_/**/T     L ¢\
13 file.type_/**/T     link ¢\
14 file.tmpl_/**/T     D
```

Listing 3.1: Code containing a combination of LCFG and CPP syntax

In line #1, character sequences `syslog.addselector` and `local.debug` are lexically identical, but the first must be recognised as a resource name, and the second as a resource value. The `creent.org` (line #4) should be treated as a parameter instead of a resource name. The keyword `defined` (line #7) must be processed as a part of a macro body instead of generating a single token. Lines #11..#14 should be treated as a macro body containing formal arguments (`T`, `L`, and `D`) instead of immediate resource assignments. The special char ¢ should be processed as a new line character and therefore excluded from resource values in lines #11..#13. Comments in lines #12..14 serve to separate formal parameters from the other text and must be removed only after macro resolving instead of being skipped by the lexer.

In the initial stage of the design, considerable attention was paid on deciding how to integrate the CPP and LCFG syntax in the grammar in such a way as to address the aforementioned language recognition problems. An important part of this decision was selecting the subset of the CPP features to be included in the grammar. It is obvious that in order to parse real source files successfully, the chosen subset should match the CPP subset which is actually being used in practice. On the other hand, CPP macro parameters and text should be firstly analysed before the actual replacements take place.

This chapter first discusses the approach applied to process the combination of LCFG syntax elements and CPP directives as a single language. Then, it introduces the proposed unified LCFG grammar. Throughout the text, we will use the term *unified* to denote the joining of LCFG/CPP elements.

3.1 Analysis of LCFG configuration files

At the beginning of this project, a set of real configuration LCFG files was provided. The set contained 4050 source files, 2104 header files and 215 schema files organised in sub-directories.

The first obvious task was to examine the source and header files in order to clearly identify the subset of the CPP features used in configurations, and to exclude the unnecessary functions. The results of this analysis are outlined below.

3.1.1 Inclusion of header files

Header files are included into LCFG source files using the preprocessing directive of the form:

```
#include <file>
```

which searches for a file in a list of implementation-defined directories. The other two forms of the directive: `#include "file"` and `#include text`, are not used.

3.1.2 Object-like macros

Object-like macros in LCFG files are used in their default form:

```
#define identifier replacement-list
```

where `replacement-list` can be empty.

3.1.3 Function-like macros

Function-like macros are used in their most used form:

```
#define identifier(identifier-list) replacement-list
```

where `identifier-list` is a comma-separated list of parameters, which can optionally be empty.

Macros that accept variable number of parameters of the following forms are not used:

```
#define identifier( ... ) replacement-list
#define identifier(identifier-list, ... ) replacement-list
```

For example, mutation functions are defined as function-like macros in `mutate.h` header file. In both object-like and function-like macros a newline terminates the macro definition. A macro may contain two or more continued lines which end with a backslash symbol (‘\’).

3.1.4 Conditionals

All the CPP conditionals are found to be used in the LCFG files:

```
#if constant-expression
#ifdef identifier
#ifndef identifier
#elif constant-expression
#else
#endif
defined identifier, defined (identifier)
```

3.1.5 Errors and warning

LCFG files use both `#error text` and `#warning text` directives.

3.1.6 Comments

LCFG files use only non-nested block comments:

```
/* comment */
```

It was already shown in Listing 3.1 that comments in some LCFG descriptions serve to separate tokens that, if adjacent, would be recognised in another manner.

3.1.7 Unnecessary CPP features

The following CPP features are not revealed in the provided set of LCFG files:

- Line control (directive `#line`)
- Pragmas (directive `#pragma`)
- Predefined macros (`__FILE__, __DATE__, __TIME__, etc.`)
- Concatenation (operator `##`)
- Stringification (operator `#`)

3.2 Unified LCFG language concepts

In literature, there are various techniques proposed to parse both the C language and its preprocessor directives as a single language. These techniques are used, for example, by source browsers and automated refactoring tools.

The proposed solution was influenced by the ideas discussed in the two papers presented next.

A solution called `SuperC` is presented in [16]. `SuperC` processes the source code in two steps. First, it uses a configuration-preserving preprocessor, which includes header files and resolves macros. Unlike CPP, the configuration-preserving preprocessor leaves static conditionals intact, thus preserving a programs variability. Secondly, a configuration-preserving parser generates an abstract syntax tree with static choice nodes for conditionals. It forks new subparsers when encountering static conditionals and merges them again after the conditionals. The paper analyses the interactions between C preprocessor and language features and identifies techniques for correctly handling them, but does not provide implementation details.

Another approach proposed by J. Favre in his paper "CPP Denotational Semantics" [15] shows that CPP can be seen as a programming language. This is carried out by mapping the CPP entities to the entities of an abstract language called `APP` (`Abstract PreProcessor`). In this mapping, CPP directives are statements, object-like macros are variables, function-like macros are functions, and files are procedures.

It was decided to adopt the approach discussed in [15] to propose the concepts of the unified LCFG language. Table 3.1 shows the correspondence between these concepts and their equivalent underlying CPP/LCFG elements.

The rationale behind the choice of CPP syntax was to provide elements that would be sufficient for the inclusion of header files, pre-definition of configuration parameters and conditional assignment of values to resources, and to exclude the unnecessary elements previously listed in Section 3.1.7.

It can be seen that the unified LCFG could be considered as an imperative language that does not provide loops.

Table 3.1: Unified LCFG language concepts

Concepts	CPP/LCFG elements
Resource	Component/attribute pair
Resource value	Configuration parameter value
Resource assignment	Configuration parameter assignment/mutation
Variable	Object-like macro name
Variable value	Object-like macro body
Variable assignment	Directive #define (object-like macro), #undef
Function	Function-like macro name
Function declaration	Directive #define (function-like macro)
Function formal parameter	Function-like macro formal parameter
Function call	Function-like macro invocation
Function actual parameter	Function-like macro invocation parameter
Procedure	A file to be included
Procedure call	#include <file>
Conditional statement	#if, #ifdef, #ifndef, #elif, #else, #endif
Expression	defined id

3.3 Unified LCFG grammar rules

This section introduces the proposed unified LCFG grammar. The grammar will be specified using a variant of Extended Backus-Naur Form (EBNF), which is widely used to define programming languages. This specialised variant makes use of operators that are more commonly used in regular expressions, and is utilised by the ANTLR tool, which will be further used in Chapter 4 to build the LCFG parser. The grammar presented below uses the following conventions:

- The alternatives of a rule are separated by the pipe sign ('|');
- Multiple elements are grouped together by using round brackets;
- The '?' sign marks an element as optional (in the standard EBNF optional elements are placed inside square brackets);
- The '*' sign marks an element that can appear zero or more times (in the standard EBNF these elements are placed inside curly brackets);
- The '+' sign marks an element that can appear one or more times;
- In a rule, elements in a sequence are separated by spaces (the standard EBNF uses commas);
- A terminal could be either a quoted literal or a name of a token.

The first step involved in designing the grammar was to decide which language constructs should be matched by the lexer and which by the parser. In order to get a clear and concise parser specification, several rules recommended in [20] were followed:

- Recognise and discard anything in the lexer that the parser does not need to handle, e.g., whitespace and comments;
- Unite together into a single token in the lexer anything that the parser can treat as a single entity. For example, the `#define` directive is presented by the lexer as a single token `DEFINE` instead of two separate tokens (`'#'` and `'define'`);
- Unite together into a single token type those lexical structures that the parser does not need to distinguish. For example, function-like macro parameters could be numbers, identifiers or strings, but they are presented as single token type `PARAM`.

The grammar has tokens generated by the lexer as its terminal symbols. These tokens are produced using the lexer rules described further in Section 4.2. Table 3.2 presents the list of main tokens used in the grammar specification. The full list of tokens also includes several special symbols such as '!', '(', ')', '<', '>', '&', '|', '=', ',', etc. Whitespaces (spaces, tabs, line ends) and comments are ignored.

Table 3.2: LCFG grammar tokens

Token	Description
ID	Identifier
RESOURCE	Resource name (component/attribute pair)
RESOURCE_VALUE	Resource value (arbitrary string)
DEFINE	Directive `#define`
MACRO_BODY_TEXT	Macro body (replacement list)
PARAM	Parameter of a function
INCLUDE	Directive `#include`
FILENAME	Name of a file to be included
IFDEF	Directive `#ifdef`
IFNDEF	Directive `#ifndef`
IF	Directive `#if`
ELIF	Directive `#elif`
ELSE	Directive `#else`
ENDIF	Directive `#endif`
ERROR	Directive `#error`
WARNING	Directive `#warning`
EOF	End of source file

The specification for the proposed unified LCFG grammar is given in the next subsections as a set of EBNF grammar rules. Non-terminals begin with a lowercase letter. The terminal symbols are presented in UPPERCASE and **bold** font. The definition of a non-terminal is introduced by its name followed by a colon, and one or more right-hand alternatives on succeeding lines.

3.3.1 Statements

The start symbol of the grammar is `sourceFile` (Listing 3.2). It represents the source file, which consists of a list of statements. Each statement is terminated by a newline. The rule `statementList` groups statements together.

```
sourceFile
        : statementList EOF
        ;
statementList
        : statement*
        ;
/* Statements */
statement
        : resourceAssignment
        | resourceMutation
        | variableAssignment
        | conditionalStatement
        | functionDeclaration
        | functionCall
        | procedureCall
        | error
        | warning
        ;
```

Listing 3.2: Statements related rules

3.3.2 Resources

Resource related rules deal with assignment of configuration values to resources and their mutation (Listing 3.3).

```
/* Resource  assignment */
resourceAssignment
        : RESOURCE resourceValue?
        ;
resourceValue
        : RESOURCE_VALUE+
        ;
/* Resource  mutation */
resourceMutation
        : '!' RESOURCE resourceValue
        ;
```

Listing 3.3: Resource related rules

In its most commonly used form, the resource name has two parts: component name and attribute, separated by a dot. While the component name is an identifier, the attribute may contain the `'_'` symbol as a separator for tags and optional context specified in square brackets. In addition, in schema files, the component part is not present, and the attribute may define a meta-resource which begins with `'@'` and may contain a

placeholder (`'$'`). In order to simplify the parser specification, it was decided to match the above elements by the lexer and unite them in a single token called `RESOURCE`. The value of the token is further analysed and split into its constituents by the LCFG parser application.

The resource value is separated from the resource name by whitespace and is terminated by the end of line. The value may contain arbitrary fields: strings, object-like macro invocations, early and late references, tag lists, mutation function calls, etc., and can optionally be empty. Since the resource value does not obey any predefined format, it is matched by the lexer as a single token called `RESOURCE_VALUE`.

Unlike resource assignment, the resource value in mutation cannot be empty.

3.3.3 Variables

Variables represent object-like macros. Defining an object-like macro is an analogy of declaring a variable and optionally assigning a value to it. The variable name is identical to the macro name and the value is the macro body string (Listing 3.4). No expansion of the macro body takes place during the assignment. The **#undef** directive "deletes" the variable.

```
/* Variable  assignment */
variableAssignment
    : DEFINE ID variableValue?
    | UNDEF ID
    ;
variableValue
    : MACRO_BODY_TEXT+
    ;
```

Listing 3.4: Variable assignment rule

3.3.4 Functions

Functions correspond to function-like macros. The function name is identical to the macro name, and the function body is the macro body string. The parameters must be identifiers, separated by commas and optionally whitespace. When invoked, the function takes a list of actual parameters in parentheses, separated by commas, and replaces each use of a parameter in its body by the corresponding argument (Listing 3.5). A function parameter can be empty, but the commas cannot be omitted. Function body can contain resource assignments, but no variable/function declarations.

```
/* Function declaration */
functionDeclaration
    : DEFINE ID '(' formalParameterList? ')' functionBody?
    ;
formalParameterList
    : ID ( ',' ID )*
    ;
```

```
functionBody
    : MACRO_BODY_TEXT+
    ;
/* Function call */
functionCall
    : ID '(' parameterList ')'
    ;
parameterList
    : parameter? ( ',' parameter? )*
    ;
parameter
    : ( ID | PARAM )+
    ;
```

Listing 3.5: Function related rules

Functions can be "deleted" similarly to variables using the **#undef** directive.

3.3.5 Procedures

The inclusion of a file is equivalent to a procedure call. The name of the procedure is given by the name of the file to be included (Listing 3.6). Procedures do not have parameters. Executing a procedure means executing the statements contained in the included file.

```
/* Procedure call */
procedureCall
    : INCLUDE FILENAME ;
```

Listing 3.6: Procedure call rule

3.3.6 Conditionals

Conditionals evaluate a condition in order to execute or not execute a list of statements (Listing 3.7). While the **#ifdef** and **#ifndef** test if a macro (variable or function) is defined, the **#if** test the value of an expression. Conditionals could be nested.

```
/* Conditionals */
conditionalStatement
    : ifStatement ( elifStatament )* ( elseStatament )? endifLine
    ;
ifStatement
    : IFDEF ID statementList
    | IFNDEF ID statementList
    | IF expr statementList
    ;
elifStatament
    : ELIF expr statementList
    ;
elseStatament
```

```
    : ELSE statementList
    ;
endifLine
    : ENDIF ;
```

Listing 3.7: Conditionals rule

3.3.7 Expressions

As shown in Listing 3.8, expressions may contain:

- Variable name (`ID`). The value of the variable (macro body) is expanded before evaluation of the expression. If the variable is not defined, its value is zero (false);
- Integer and character constants (`PARAM`);
- Function call (`ID '('parameterList')'`). The value of the function (macro body) is expanded before evaluation of the expression. If the function is not defined, its value is zero (false);
- Comparison operators (<=, >=, <, >, ==, !=);
- Logical operators (`&&` and `||`);
- The `defined` operator, which check whether macros (variables or functions) are defined.

```
/* Expression */
expr
    : ID
    | PARAM
    | ID '(' parameterList ')'
    | '(' expr ')'
    | '!' expr
    | expr ( '<=' | '>=' | '<' | '>' ) expr
    | expr ( '==' | '!=' )  expr
    | expr ( '&&' | '||' )  expr
    | 'defined' ID
    | 'defined' '(' ID ')'
    ;
```

Listing 3.8: Expression rule

Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations and shifts, which are part of the C standard, were not revealed in the LCFG header files. Therefore, they were not included in the expression rule of the grammar.

The grammar specified in this section is the basis for the parser implementation presented in the next chapter.

Chapter 4

LCFG parser

This chapter describes the design and implementation of an LCFG lexer and parser and their further integration into an LCFG language application.

4.1 Tools

Java has been chosen as the implementation language for the project. The rationale behind the choice of Java was not only its popularity but also because it is platform independent. Moreover, although the LCFG tool primary focuses on UNIX-like systems, the LCFG language itself is designed to be processed on different platforms.

Parsers are usually generated from a grammar specification using parser generators rather than coded by hand. A parser generator converts the grammar into a program that recognises the grammar. Over the years, many parser generators for various language classes have been developed (a large list of notable generators can be found in [1]). A comprehensive list of Java-based parser generators along with their features is provided in [22]. A brief overview of the most popular tools that can generate parsers usable from Java is presented below.

CUP (**C**onstructor of **U**seful **P**arsers) [2] is a Java version of the classic YACC parser generator. CUP only generates a parser but may integrate with external lexical generators, e.g., JFlex [18]. The action code is embedded in the grammar specification; therefore, the latter may result in one huge file that is difficult to debug. Since CUP does not generate an abstract syntax tree, the programmer has to write the appropriate code to build nodes for every production alternative of the grammar.

JavaCC [3] is another widely used parser generator for Java. The grammar file, similarly to CUP, contains actions and all the custom code needed by the parser. JavaCC itself does not build a syntax tree, but it is combined with a tool called JJTree that does it.

ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) [19] is a popular tool in academic and industrial fields that can output parsers in many languages. The latest version

(ANTLR v4) automatically generates a parse tree and parse-tree walkers in the form of listener and visitor pattern implementations.

Although all of the aforementioned parser generators seemed suitable for generation of a parser for LCFG, ANTLR was chosen as the tool to be used, mainly for the following features:

- Integration of the lexer and the parser; lexer and parser rules can be defined in a single specification file;
- The application code is decoupled from grammar specification;
- Support for parse tree listener and visitor callback functions;
- Support for lexical modes and semantic predicates in lexer and parser rules;
- ANTLR provides a flexible testing tool called TestRig in its runtime library;
- The ANTLR documentation gives many examples of grammar specifications.

The subsequent sections present the steps carried out to build the LCFG parser:

1. Creating an ANTLR specification file containing the lexer rules for the unified LCFG language.
2. Creating an ANTLR specification file containing the parser rules.
3. Generating the lexer and parser using the ANTLR tool.
4. Integrating the lexer and parser into a language application that processes the syntax tree generated by the parser and composes the output XML profile.

4.2 Lexer rules specification

Lexical analysis is known to be the first phase of the language-processing pipeline for any language [7]. The lexer converts the stream of input characters into a sequence of tokens. This section provides details about the lexical rules composed to match and generate the set of tokens for the LCFG grammar. This set was defined previously in Table 3.2.

The basic syntax of a lexical rule in ANTR specification is:

```
<token_name> : <definition> ;
```

where the definition of the rule is specified using regular expressions.

ANTLR allows some frequently used regular expressions to be defined as fragments; these fragment rules do not result in tokens visible to the parser. The fragment rules defined in the LCFG lexer specification are shown in Listing 4.1.

```
fragment ATTRIBUTE:     (LETTER | '@') ~[\r\n\t ]* ;
fragment IDENT:         LETTER (LETTER | DIGIT)* ;
fragment LETTER:        [a-zA-Z_] ;
```

```
fragment DIGIT:          [0-9] ;
fragment DOT:            '.' ;
```

Listing 4.1: Fragment rules

While the ATTRIBUTE fragment is used in recognising the attribute part of resource names, the IDENT fragment matches resource names, macros (variables and function names) and formal parameters of functions.

Matching precisely the tokens of the unified LCFG grammar turned out to be somewhat complicated, mainly for two reasons. First, the LCFG source files contain a mix of LCFG and CPP formats. Secondly, the macro replacement text (macro body) and resource values can contain random text, which is a possible source of ambiguity.

Listing 4.2 illustrates some of the context-sensitive issues that were previously mentioned in Chapter 3. Although the syslog.addselector and local.debug are lexically identical, the first must be recognised as a RESOURCE token and the second as a RESOURCE_VALUE. The creent.org matches the RESOURCE token, but must be presented to the parser as a function parameter (PARAM). The keyword defined must be handled as a part of a macro body instead of being a single token.

```
syslog.addselector  local.debug
DNS_SLAVE_FWD_Z(creent,creent.org)
!dns.file_creent    mSET(zone.creent)
#error _NETMASK not defined
```

Listing 4.2: Examples of context-sensitive problems in a LCFG source file

The ANTLR documentation [20] suggests solving similar context-sensitive problems using lexical modes and/or semantic predicates. Lexical modes allow grouping of lexical rules by context. The lexer can return only those tokens that are matched by a rule specified in the current mode. The lexer switches back and forth between modes when it sees specific character sequences. Semantic predicates are Boolean expressions placed in a rule. When the lexer encounters a false predicate, it deactivates that rule.

The LCFG lexer rules were defined in a specification file named LCFGLexer.g4. Figure 4.1 shows the specified lexical modes and transitions between them.

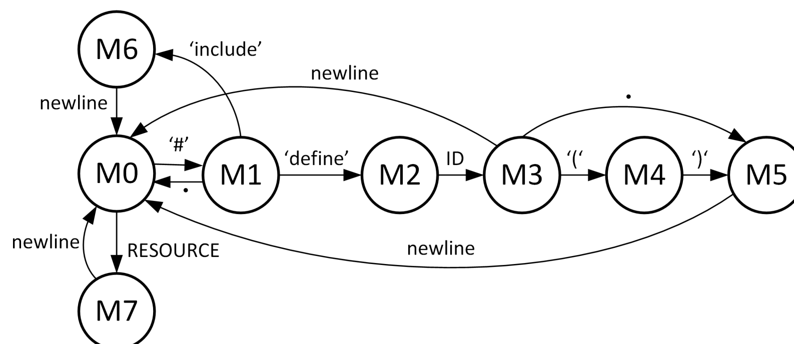

Figure 4.1: Lexical modes and transitions

The default mode is M0. In this mode the lexer recognises resource tokens (component/attribute pairs), mutation symbols, identifiers of variables/functions, parentheses, actual parameters of functions, comparison/logical operators, defined operator, and directive start symbol (`'#'`). In this state, the lexer matches but throws out the whitespaces, comments and backslash newlines.

M1 mode is entered on encountering the directive start symbol. In this mode, the lexer matches the CPP directives and outputs their tokens. Special cases are #include and #define. The #include enters mode M6, in which the character sequence until a newline forms the name of a header file. The #define triggers a transition to M2, in which the lexer only expects an identifier of a variable or a function. A transition from M3 to M0 means that no macro body for an object-like macro is provided. Mode M4 matches a parameter list of a function. Being in M5, the lexer matches the entire input until a newline and sends a MACRO_BODY_TEXT token (variable value or function body). In M7, the lexer captures everything until it sees a newline and sends a RESOURCE_VALUE token to the parser. In M5 and M7, the lexer skips backslash newlines but keeps spaces, tabs, and comments unchanged.

Listing 4.3 shows a fragment of the LCFG lexer specification, which demonstrates the use of semantic predicates. We define the nesting variable that is used to balance the brackets. The lexer will choose the rule RESOURCE only if the semantic predicate (nesting == 0) is true.

```
@lexer::members {
    int nesting = 0;
}
// Resource
RESOURCE: IDENT DOT ATTRIBUTE {nesting == 0}?
                                        -> mode(RESOURCE_VALUE_MODE);
// Parentheses
LP:     '('     { nesting++; } ;
RP:     ')'     { nesting--; } ;
```

Listing 4.3: Usage of semantic predicates

Since the resource pattern is only matched outside brackets, the above technique resolves ambiguities similar to that in:

```
DNS_SLAVE_FWD_Z(creent,creent.org)
```

The full specification for the LCFG lexer can be found in Appendix A.1.

4.3 Parser rules specification

The LCFG parser specification was composed in a file named LCFGParser.g4. The main parser rules specified in LCFGParser.g4 were previously described in Section 3.3. In addition, a small experimental extension of the LCFG language is provided. The latter extends the possibilities to modify resource values adding several mutation operators, proposed by Stephen Quinney at the beginning of the project:

- ‘?’ - mutate a resource value if already set;
- ‘~’ - mutate a resource value if not already set;
- ‘=’ - mutate a resource value and also make it immutable.

The final specification for the LCFG parser is given in Appendix A.2.

4.4 LCFG language application

The following ANTLR commands were used to generate the LCFG lexer and parser from their specifications:

```
antlr4 LCFGLexer.g4
antlr4 -visitor -no-listener LCFGParser.g4
```

The lexer and parser classes generated by ANTLR using the above commands are named `LCFGLexer` and `LCFGParser`, respectively.

As discussed previously, ANTLR technology allows grammars to be encapsulated from application code. The –`visitor` option tells ANTLR to generate a parse-tree visitor interface and a base class with default implementations for the visitor methods [20].

The generated lexer and parser were further integrated into an LCFG language application, whose architecture is shown in Figure 4.2.

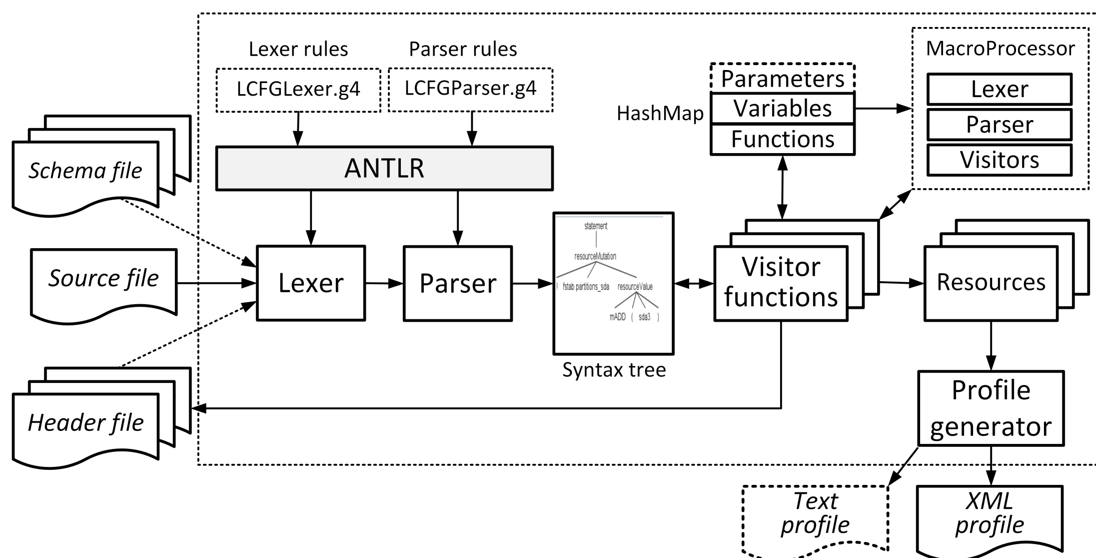

Figure 4.2: LCFG application architecture

This application takes the source file name as a command line argument and invokes the parser by the code shown in Listing 4.4. Once the parse tree is generated by the parser, it is processed using the ANTLR visitor mechanism. The latter performs a

depth-first walk of the parse tree and invokes user-defined visitor functions to access and process the nodes of the syntax tree [21].

```
// Create an input stream that reads from file
ANTLRInputStream input = new ANTLRInputStream(new FileInputStream(
    srcFile));
// Create a lexer that feeds off of input stream
LCFGLexer lexer = new LCFGLexer(input);
// Create a buffer of tokens pulled from the lexer
CommonTokenStream tokens = new CommonTokenStream(lexer);
// Create a parser that feeds off the tokens buffer
LCFGParser parser = new LCFGParser(tokens);
// Begin parsing at init rule (sourceFile)
ParseTree tree = parser.sourceFile();
// Create a visitor and visit the generated syntax tree
LCFGVisitor visitor = new LCFGVisitor(resourcesMap, includePaths);
visitor.visit(tree);
```

Listing 4.4: Invoking the LCFG parser

The application deals with four main classes of objects: `Variables`, `Functions`, `Procedures`, and `Resources`. A global hash map contains the declared variables and functions. This map is updated by visitor functions that process the parse tree in an interpreter-like fashion. Variable and function declarations add new items to the map. The `#undef` directive removes an item from the map. When a function is called, its parameters become a part of the global map but are removed after the execution. The execution of a function is performed by expanding its body and replacing the formal parameters with the values of the actual parameters. On encountering a procedure call (header file inclusion), the corresponding visitor function creates a new copy of the lexer/parser classes and feeds the lexer with the included file. When a variable/function is referenced, its value (macro body) is processed by a sub-parser (`MacroProcessor`). The `MacroProcessor` expands the macro body by recursively replacing the identifiers with their values from the global map. A second global map contains the resources being declared. Each resource assignment adds a new resource to the map. Finally, the XML generator outputs the configuration profile in XML or text format.

4.4.1 Object classes

This subsection describes the main object classes used by the LCFG language application: `Variables`, `Functions`, `Procedures` and `Resources`.

As discussed previously, variables correspond to object-like macros. When declared, a variable takes its name from the macro name and its body from the macro body (the macro body can be empty). A class called `Variable` was created to encapsulate the concept of a variable. The variable value is of `String` type. A method called `getExpandedValue()` is provided to expand this value (macro body) by using the `MacroProcessor`.

A class called `Function` was composed to represents function-like macros. In addition to the function name and its body, an object of this class stores the list of formal pa-

rameters for the function. A function can be called by invoking its `execute()` method. The execution is performed in two steps. First, the function body is expanded using the `MacroProcessor`; when the body is expanded, each use of a formal parameter in it is replaced by the corresponding actual parameter. In the second step, a new instance of the LCFG parser is created to process the expanded body.

There is a global `HashMap` structure created to store all the defined `Variable` and `Function` objects. This map is accessed by the `MacroProcessor` in macro resolving operation. Function parameters could be treated as variables: the formal parameter is the variable, and the string representing the actual parameter is its value. When a function is called, its formal and actual parameter pairs are added to the global map. After execution, these pairs should be removed from the map. Since the function calls could be nested, it is convenient to define a stack of maps: the first element of the stack is the map of globally defined `Variable` and `Function` objects, and the next elements contain the maps of parameters for the functions that have been called but not finished yet. Figure 4.3 shows the class diagram for the `Variable`, `Function` and the global stack of maps called `DefinedObjectsStack`. This class diagram is generated from the Java code of the project using the `ObjectAid UML Explorer` tool [5].

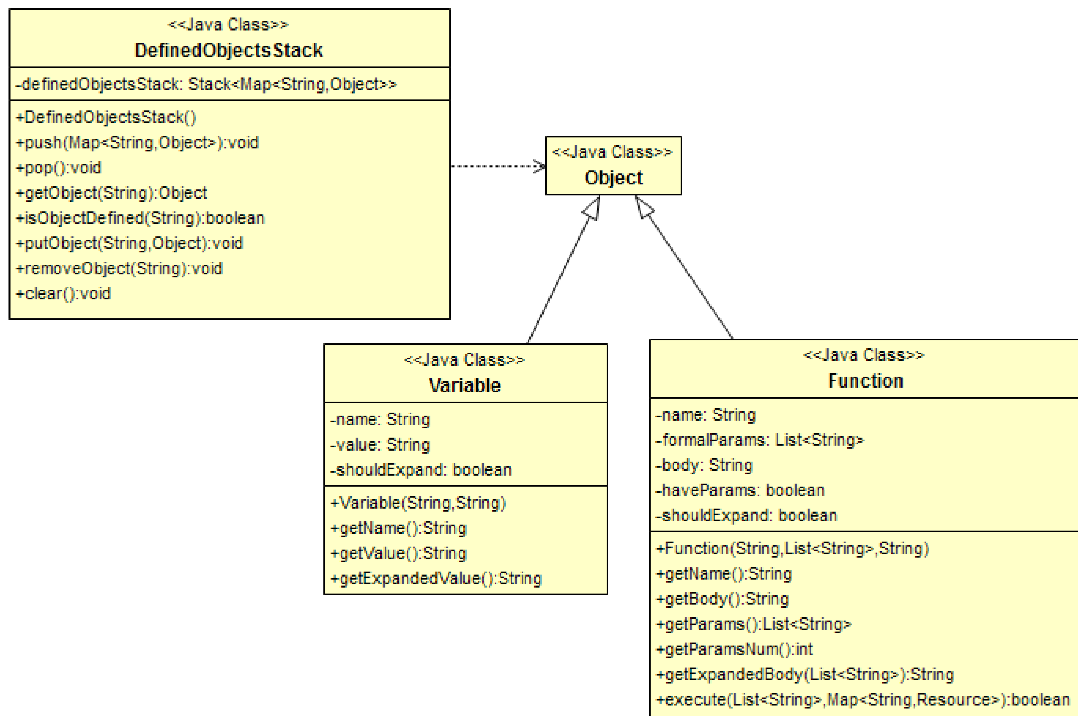

Figure 4.3: Class diagram for the `Variable`, `Function` and global stack of maps

As stated before, the inclusion of a header file is equivalent to a procedure call. The constructor of the `Procedure` class takes the name of the header file and a list of header search paths as parameters. The procedure `execute()` method first searches for the file in the provided include paths, and then creates a new copy of the LCFG lexer and parser classes to process the included file. This approach allows for automatically handling nested header files. The class diagram for the `Procedure` class is shown in Figure 4.4.

The additional parameter `resourcesMap` passed to the constructor specifies the map of resources to be updated by the invoked LCFG parser.

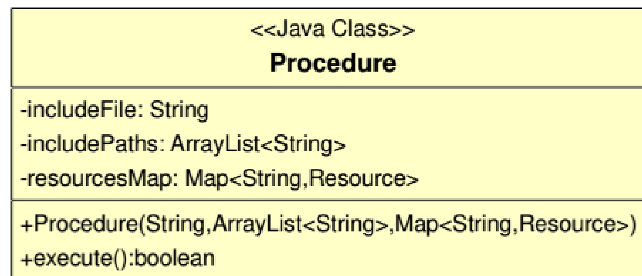

Figure 4.4: Class diagram for the `Procedure` class

Every configuration resource collected from the parse tree is stored in an object of the `Resource` class (Figure 4.5). A global `HashMap` called `resourcesMap`, being an attribute of the `LCFGVisitor` class, holds the defined resources. The component/attribute parts of the resource name can be accessed individually. Both the resource name and value can contain macros that should be expanded. The `expandName()` and `expandValue()` methods are provided for this purpose. The `replaceEarlyReferences()` and `replaceLateReferences()` methods deal with early references and late references, respectively. The `MutationFunc` enumeration type specifies the set of functions implemented to mutate the value of the resource.

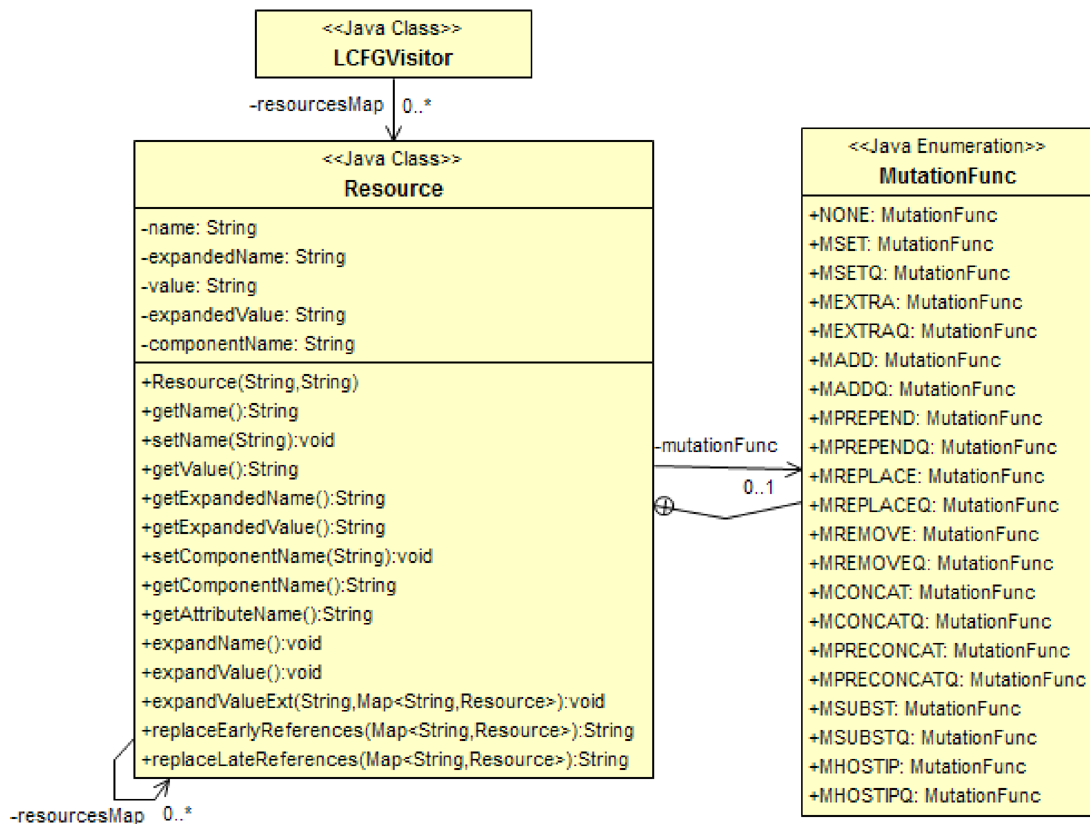

Figure 4.5: Class diagram for the `Resource` class

4.4.2 Visitor functions

Using the –`visitor` option, ANTLR generates a parse–tree visitor interface called `LCFGParserVisitor` and a base class (`LCFGParserBaseVisitor`). The interface contains `visit()` method for each rule of the grammar. The base class contains a default implementation of each `visit()` method. The default implementation of methods calls `visit()` on all children [20].

To visit the parse tree, a class called `LCFGVisitor` was created. It is a subclass of the auto-generated `LCFGParserBaseVisitor` class. The `LCFGVisitor` class takes a map of resources to be defined and a list of header search paths as parameters. As shown in the UML class diagram from Figure 4.6, the `LCFGVisitor` class implements methods that get called upon visiting of the non-terminal nodes in the parse tree. These methods override the default implementation of the visiting methods in the base class and perform actions to gather the LCFG resources and configuration parameters from the nodes of the parse tree.

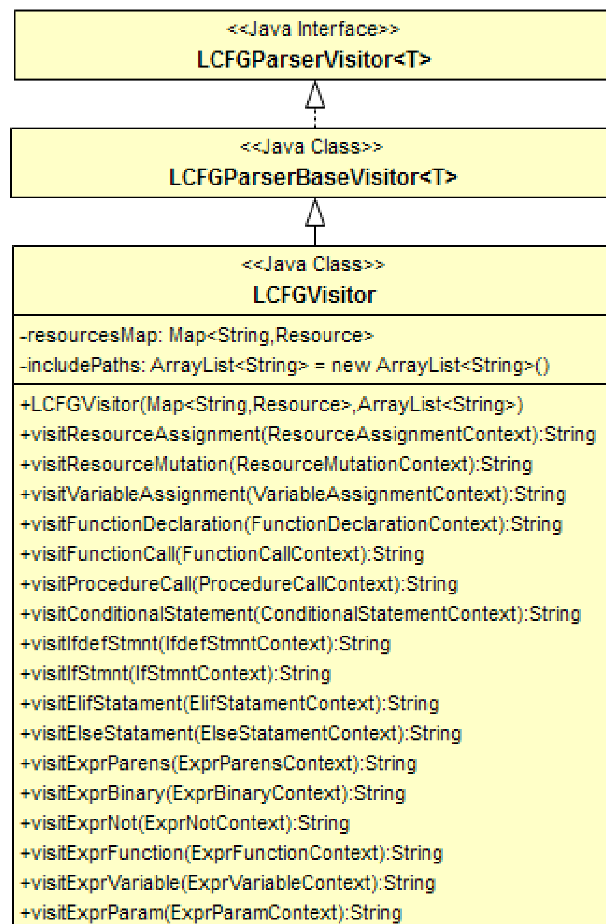

Figure 4.6: Class diagram for visitor classes

The purpose of the methods of the `LCFGVisitor` class is as follows:

- `visitResourceAssignment()`: Create a new `Resource` object, expand both its name and value using the `MacroProcessor`, and put the object into the global

resources map. At this point, all the early references found are resolved.

- `visitResourceMutation()`: Mutate the value of a resource that is already present in the global resources map. In general, the mutation could be performed using the predefined macros specified in `mutate.h` (the method first checks for these macros), but the mutation value can be also any arbitrary string.
- `visitVariableAssignment()`: Create a new `Variable` object and put it into the map of globally defined objects. The variable value is not macro-expanded at this stage, because it may depend on other macros that could be re-defined by the next usage of the variable.
- `visitFunctionDeclaration()`: Create a new `Function` object and put it into the map of globally defined objects. The function body and its formal parameters are not macro-expanded at this stage.
- `visitFunctionCall()`: Execute a function found in the map of globally defined objects. First, the function body is expanded using the `MacroProcessor`, then the function `execute()` method is invoked.
- `visitProcedureCall()`: Create a new `Procedure` object and include a header file by calling the procedure `execute()` method. Included files could in turn be nested, which is analogy of nested procedure calls.
- `visitIfdefStmnt()`, `visitIfStmnt()`, `visitElifStatament()`: Test the expression in its context, and if it evaluates to true, visit the statements listed in the context.
- `visitElseStatament()`: Visit the statements listed in its context.
- `visitExpr()` functions: Evaluate the expression in its context to `true` or `false`. The `visitExprDefined()` function evaluates to `true` if the identifier in its context is currently defined in the map of globally defined objects.
- `visitError()`, `visitWarning()`: Output the text in its context.

An example of an overridden visitor function is shown in Listing 4.5. It presents the implementation of the `visitProcedureCall()` method associated with the procedure call rule.

```
@Override
public String visitProcedureCall(LCFGParser.ProcedureCallContext ctx
    ) {
  String includeFile = ctx.FILENAME().getText();
  Procedure procedure = new Procedure(includeFile, includePaths,
      resourcesMap);
  if (procedure.execute() == true) {
    System.out.println("File " + includeFile + " included");
    return Boolean.toString(true);
  }
  return Boolean.toString(false);
}
```

Listing 4.5: Implementation of `visitProcedureCall()` method

4.4.3 Macro processor

The `MacroProcessor` is an important module of the LCFG application created for resolving macros. It is intended to process strings, rather than files. The `MacroProcessor` takes a string as an input, recognises and expands the macro expressions placed in it, and returns the expanded string.

Four kinds of strings in LCFG source files may refer to pre-defined macros: variable values, function bodies, resource names, and resource values. The resource names can only contain object-like macro names (variable names), while the other kinds of strings may invoke function-like macros (function calls) as well.

Two ideas on how macros should be resolved were considered. The first one was to simply split the input string by Java code using regular expressions that match identifiers and functions, and then to replace the identified macros with their definitions retrieved from the global map of defined objects. The attempt to implement this approach was not successful: it resulted in cumbersome code and difficulties in resolving function-like and nested macros.

The second idea was to run a new copy of a lexer-parser pair on encountering a macro in the input string and thus recursively expand macros until all of them are resolved. A similar approach was already applied for handling nested header files. However, using directly the `LCFGLexer`/`LCFGParser` classes for handling macro expansions turned out to be impractical for several reasons:

- The `LCFGLexer` removes whitespaces, while they should remain intact in the macro body being expanded.
- The `LCFGParser` has a number of grammar rules that are unnecessary in this context. More specifically, a macro body may contain variable names and function calls, but no variable assignments, function declarations, procedure calls, conditionals, and expressions. Furthermore, the `LCFGParser` has no grammar rule to handle isolated variable names.
- The function call visitor function in the `LCFGParser` executes a function, while in this context it should only replace the parameters and return the expanded body of the function.

Because of the considerations mentioned above, a decision was made to create a separate grammar and a parser which is only responsible for resolving macros. As shown in Listing 4.6, the defined macro-processing grammar uses a small subset of the previously described LCFG grammar. The grammar handles identifiers and function call patterns, which would probably be macro invocations. A rule called `otherText` at the end matches everything else in the input string that should remain unchanged.

```
macro
    : body* EOF ;
body
    : otherText
    | functionCall
    | variableId
```

```
    ;
variableId
    : ID ;
functionCall
    : ID '(' parameterList ')' ;
parameterList
    : parameter? ( COMMA parameter? )*
    ;
parameter
    : (ID | PARAM)+ ;
otherText
    : OTHER_TEXT+ ;
```

Listing 4.6: Grammar for the `MacroProcessor`

The final specification for the LCFG `MacroProcessor` can be found in Appendix A.3.

Having the above grammar defined in a file named `MacroProcessor.g4`, the parser was generated by calling the following ANTLR command:

```
antlr4 -visitor -no-listener MacroProcessor.g4
```

The above command auto-generates a lexer (`MacroProcessorLexer`), a parser (`MacroProcessorParser`), a visitor interface (`MacroProcessorVisitor`), and a base visitor class (`MacroProcessorBaseVisitor`).

The class implementing the `MacroProcessorVisitor` interface within the macro processor was named `MacroVisitor`. The actual macro processing is carried out by the following overridden visitor methods:

- `visitMacro()`: This method is called on visiting the starting node of the generated parse tree.
- `visitVariableId()`: This method is called on encountering an identifier in the input string. The method checks if the identifier matches a name of an existing variable in the map of globally defined objects. If so, the method returns the expanded value of the variable, otherwise, the result is the unchanged identifier string itself.
- `visitFunctionCall()`: This method is invoked on encountering a function call in the input string. The method checks if the identifier matches a name of an existing function in the map of globally defined objects. If so, the method visits the children of its `parameterList` context and collects the actual parameters. Their number must match the number of parameters in the function definition. If the above conditions are met, the result is the expanded body of the function, where each use of a parameter in its body is substituted by its corresponding actual value. Otherwise, the method returns the original string.
- `visitParameter()`: This method is called on visiting nodes containing actual parameters of a function. It handles the parameter string similarly to `visitVariableId()`. This allows all actual parameters to a function to be completely macro-expanded before they are replaced into the function body.

- `visitOtherText()`: The method simply returns the original string extracted from the context of its corresponding node.

The final result of a macro expansion is produced by the `visitMacro()` method, which merges the expanded strings returned by visitor functions called on its children nodes (Listing 4.7).

```
@Override
public String visitMacro(MacroProcessorParser.MacroContext ctx) {
StringBuilder result = new StringBuilder();
    for (MacroProcessorParser.BodyContext body : ctx.body()) {
        result.append(visit(body));
    }
    return result.toString();
}
```

Listing 4.7: Implementation of `visitMacro()` method

All macro definitions are checked for more macros to replace. As was shown in Figure 4.3, both `Variable` and `Function` classes provide methods to expand their assigned macro bodies: `getExpandedValue()` and `getExpandedBody()`, respectively. To be able to recursively resolve nested macros, each of these methods invokes its own copy of the `MacroProcessor` parser on the string to be expanded.

To illustrate the work of the macro processing mechanism described above, let us examine in more detail the steps used to process the example LCFG source code shown in Listing 4.8.

```
#define HOST    test
#define NAME    HOST.inf.ed.ac.uk

dhclient.hostname   NAME
```

Listing 4.8: Nested macros in a LCFG code fragment

1. The code fragment is processed by the LCFG parser. Figure 4.7 presents a graphical representation of the resulting parse tree.

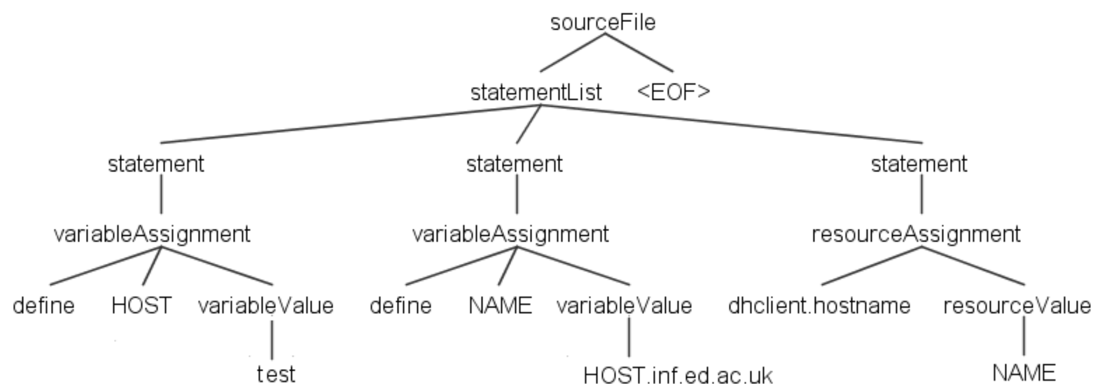

Figure 4.7: Parse tree for the code from Listing 4.8

2. The nodes of the parse tree are visited by the corresponding methods of the `LCFGVisitor` class. Specifically, the `visitVariableAssignment()` function invoked on the first two statements adds two variables to the map of globally defined objects:

```
Variable1: Name=HOST, Value=test
Variable2: Name=NAME, Value=HOST.inf.ed.ac.uk
```

3. The `visitResourceAssignment()` function invoked on the third statement declares a resource:

```
Resource: Name=dhclient.hostname, Value=NAME
```

   At this point, the resource value is to be expanded. The `Resource` class provides a method `expandValue()` for this purpose. This method creates and starts a `MacroProcessor` parser to process the resource value string, which produces the parse tree depicted in Figure 4.8.

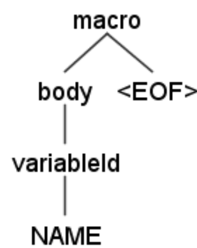

Figure 4.8: Parse tree for the resource value (string `NAME`)

4. Since the `visitVariableId()` method of the started `MacroProcessor` finds the identifier NAME in the map of globally defined objects, it calls the `getExpandedValue()` method of `Variable2`. The latter starts a new (second) `MacroProcessor` parser to process the variable value: string `HOST.inf.ed.ac.uk`.

5. The resulting parse tree of the second `MacroProcessor` (Figure 4.9) is visited. Only the first out of the five identifiers is found in the map of globally defined objects, and the `getExpandedValue()` method of `Variable1` is called.

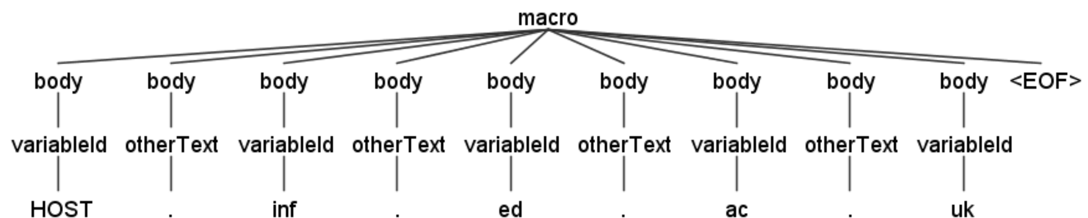

Figure 4.9: Parse tree for the string `HOST.inf.ed.ac.uk`

6. The `getExpandedValue()` method of `Variable1` starts a new (third) instance of the `MacroProcessor` to process its value: string `test`. The resulting parse tree

(not shown) is similar to that from Figure 4.8. The variable identifier `test` is not found in the map of globally defined objects, and therefore returned as a result by the `visitMacro()` method of the third instance of the `MacroProcessor`.

7. The `visitMacro()` method of the second `MacroProcessor` receives the resulting string (`test`) and continues visiting the remaining nodes from the parse tree in Figure 4.9. None of the other variable identifiers (`inf`, `ed`, `ac`, `uk`) is found to be a declared variable, so the final result merged and returned by the second `MacroProcessor` is the string `test.inf.ed.ac.uk`.

8. The `visitMacro()` method of the first `MacroProcessor` produces as a result the string `test.inf.ed.ac.uk`, which is returned to the `expandValue()` method invoked in step 3. Finally, the value of the resource `dhclient.hostname` is evaluated to `test.inf.ed.ac.uk`.

Since the macro body may contain arbitrary strings, it is important to check if the expanded string is syntactically correct. For example, consider the following code:

```
#define zones inf dcs
dns.zones   informatics
```

If this code were first processed by CPP, due to purely lexical substitutions, the resulting resource assignment would be:

```
dns.inf = dcs informatics
```

which is not what was expected.

For the above case, the LCFG application reports an error:

```
ERROR: Invalid attribute name (inf dcs) for resource: dns.inf dcs
```

Similarly, the following code:

```
#define components hardware file
components.schema   2

#define server !client
server.ack  true
```

would produce errors, too:

```
ERROR: Invalid component name (hardware file) for resource: hardware
    file.schema
ERROR: Invalid component name (!client) for resource: !client.ack
```

During evaluation, an example of self-referential macro (one whose name is present in its definition) was found in `Profiles/ert` source file:

```
#define ERT_COUNTER ERT_COUNTER/**/x
```

Since this macro produced infinite recursion in its expansion, the `MacroProcessor` was fixed to return such identifiers in macro bodies unchanged.

4.4.4 Profile generator

The purpose of this module is to generate the resulting profile. In its first phase it obtains default values for resources from schema files of components listed by `profile.components` resource. The analysis of schema files shows that they have the same format as the source files, with the following two differences:

- Resources are specified by only their attribute parts (the component part is presented in the name of the schema file).
- Each resource may have its respective meta-resource (prefixed with '`@`').

Rather than composing separate grammar specification, it was decided to extend the current one to parse schema files as well. This was worked out by a modification to the lexer specification. By adding an option named `parseDefaults` the lexer can now be switched between recognising resources from source or schema files, thus keeping the parser specification unchanged.

After the default resources have been added, all the `late references` are found and processed.

One purpose of meta-resources is to specify directives to the LCFG compiler. These are illustrated by example in Listing 4.9.

```
/* Publish to a spanning map */
@cluster %publish: rootmail
/* Subscribe to a spanning map */
@map %subscribe: clients
/* Validation */
poll = %string(interval): /^(\d+(h|m|s)(\+\d+(h|m|s))?)?$/
/* Ordering of components */
@components             ; $.ng_cforder
```

Listing 4.9: Examples of meta-resources

Spanning maps are used to refer to the value of a resource defined in an external profile file. Validation code, where provided, is executed in the context of the current compiler. The handling of the above directives by the LCFG application is a part of future work on this project. For example, validation code that obeys PERL syntax needs to be modified to match the regular expressions format used by Java. For now, all the resources are assumed to be of type `string` and no validation is performed when they are processed by the LCFG application.

The meta-resource fields considered by the LCFG application are the templates used to process tag list in a similar way as the original compiler. This is necessary in order to make the profile files comparable for evaluation and testing purposes.

To demonstrate the way of creating nested lists of records, let us consider the following example fragment taken from `dns-11.def`:

```
@zones      file_$ zone_$
zones
file_$
```

```
zone_$
```

Here, the meta-resource @zones defines two templates: file_$ and zone_$. These templates have a '$' as a placeholder for each tag value.

Next, suppose the following resources are defined in a source file:

```
dns.zones        inf dcs
dns.file_dcs     zone.dcs
dns.file_inf     zone.inf
dns.zone_dcs     dcs.ed.ac.uk
dns.zone_inf     inf.ed.ac.uk
```

Since the dns.zones has its corresponding meta-resource, it is handled as a tag list rather than an ordinary resource. The tags inf and dcs, replacing the placeholders in each of templates, are then used to identify the list elements. The resulting nested structure from the above example is:

```
dns
  zones
    inf
      file = zone.inf
      zone = inf.ed.ac.uk
    dcs
      file = zone.dcs
      zone = dcs.ed.ac.uk
```

The format of a profile file generated by the original compiler contains the resources to be configured on the host, along with an attached list of software packages to be installed. For now, the package lists composing mechanism used by the current compiler is not investigated in detail. Therefore, including package lists in the LCFG application output is left as a part of future work.

Three alternative formats of the resulting profile are provided, as specified in the next subsection.

4.4.5 Usage

From the console, the LCFG application can be invoked by executing the main function in the class LCFG. The arguments consist of:

- <source_file>: LCFG profile (source) file to be parsed
- [-**P**<profiles_path>]: Specifies a path for profile (source) files
- [-**I**include_paths>]: Specifies a list of include paths (separated by ';')
- [-**D**<defaults_path>]: Specifies a path for defaults (.def) files
- [-**V**]: Enable debug output messages
- [-**X**]: Enable XML output file generation

- [-**T**]: Select text output file generation

The arguments given in square brackets are optional. The source file to be parsed can be located in the path specified by –P option or in the current directory. The LCFG application looks for header files in the list of paths specified by –I command line option and in the current working directory. Schema (default) files can be located in the path specified by the –D option or in the current directory. The output file is placed in the folder out.

The default LCFG application output is a text file, whose format is identical to the so-called Simple format generated by the production LCFG compiler (mkxprof). The presence of a mandatory profile component is checked, and the generated profile contains only the components specified by the profile.components resource. The default resources for components are included from their corresponding schema files.

With the –X option specified, the LCFG application is intended to output an XML profile which is similar to the XML format of mkxprof.

The format specified by the option –T is provided mainly for testing purposes, where the source file may only encapsulate specific LCFG fragments. The output is a plain text file, which contains a list of resources sorted by their component names. It includes all the resources found regardless of their list specified in the profile.components resource. In addition, the presence of the profile component itself is not checked. This output format does not include resources from schema (.def) files.

Chapter 5

Evaluation

The evaluation of the proposed LCFG grammar and parser application was carried out in two stages. In the first stage, the testing was performed using representative fragments from LCFG source files that exemplify the main concepts of the grammar. The second stage was focused on testing the LCFG application against real configuration files.

The initial step was to perform basic tests to check if the proposed LCFG grammar is able to accept correct source code fragments. For this purpose, a set of example configuration files was composed. These test files cover the range of the rules of the unified LCFG grammar. First, each of these files was tested using the ANTLR test tool `TestRig` to make sure that the created parser generates correct parse trees. The graphical representations of the generated parse trees presented below were produced using the following command used to invoke the test tool:

```
grun LCFG sourceFile -gui <test_file>
```

Next, the profiles generated by the LCFG application executed on the testing examples were compared with the expected results. The following section presents some of the test cases provided to evaluate the main features of the proposed LCFG grammar and implemented parser. Since these examples do not include any real header and schema files, they were generated using the option `–T`. With this option set, the resulting output only contains resource values without expanding tag lists where present.

5.1 Basic tests

5.1.1 Resource assignment test

The first testing example, which is shown in Listing 5.1, is used to test the resource assignment rule of the grammar. It produces the parse tree shown in Figure 5.1.

```
/* Resource assignment testing example */
dns.type                  server
```

```
fstab.entries            proc pts shm sys
hardware.modconf         /etc/modprobe.conf
hardware.keytable
```

Listing 5.1: Resource assignment examples

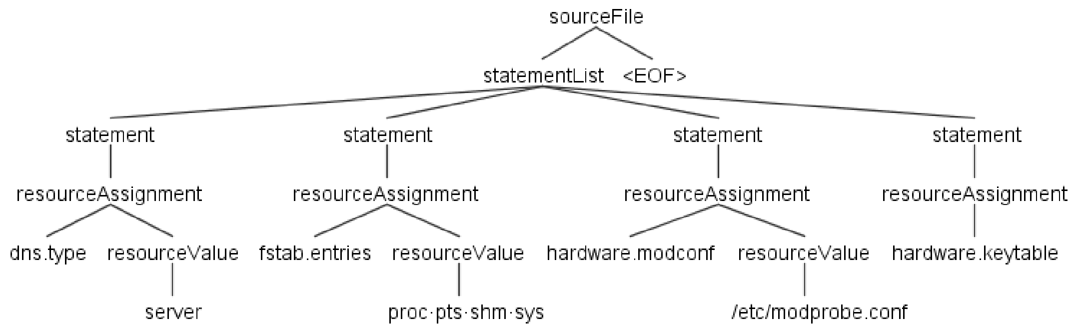

Figure 5.1: Parse tree for the testing example from Listing 5.1

The resulting output generated by the LCFG application is as expected:

```
dns.type = server
fstab.entries = proc pts shm sys
hardware.keytable =
hardware.modconf = /etc/modprobe.conf
```

5.1.2 Resource mutation test

The code shown in Listing 5.2 was composed to test the resource mutation rule. A fragment of its parse tree is depicted in Figure 5.2.

```
/* Resource mutation testing example */
fstab.partitions_sda     sda
!fstab.partitions_sda    mSET(sda1)
!fstab.partitions_sda    mADD(sda2)
!fstab.partitions_sda    mADD(sda3)
!fstab.partitions_sda    mREMOVE(sda2)
```

Listing 5.2: Resource mutation example

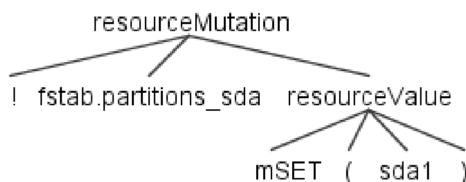

Figure 5.2: A fragment of the parse tree for the testing example from Listing 5.2

The sequence of mutations applied to the `fstab.partitions_sda` resource results in its expected final value (`sda1 sda3`), which is the correct LCFG application output:

```
fstab.partitions_sda = sda1 sda3
```

5.1.3 Variable assignment test

The test example from Listing 5.3 is used to test the variable assignment rule and nested macros in resource assignments. Figure 5.3 shows a part of its parse tree illustrating the `#define` and `#undef` directives.

```
/* Variable assignment and nested macros testing example */
#define SVN_HOST svntest
#define DICE_OPTIONS_SVN_NAME    SVN_HOST.inf.ed.ac.uk
apacheconf.vhostname_svn1        DICE_OPTIONS_SVN_NAME
#undef SVN_HOST
apacheconf.vhostname_svn2        DICE_OPTIONS_SVN_NAME
```

Listing 5.3: Variable assignment/nested macros testing example

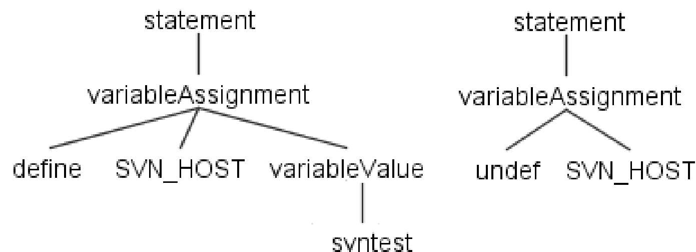

Figure 5.3: A fragment of the parse tree for variable assignments

Processing of the code from Listing 5.3 involves its handling by the `MacroProcessor`. The resulting resource values are:

```
apacheconf.vhostname_svn1 = svntest.inf.ed.ac.uk
apacheconf.vhostname_svn2 = SVN_HOST.inf.ed.ac.uk
```

Since the macro `SVN_HOST` is undefined before the resource assignment in the last line, it is not expanded in the value of `apacheconf.vhostname_svn2` resource.

5.1.4 Conditionals test

The `#ifdef` directive test was performed using the example from Listing 5.4.

```
/* #ifdef testing example */
#define LINUX_SL6
#ifdef LINUX_SL6
hardware.keytable_file  /etc/sysconfig/keyboard
#else
hardware.keytable_file  /* INTENTIONALLY EMPTY */
```

```
#endif
```

Listing 5.4: `#ifdef` testing example

Figure 5.4 shows the generated parse tree.

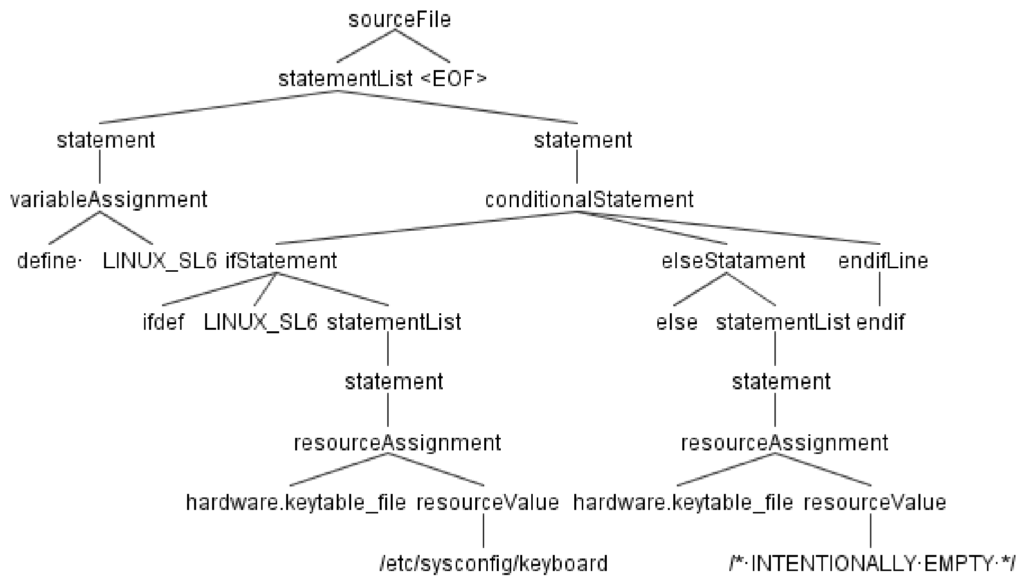

Figure 5.4: Parse tree for the testing example from Listing 5.4

Since the `LINUX_SL6` variable is initially defined, the resulting resource assignment is:

```
hardware.keytable_file = /etc/sysconfig/keyboard
```

Listing 5.5 presents the testing example for `#if`, `#elif`, `#else` conditionals.

```
/* #if, #elif, #else testing example */
#define LINUX_EL7
#define OS_RELEASE_MINOR        2
#if defined(LINUX_EL7) && OS_RELEASE_MINOR >= 2
network.netmanconf      /etc/NetworkManager/conf.d/99-lcfg.conf_1
#elif defined(LINUX_EL6)
network.netmanconf      /etc/NetworkManager/conf.d/99-lcfg.conf_2
#else
network.netmanconf      /etc/NetworkManager/conf.d/99-lcfg.conf_3
#endif
```

Listing 5.5: `#if`, `#elif`, `#else` testing example

As expected, the resulting resource assignment is:

```
network.netmanconf = /etc/NetworkManager/conf.d/99-lcfg.conf_1
```

5.1.5 Functions test

Listing 5.6 shows the code used to test function declarations and function calls. Function declarations make use of multi-line macro bodies, in which the resource assignment lines are separated by the special symbol ¢. This symbol is replaced by a new line (\n) by the visiting function for the function declaration rule.

```
/* Function (parameterised macro) testing example */
/* Function declarations */
#define NTP_SERVER(which) \
!ntp.serversPeers       mADD(which) ¢\
!ntp.name_/**/which     mSET(which.inf.ed.ac.uk)

#define ADD_SNMP_M2_VIEW(B,C) \
!snmp.views             mADD(B) ¢\
!snmp.viewComment_/**/B mSET(C) ¢\
!snmp.viewText_/**/B    mSET(view all included .1.3.6.1.2.1.B ff)

/* Function calls */
NTP_SERVER(ntp0)
ADD_SNMP_M2_VIEW(1,system)
```

Listing 5.6: Function testing example

Figure 5.5 shows a fragment of the parse tree illustrating the NTP_SERVER function declaration and its call.

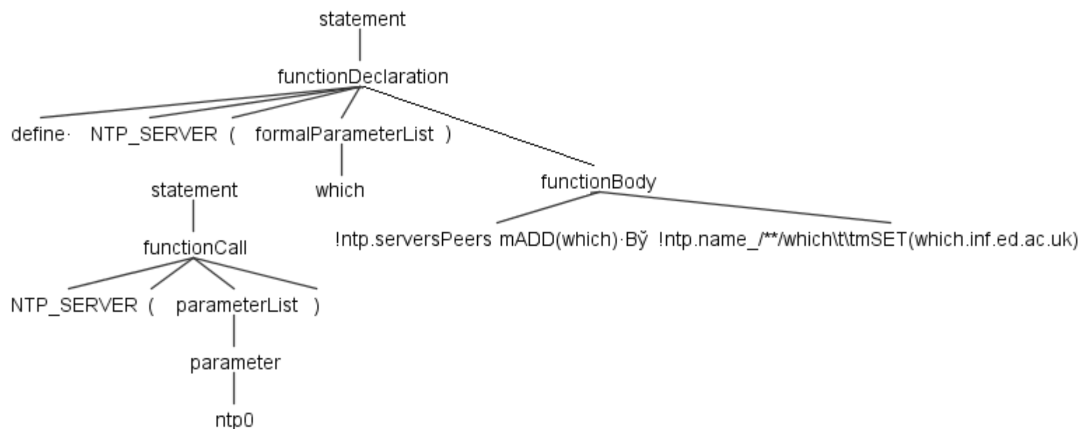

Figure 5.5: Parse tree for a function declaration and call

The resulting resources after expanding the function bodies by the MacroProcessor are:

```
ntp.name_ntp0 = ntp0.inf.ed.ac.uk
ntp.serversPeers = ntp0
snmp.viewText_1 = view all included .1.3.6.1.2.1.1 ff
snmp.viewComment_1 = system
snmp.views = 1
```

5.1.6 Procedure call test

This test was provided using the header file shown in Listing 5.7. As can be seen, it combines conditionals, resource/variable assignments, and function declaration.

```
/* File: example7.h */
/* Header file for procedure call testing example */
#ifndef _DNS_H
#define _DNS_H
#ifndef DNS_MASTER
#define DNS_MASTER    129.215.64.232
#endif

dns.type_default      slave

#define DNS_SLAVE_FWD(Z) \
!dns.zones            mADD(Z) ¢\
!dns.update_inf       mADD(Z) ¢\
!dns.update_all       mADD(Z) ¢\
!dns.zone_/**/Z       mSET(Z.ed.ac.uk) ¢\
!dns.file_/**/Z       mSET(zone.Z) ¢\
!dns.type_/**/Z       mSET(slave) ¢\
!dns.masters_/**/Z    mSET(DNS_MASTER)
#endif /* _DNS_H  */
```

Listing 5.7: Header file for procedure call testing example

This header file is then included using the procedure call concept in the source file as shown in Listing 5.8.

```
/* Procedure (file inclusion) testing example */
#include "example7.h"

dns.masters_default   DNS_MASTER
dns.local_netmask 255.255.255.0
DNS_SLAVE_FWD(cstr)
```

Listing 5.8: Source file for procedure call test

Figure 5.6 shows a fragment of the parse tree, illustrating the procedure call statement.

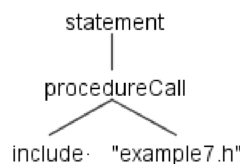

Figure 5.6: Parse tree for procedure call statement

As expected, the final output from the LCFG application combines the resources from both the source file and the included header file:

```
dns.type_default = slave
```

```
dns.file_cstr = zone.cstr
dns.masters_default = 129.215.64.232
dns.masters_cstr = 129.215.64.232
dns.local_netmask = 255.255.255.0
dns.update_inf = cstr
dns.update_all = cstr
dns.zones = cstr
dns.type_cstr = slave
dns.zone_cstr = cstr.ed.ac.uk
```

5.1.7 References test

Listing 5.9 shows the `late references` testing example.

```
/* Late references testing example */
hardware.modprobe_cmd       /sbin/modprobe
hardware.default_modloader  <%hardware.modprobe_cmd%>

inv.allocated john
auth.users <%inv.allocated%>
!inv.allocated mADD(jane)
```

Listing 5.9: `Late references` testing example

As was specified, resources that use `late references` map their values to the final values of the referenced resources. For example, the value of `auth.users` should be `john jane`, which is shown in the output generated by the LCFG application:

```
inv.allocated = john jane
auth.users = john jane
hardware.modprobe_cmd = /sbin/modprobe
hardware.default_modloader = /sbin/modprobe
```

An example for testing `early references` is shown in Listing 5.10.

```
/* Early references testing example */
hardware.modprobe_cmd       /sbin/modprobe1
hardware.default_modloader  <%%hardware.modprobe_cmd%%>
!hardware.modprobe_cmd      mSET(/sbin/modprobe2)

inv.allocated john
auth.users <%%inv.allocated%%>
!inv.allocated mADD(jane)
```

Listing 5.10: `Early references` testing example

Since `early references` by specification assign the current value of a resource, the value of `auth.users` should be `john`, which is confirmed by the LCFG application output:

```
inv.allocated = john jane
auth.users = john
```

```
hardware.modprobe_cmd = /sbin/modprobe2
hardware.default_modloader = /sbin/modprobe1
```

5.1.8 Extended mutation operators test

As was previously discussed in Section 4.3, an additional experimental set of mutation operators was implemented. Listing 5.11 shows the testing example composed to test and demonstrate this extension.

```
/* Extended mutation operators testing example */
/* Mutate a resource value if already set */
fstab.partitions_sda     sda
fstab.partitions_sdb
?fstab.partitions_sda    mSET(sda1)
?fstab.partitions_sdb    mSET(sdb1)

/* Mutate a resource value if not already set */
fstab.partitions_sdc     sdc
fstab.partitions_sdd
~fstab.partitions_sdc    mSET(sdc1)
~fstab.partitions_sdd    mSET(sdd1)

/* Mutate a resource value and also make it immutable */
fstab.partitions_sde     sde
=fstab.partitions_sde    mSET(sde1)
=fstab.partitions_sde    mSET(sde2)
```

Listing 5.11: Extended mutation operators testing example

The analysis of the LCFG application output generated from the testing example shows that the new operators work as intended. For example, the value of `fstab.partitions_sda` is mutated in contrast to `fstab.partitions_sdc`, and the value of `fstab.partitions_sde` has been mutated only once, since it has been made immutable from that point:

```
fstab.partitions_sda = sda1
fstab.partitions_sdb =
fstab.partitions_sdc = sdc
fstab.partitions_sdd = sdd1
fstab.partitions_sde = sde1
```

Another set of tests was performed on LCFG fragments that usually cause problems when handled by the C preprocessor. For example, the "problematic" declaration previously shown in Section 2.4:

```
#define DOMAIN inf.ed.ac.uk
mail.address `student' <student@DOMAIN>
```

is parsed correctly by both the LCFG parser and the `MacroProcessor`, which is illustrated by their corresponding parse trees shown in the a) and b) parts of Figure 5.7.

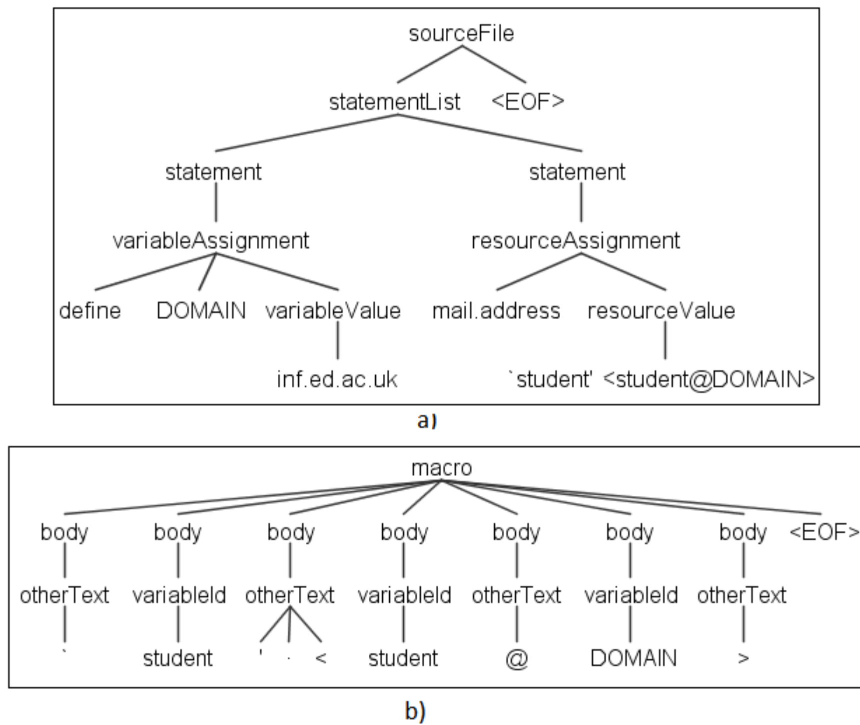

Figure 5.7: Parse trees for a "problematic" LCFG declaration

The resulting assignment is correct, too:

```
mail.address = `student' <student@inf.ed.ac.uk>
```

5.2 System tests

The next stage of the evaluation process was to generate a set of profiles from real configuration files and to validate them against the output of the production LCFG compiler (`mkxprof`). Although the LCFG system is currently used on DICE machines, its compiler is only running on dedicated servers as a daemon service. Therefore, it was unclear how to manually generate individual profiles on a particular host. It was decided to download the LCFG compiler package from the LCFG website [4] and to set it up to work on a particular DICE machine. After managing to run the LCFG compiler in standalone mode, a batch script was composed to automate the compilation of source files. For testing purposes, a set of 100 source files was manually selected.

After initial compilation, it became apparent that the production LCFG compiler does not publish individual profiles for so-called "self-managed" hosts, i.e. machines that do not automatically configure themselves (their files include `dice/os/selfmanaged.h` which in turn sets the output format to `stub`: `profile.format=stub`). In order to keep these profiles in the testing set, a small modification to force the `mkxprof` to generate these files was made in its PERL script.

The next question that came up was how to compare output files. The attempt to fix the LCFG application output to match exactly the `mkxprof`'s XML format was not successful. The reason was mainly the lack of detailed information on how `mkxprof` chooses unique names for individual elements that appear several times. In addition, `mkxprof` XML output may contain contexts and derivations that the implemented LCFG application does not handle. Luckily, it was found that `mkxprof` comes with a library called `Simple.pm` that can be used to generate trivial profile representations in plain text format.

In order to make the output files directly comparable, the profile generation module of the LCFG application was redesigned to match strictly the `mkxprof`'s trivial profile format. This involved building multilevel structures from tag lists using the corresponding templates from schema files, as well as sorting the output by components and resources. However, as it was previously mentioned in Section 4.4.4, this output does not contain an attached list of software packages to be installed on the target host. For this reason, package lists are not taken into account when comparing output profile files.

As a profile file usually contains thousands of lines, the comparison task could not be done manually. Therefore, an additional application, called `LCFGCompare`, was written to automate this process. Unlike conventional `diff` tools that compare files line by line, `LCFGCompare` compares resources previously grouped by components. For this purpose, `LCFGCompare` constructs a number of hash maps (one per component). The main idea behind this was to make the evaluation process more efficient by identifying and testing in isolation the components that possibly cause a large number of differences.

After the first bulk comparison of outputs, a big number of differences were observed. The investigation determined several sources of errors:

- Parsing errors, caused by wrong tokens recognised by the lexer.
- Bugs in parse tree visitor functions of the LCFG application.
- Problems with some of the LCFG source files themselves.

While the first two issues were cleaned up by fixes in the lexer grammar and Java code of the application, it is worth pointing out some fragments from source files that cause errors. The latter are discussed at the end of this section.

Based on the operating system header files included, the source files used for testing fall into three major groups:

- Files for hosts running Scientific Linux 6 (include `dice/os/sl6_64.h`).
- Files for hosts running Scientific Linux 7 (include `dice/os/sl7.h`).
- Files for other hosts (include `dice/os/selfmanaged.h`, `dice/os/printer.h`, `dice/os/managed-desktop.h`, etc.).

The results of testing the LCFG application output against the output of the production compiler (version 3.6.2) for each of the above-mentioned groups can be found

in Appendix B. Table 5.1 summarises the results for the three testing groups. Each row shows the average number of header files included per host, the average number of components and resources per file recognised by the LCFG application, and the average number of differences in resource values from the output of the production compiler.

Table 5.1: Average results for the three testing groups

Host OS	Source files	Headers per file	Components per file	Resources per file	Missing/ extra resources	Differences
SL6	36	405	51	2688	45/1	4
SL7	19	465	57	3038	0/1	3
Other	45	14	4	107	0/0	0

It can be seen that for the first two groups the comparing tool reports one extra resource in the LCFG application output. The reason for this is a missing by mistake comment terminating char in line #150 from `routing-4.def` schema file:

```
150 qzWqInterval     /* use the default *
151 qzWqMinRestart  /* use the default */
```

In this case, the CPP accepts the next line as a comment, too. As a result, after removing comments, the resource defined in line #151 (`qzWqMinRestart`) is not produced by `mkxprof`.

The investigation of missing resources shows that these are mainly resources imported from spanning maps, which are not supported by the LCFG application.

Another discrepancy between outputs is due to a specific function-like macro declaration found in `dice/options/theon-basecamp.h`:

```
125 #define BASECAMP_IFRIEND(U,D,G)        \
126 !pgluser.caps             mADD(G)      ¢\
127 !pgluser.users_/**/G      mADD(U@D)    ¢\
128 #ifdef DICE_OPTIONS_SUBVERSION_WEBDAV_SERVER ¢\
129 !subversion.authzmembers_theon_/**/G
130     mADD(U%D@FRIEND.INF.ED.AC.UK) ¢\
131 #endif
```

The issue with this code is that the proposed `MacroProcessor` grammar does not recognise conditionals within a macro body like this in line #128.

The next “problematic” fragment is from `dice/options/exam-desktop.h`:

```
181 file.tmpl_exampost       #!/bin/bash\n\
182 action=$1 # typically lock|unlock|hold\n\
183 change=$2 # typically 1|0 (course/paper changes\n
```

This fragment produces a parsing error due to a missing right parenthesis in line #183 (while the CPP does not report an error, the LCFG parser checks for balanced parentheses). To parse successfully source files that include `exam-desktop.h`, this parenthesis was added manually.

Another fragment to mention is from lines #307..#312 of the same `exam-desktop.h` file:

```
307 !webpic.css_uri mSETQ('data:text/css,.bottom{bottom:initial;font-
        weight:bolder;}
308 .polblock{color:#A00033;}
309 .pollog{color:darkgreen;}
310 .polreadonly{color:orangered;}
311 .polundefined{background-color:yellow;color:black;font-weight:bolder
        ;}
312 ')
```

While the CPP reports missing terminating quote character, this line is successfully parsed by the LCFG parser.

An interesting difference is due to line #15 from `lcfg/defaults/file.h`:

```
REGISTER_COMPONENT_WITH_SYSTEMD(file,lcfg-client.service, lcfg-auth.
    service,lcfgmultiuser)
```

The only distinction in this macro invocation is an extra space between its second and third parameters. It seems that `mkxprof` joins these parameters in resource assignments in its body, which could be seen from Figure 5.8. This figure presents a comparison window for host `cranston` generated by WinDiff comparison tool, where differences in `mkxprof`'s output are coloured in red and these in LCFG application output in yellow.

```
7489          wanted_units =
7490        lcfgfile
 7491  <!     after = lcfg-client.service lcfg-auth.service
(7491  !>     after = lcfg-auth.service
7492          before =
7493          conflicts =
7494          description = The LCFG file component
7495          environmentfile =
7496          execreload = /usr/bin/om file reload
7497          execrestart =
7498          execstart = /usr/bin/om file start
7499          execstop = /usr/bin/om file stop
7500          genopts
7501          link =
7502          linkdir = lib
7503          realname = lcfg-file.service
7504          required_units =
7505   <!     requires =
       !>     requires = lcfg-client.service
7506          servicetype = oneshot
7507          specopts
7508            rae
7509              specopt = RemainAfterExit=yes
```

Figure 5.8: An excerpt from WinDiff comparison window

Although the last testing results show minimal differences, there might still be LCFG

language constructs which would not be successfully recognised. This is a field of further detailed evaluation using the whole set of LCFG configuration files.

Chapter 6

Conclusion

This report presented the creation of an explicit grammar for LCFG and the design, implementation, and evaluation of a parser for the language. The main results of the work done could be summarised as follows:

- A unified grammar, combining the basic LCFG syntax with a subset of CPP directives, was proposed. The CPP subset supported by the grammar is sufficient to handle real source files actually being used in practice, and excludes the unnecessary features. The grammar was composed in such a way as to allow parsing the LCFG files without the need of the C preprocessor.
- Lexer and parser specifications for the grammar were developed. These specifications conform to the format supported by the ANTLR parser generator tool, which was next used to generate a lexer and a parser for the grammar.
- An LCFG language application was developed. This application processes the syntax tree generated by the parser in an interpreter-like fashion, collects the configuration parameters, and generates an individual output profile for a given host. For resolving macros, an additional sub-parser was created.
- A small set of additional operators for modifying resource values (mutation) was implemented and tested.
- The parser and the LCFG language application were evaluated using testing examples and real system configurations.

The biggest challenge in the project was specifying the lexer and parser rules for the unified grammar, and drawing the line between the lexer and the parser. The aim to simplify the parser rules and at the same time to "parse any LCFG syntax at any price" resulted in a very complex lexer specification, as can be seen from Appendix A.1. It could be said that the proposed grammar is an appropriate basis for a parser, but does not provide a good exposition of LCFG language details. For example, looking only at parser rules, it is not clear that the `RESOURCE` token represents a resource name composed of a component/attribute pair, and that the attribute might contain an optional tag. This also means that some constructs (for example tag lists) need to be further decomposed by the LCFG language application. A more detailed LCFG parser rules

specification is definitely a field for future experiments and improvements.

While an attempt was made for the LCFG application to cover as many of the features of the original compiler as possible, many of them are subject to further understanding, investigation, and integration. In particular, due to time constraints, features considered to be more specific to real deployment and not a crucial part of the parsing are left for future work. The supported features and limitations are outlined below.

Source and header files

Source and header files are processed correctly by the parser, generated from grammar specification. Resource assignments, both from source files and hierarchically included header files are fully supported. All the mutation functions currently included in `mutate.h` file are supported. The subset of CPP features used in the available configuration files is supported as well.

Schema files

Schema files are parsed successfully. Default resources from schema are correctly added to a generated profile. However, the set of directives included in meta-resources, which is used by the production compiler, is not handled by the LCFG application. These include directives for validation, publishing/subscribing to spanning maps, ordering, etc.

Tag lists

Tag lists are not included as separate entities in the grammar. Instead, they are handled by the profile generation module of the LCFG application, which builds multi-level list structures using the templates extracted from corresponding meta-resources.

Early and late references

Although references are not included as separate entities in the grammar, they are recognised and handled by the LCFG application while processing the syntax tree.

Contexts

Contexts are not included as separate entities in the grammar. They are recognised by the LCFG application, but resources containing contexts are not included in the generated profile.

Spanning maps

Spanning maps are not supported by the LCFG application.

Package lists

The list of packages is recognised as a value of the `profile.packages` resource, but is not processed by the LCFG application.

ANTLR can certainly be considered as the right choice for implementing the parser. Since the proposed grammar specifications are language independent, they could be used as a basis for writing alternative LCFG compiler implementations and analysis tools in other languages supported by ANTLR (C#, C++, Python, JavaScript, etc.).

Bibliography

[1] Comparison of parser generators. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_parser_generators. [Accessed: 2018-12-18].

[2] CUP. [Online]. Available: http://www2.cs.tum.edu/projects/cup/docs.php. [Accessed: 2018-12-20].

[3] JavaCC - The Java parser generator. [Online]. Available: https://javacc.org/. [Accessed: 2018-12-20].

[4] LCFG: A large scale UNIX configuration system. [Online]. Available: https://www.lcfg.org/. [Accessed: 2019-03-02].

[5] ObjectAid UML Explorer. [Online]. Available: https://www.objectaid.com/home. [Accessed: 2019-01-25].

[6] Using a C preprocessor as an HTML authoring tool. [Online]. Available: http://jkorpela.fi/html/cpre.html. [Accessed: 2019-02-10].

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[8] Paul Anderson. Towards a high-level machine configuration system. In *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pages 19–26, Berkeley, CA, September 1994. Usenix.

[9] Paul Anderson. *The Complete Guide to LCFG*, 2005.

[10] Paul Anderson. *System Configuration*, volume 14 of *Short Topics in System Administration*. SAGE, 2006.

[11] Paul Anderson. *LCFG: a Practical Tool for System Configuration*, volume 17 of *Short Topics in System Administration*. Usenix Association, 2008.

[12] Paul Anderson and Alastair Scobie. LCFG - the Next Generation. In *UKUUG Winter Conference*. UKUUG, 2002.

[13] Mark Burgess. Cfengine: a site configuration engine. *USENIX Computing Systems*, 8(3), 1994.

[14] Eelco Dolstra and Armijn Hemel. Purely functional system configuration management. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 13:1–13:6, Berkeley, CA, USA, 2007. USENIX Association.

[15] Jean-Marie Favre. CPP denotational semantics. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, 2003.

[16] Paul Gazzillo and Robert Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI 2012)*, 2012.

[17] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[18] Gerwin Klein. JFlex User‘s Manual. [Online]. Available: https://www.jflex.de/manual.html. [Accessed: 2018-12-28].

[19] Terence Parr. ANTLR. [Online]. Available: https://www.antlr.org/. [Accessed: 2018-12-22].

[20] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2 edition, 2013.

[21] Federico Tomassetti. The ANTLR mega tutorial. [Online]. Available: https://tomassetti.me/antlr-mega-tutorial/. [Accessed: 2019-01-18].

[22] Federico Tomassetti. Parsing in Java: all the tools and libraries you can use. [Online]. Available: https://tomassetti.me/parsing-in-java/. [Accessed: 2018-12-20].

Appendices

Appendix A

LCFG grammar specification

A.1 Lexer rules

```
/* LCFG lexer grammar */

lexer grammar LCFGLexer;

channels { COMMENTS_CHANNEL }

@lexer::members {
  private int nesting = 0;
  private boolean inDirective = false;
  public boolean parseDefaults = false;
}

// ---------------------------------------------------------------
// Default mode (M0)
// Resource
RESOURCE:   IDENT DOT ATTRIBUTE {nesting == 0 && parseDefaults ==
    false}? -> mode(RESOURCE_VALUE_MODE) ;
// Attribute in defaults (schema) file
ATTR:   ATTRIBUTE {nesting == 0 && parseDefaults == true && !
    inDirective}? -> type(RESOURCE), mode(RESOURCE_VALUE_MODE) ;
// Directive 'defined'
DEFINED:    'defined' [ \t]* ;
// Identifier
ID:     IDENT ;
// Mutation symbols
EXCL:   '!' ;
MUT1:   '?' {nesting == 0}? -> type(EXCL) ;
MUT2:   '~' {nesting == 0}? -> type(EXCL) ;
MUT3:   '=' {nesting == 0}? -> type(EXCL) ;

// Separators
LP:     '(' {nesting == 0 || inDirective}? {nesting++;} ;
RP:     ')' {nesting == 1 || inDirective}? {nesting--;} ;
COMMA:  ',' ;
// Comparison/logical operators
```

```
EQUAL:       '==' ;
NOTEQUAL:    '!=' ;
AND:         '&&' ;
OR:          '||' ;
LT:          '<' ;
GT:          '>' ;
LE:          '<=';
GE:          '>=';
// Directive start
SHARP:  '#' [ \t]* {nesting == 0}? {inDirective = true;} -> channel(
    HIDDEN), mode(DIRECTIVE_MODE) ;
// Comments and whitespaces
COMMENT:     '/*' .*? '*/'  -> channel(COMMENTS_CHANNEL) ;
WS:      [ \t]+  {nesting == 0 || inDirective}? -> channel(HIDDEN) ;
// Backslash newline
BS_NL:  '\\' '\r'? '\n' -> skip ;
// New line
NL:      '\r'? '\n'  {inDirective = false;} -> channel(HIDDEN) ;
// Constant as parameter
PARAM_CONST:     (INT | FLOAT)  -> type(PARAM) ;
// Reference start <% as parameter
PARAM_REFL:      '<%' {nesting > 0}? -> type(PARAM) ;
// Reference end %> as parameter
PARAM_REFR:      '%>' {nesting > 0}? -> type(PARAM) ;
// String as parameter
PARAM_STR1:      '"' (~('\\' | '"') | '\\' .)* '"' {nesting > 0}? ->
    type(PARAM) ;
PARAM_STR2:      '\'' (~('\\' | '\'') | '\\' .)* '\'' {nesting > 0}?
    -> type(PARAM) ;
// Text \n as parameter
PARAM_NL:    '\\' ~[\n] {nesting > 0}? -> type(PARAM) ;
// Whitespaces as parameter
PARAM_WS:    [ \t]+  {nesting > 0 && !inDirective}? -> type(PARAM) ;
// '(' as parameter
PARAM_LP:    '(' {nesting > 0}? {nesting++;} -> type(PARAM) ;
// ')' as parameter
PARAM_RP:    ')' {nesting > 1}? {nesting--;} -> type(PARAM) ;
// Parameter
PARAM:       ~[,\\]+? {nesting > 0}? ;
// Catchall Rule
ANY:         . ;

// ---------------------------------------------------------------
// Resource value mode (M7)
mode RESOURCE_VALUE_MODE ;
// Resource value - everything until a new line
VALUE_LP:        '(' {nesting++;}     -> type(RESOURCE_VALUE) ;
VALUE_RP:        ')' {nesting--;}     -> type(RESOURCE_VALUE) ;
VALUE_BS_NL:     '\\' '\r'? '\n'      -> skip;
VALUE_SL_ESC:    '\\' .               -> type(RESOURCE_VALUE);
VALUE_IGNORE_NEWLINE: '\r'? '\n' {nesting > 0}? -> skip ;
RESOURCE_VALUE: ~[\r\n()\\]+ ;
VALUE_END:        '\r'? '\n' {nesting == 0}? -> channel(HIDDEN), type
    (NL), mode(DEFAULT_MODE) ;

// ---------------------------------------------------------------
```

```
// Directives mode (M1)
mode DIRECTIVE_MODE ;

INCLUDE:    'include' [ \t]+  -> mode(INCLUDE_MODE) ;
DEFINE:     'define' [ \t]+ -> mode(DEFINE_MODE) ;
IF:         'if'    -> mode(DEFAULT_MODE) ;
ELIF:       'elif'  -> mode(DEFAULT_MODE) ;
ELSE:       'else'  -> mode(DEFAULT_MODE) ;
UNDEF:      'undef' -> mode(DEFAULT_MODE) ;
IFDEF:      'ifdef' -> mode(DEFAULT_MODE) ;
IFNDEF:     'ifndef'    -> mode(DEFAULT_MODE) ;
ERROR:      'error' [ \t]+    -> mode(MACRO_BODY_MODE) ;
WARNING:    'warning' [ \t]+  -> mode(MACRO_BODY_MODE) ;
ENDIF:      'endif' -> mode(DEFAULT_MODE) ;

// ---------------------------------------------------------------
// Include mode (M6)
mode INCLUDE_MODE ;
INCLUDE_NL:     '\r'? '\n' {inDirective = false;} -> channel(HIDDEN)
    , mode(DEFAULT_MODE);
FILENAME:       ~[\r\n]+ ;

// ---------------------------------------------------------------
// Macro declaration mode (M2)
mode DEFINE_MODE ;
MACRO_NAME:    IDENT  -> type(ID), mode(MACRO_NAME_MODE) ;

// ---------------------------------------------------------------
// Macro name mode (M3)
mode MACRO_NAME_MODE;
MACRO_NAME_LP:      '('  -> type(LP), mode(MACRO_PARAM_MODE) ;
MACRO_NAME_WS:      [ \t]+ -> channel(HIDDEN), type(WS), mode(
    MACRO_BODY_MODE) ;
MACRO_NAME_BS_NL:   '\\' '\r'? '\n' -> skip ;
MACRO_MANE_NL:      '\r'? '\n' {inDirective = false;} -> channel(
    HIDDEN), mode(DEFAULT_MODE) ;
MACRO_MANE_COMMENT: '/*' .*? '*/' -> channel(COMMENTS_CHANNEL) ;
MACRO_NAME_OTHER: . -> type(MACRO_BODY_TEXT),mode(MACRO_BODY_MODE) ;

// ---------------------------------------------------------------
// Macro params mode (M4)
mode MACRO_PARAM_MODE;
MACRO_PARAM_ID:     IDENT  -> type(ID) ;
MACRO_PARAM_COMMA:  ','  -> type(COMMA) ;
MACRO_PARAM_RP:     ')'    -> type(RP), mode(MACRO_BODY_MODE) ;
MACRO_PARAM_WS:     [ \t]+   -> channel(HIDDEN), type(WS) ;
MACRO_PARAM_COMMENT: '/*' .*? '*/'  -> channel(COMMENTS_CHANNEL) ;
MACRO_PARAM_BS_NL:  '\\' '\r'? '\n' -> channel(HIDDEN) ;

// ---------------------------------------------------------------
// Macro body mode (M5)
mode MACRO_BODY_MODE;
MACRO_BODY_BS_NL:   '\\' '\r'? '\n' -> channel(HIDDEN);
MACRO_BODY_SL_ESC:  '\\' .   -> type(MACRO_BODY_TEXT);
MACRO_BODY_NL:      '\r'? '\n'  {inDirective = false;} -> channel(
    HIDDEN), mode(DEFAULT_MODE);
```

```
MACRO_BODY_TEXT:    ~[\r\n\\]+ ;

// -------------------------------------------------------------
// Fragments
fragment ATTRIBUTE:  (LETTER | '@') ~[\r\n\t ]* ;
fragment IDENT:   LETTER (LETTER | DIGIT)* ;
fragment LETTER:  [a-zA-Z_] ;
fragment DIGIT:   [0-9] ;
fragment INT: DIGIT+ ;
fragment FLOAT:   INT ('.' INT)? ;
fragment DOT: '.' ;
```

A.2 Parser rules

```
/* LCFG parser grammar */

parser grammar LCFGParser;

options { tokenVocab=LCFGLexer; }

@parser::members {
  public static boolean debug = false;
}

sourceFile
    : statementList EOF
    ;

statementList
    : statement*
    ;

statement
    : resourceAssignment
    | resourceMutation
    | variableAssignment
    | functionDeclaration
    | functionCall
    | procedureCall
    | conditionalStatement
    | error
    | warning
    ;

resourceAssignment
    : RESOURCE resourceValue?
    ;
resourceValue
    : RESOURCE_VALUE+
    ;
resourceMutation
    : EXCL RESOURCE resourceValue
```

```
    ;

variableAssignment
    : DEFINE ID variableValue?
    | UNDEF ID
    ;
variableValue
    : MACRO_BODY_TEXT+
    ;

functionDeclaration
    : DEFINE ID LP formalParameterList? RP functionBody?
    ;
formalParameterList
    : ID (COMMA ID)*
    ;
functionBody
    : MACRO_BODY_TEXT+
    ;
functionCall
    : ID LP parameterList RP
    ;
parameterList
    : parameter? (COMMA parameter?)*
    ;
parameter
    : (ID | PARAM)+
    ;

procedureCall
    : INCLUDE FILENAME
    ;

conditionalStatement
    : ifStatement ( elifStatament )* ( elseStatament )? endifLine
    ;

ifStatement
    : IFDEF ID statementList    # ifdefStmnt
    | IFNDEF ID statementList   # ifdefStmnt
    | IF expr statementList     # ifStmnt
    ;
elifStatament
    : ELIF expr statementList
    ;
elseStatament
    : ELSE statementList
    ;
endifLine
    : ENDIF
    ;

expr
    : ID                            #exprVariable // Variable
    | PARAM                         #exprParam    // Parameter
    | ID LP parameterList RP        #exprFunction // Function call
```

```
    | LP expr RP                       #exprParens    // Parenthesis
    | EXCL expr                        #exprNot       // Not
    | left=expr op=(LT | GT | LE | GE) right=expr #exprBinary
    | left=expr op=(EQUAL | NOTEQUAL) right=expr  #exprBinary
    | left=expr op=(AND | OR) right=expr    #exprBinary // And/Or
    | DEFINED ID        #exprDefined // Directive: #defined
    | DEFINED LP ID RP  #exprDefined // Directive: #defined
    ;

error
    : ERROR txt
    ;

warning
    : WARNING txt
    ;

txt
    : MACRO_BODY_TEXT+
    ;
```

A.3 MacroProcessor rules

```
/* LCFG macro processor grammar */

grammar MacroProcessor;

@lexer::members {
  int nesting = 0;
}

// ---- Parser rules ---------------------------------
macro
    : body* EOF
    ;

body
    : otherText
    | functionCall
    | variableId
    ;

variableId
    : ID
    ;

functionCall
    : ID LP parameterList RP
    ;
parameterList
    : parameter? (COMMA parameter?)*
    ;
```

```
parameter
    : (ID | PARAM)+
    ;

otherText
    : OTHER_TEXT+
    | LP parameterList RP
    ;

// ---- Lexer rules ---------------------------------
// Identifier
ID:      IDENT ;
// Separators
LP:      '('    {nesting == 0}? {nesting++;} ;
RP:      ')'    {nesting == 1}? {nesting--;} ;
COMMA:   ','    {nesting == 1}? ;
// String as parameter
PARAM_STR: '"' (~('\\' | '"') | '\\' .)* '"' {nesting > 0}? -> type(
   PARAM) ;
PARAM_STR2: '\'' (~('\\' | '\'') | '\\' .)* '\'' {nesting > 0}? ->
   type(PARAM) ;
// Comments
COMMENT:    '/*' .*? '*/'    -> skip ; //channel(HIDDEN) ;
// Parameter
PARAM:       ~[,()]+ {nesting == 1}? ;
// Comma as parameter
PARAM_COMMA: ',' {nesting > 1}? -> type(PARAM) ;
// '(' as parameter
PARAM_LP:    '(' {nesting > 0}? {nesting++;} -> type(PARAM) ;
// ')' as parameter
PARAM_RP:    ')' {nesting > 1}? {nesting--;} -> type(PARAM) ;
// Other text as parameter
PARAM_OTHER_TEXT: . {nesting > 0}? -> type(PARAM) ;
// Other text
OTHER_TEXT:  . ;

fragment IDENT:          LETTER (LETTER | DIGIT)* ;
fragment LETTER:         [a-zA-Z_] ;
fragment DIGIT:          [0-9] ;
```

Appendix B

Evaluation results

B.1 Profiles comparison report (summary)

This report is generated by the WinDiff comparison tool.
`/Data/1` directory contains profiles generated by `mkxprof`.
`/Data/2` directory contains profiles generated by the LCFG application.

```
-- /Data/1 : /Data/2 -- includes identical,differing files
./4man/simple/profile   identical
./aciu/simple/profile   different (/Data/2 is more recent)
./altino/simple/profile different (/Data/2 is more recent)
./ard/simple/profile    identical
./ardeola/simple/profile        identical
./arezzo/simple/profile identical
./arta/simple/profile   different (/Data/2 is more recent)
./artist/simple/profile identical
./aruba/simple/profile  identical
./astonmartin/simple/profile    identical
./babel/simple/profile  different (/Data/2 is more recent)
./baked/simple/profile  identical
./bart/simple/profile   identical
./blackwell/simple/profile      different (/Data/2 is more recent)
./blogvm/simple/profile different (/Data/2 is more recent)
./bocian/simple/profile different (/Data/2 is more recent)
./bolt/simple/profile   different (/Data/2 is more recent)
./bombay/simple/profile identical
./caley/simple/profile  identical
./carnero/simple/profile        different (/Data/2 is more recent)
./caxton/simple/profile identical
./charon/simple/profile identical
./chatty/simple/profile different (/Data/2 is more recent)
./claise/simple/profile identical
./collins/simple/profile        different (/Data/2 is more recent)
./cowan/simple/profile  identical
./cranston/simple/profile       different (/Data/2 is more recent)
./djokovic/simple/profile       different (/Data/2 is more recent)
./eden/simple/profile   identical
```

```
./empoli/simple/profile identical
./ert/simple/profile    different (/Data/2 is more recent)
./farg/simple/profile   different (/Data/2 is more recent)
./fimo/simple/profile   different (/Data/2 is more recent)
./fox/simple/profile    identical
./gala/simple/profile   different (/Data/2 is more recent)
./garbo/simple/profile  identical
./hakone/simple/profile identical
./helotrix/simple/profile       identical
./hobgoblin/simple/profile      different (/Data/2 is more recent)
./holywood/simple/profile       different (/Data/1 is more recent)
./honda/simple/profile  different (/Data/2 is more recent)
./hopper/simple/profile identical
./hornet/simple/profile identical
./horton/simple/profile different (/Data/2 is more recent)
./icarus/simple/profile identical
./idoru/simple/profile  different (/Data/2 is more recent)
./innerwick/simple/profile      different (/Data/2 is more recent)
./ivon/simple/profile   different (/Data/2 is more recent)
./jerry/simple/profile  different (/Data/2 is more recent)
./kegan/simple/profile  different (/Data/2 is more recent)
./kelvin/simple/profile different (/Data/2 is more recent)
./lazar/simple/profile  different (/Data/2 is more recent)
./lindor/simple/profile different (/Data/2 is more recent)
./lion/simple/profile   identical
./luse/simple/profile   identical
./lychee/simple/profile identical
./lyng/simple/profile   identical
./maguire/simple/profile        different (/Data/2 is more recent)
./march/simple/profile  different (/Data/2 is more recent)
./modin/simple/profile  different (/Data/2 is more recent)
./mullo/simple/profile  different (/Data/2 is more recent)
./napa/simple/profile   identical
./neave/simple/profile  identical
./neeps/simple/profile  different (/Data/2 is more recent)
./nix/simple/profile    different (/Data/2 is more recent)
./nora/simple/profile   different (/Data/2 is more recent)
./oakwood/simple/profile        different (/Data/2 is more recent)
./otaka/simple/profile  different (/Data/2 is more recent)
./paeroa/simple/profile different (/Data/2 is more recent)
./pavarotti/simple/profile      identical
./pergamon/simple/profile       different (/Data/2 is more recent)
./pike/simple/profile   identical
./pollock/simple/profile        identical
./raven/simple/profile  different (/Data/2 is more recent)
./reeves/simple/profile different (/Data/2 is more recent)
./rocklin/simple/profile        identical
./salinas/simple/profile        identical
./sashko/simple/profile different (/Data/2 is more recent)
./scapa/simple/profile  identical
./selma/simple/profile  different (/Data/2 is more recent)
./shiel/simple/profile  identical
./skelp/simple/profile  different (/Data/2 is more recent)
./slatkin/simple/profile        different (/Data/2 is more recent)
./sobotka/simple/profile        identical
./sofia/simple/profile  different (/Data/2 is more recent)
```

```
./stoner/simple/profile different (/Data/2 is more recent)
./tarski/simple/profile identical
./titan/simple/profile  different (/Data/2 is more recent)
./turing/simple/profile different (/Data/2 is more recent)
./tycho/simple/profile  different (/Data/2 is more recent)
./velma/simple/profile  different (/Data/2 is more recent)
./vili/simple/profile   identical
./white/simple/profile  different (/Data/2 is more recent)
./yak/simple/profile    different (/Data/2 is more recent)
./yashin/simple/profile identical
./yukon/simple/profile  different (/Data/2 is more recent)
./zamora/simple/profile different (/Data/2 is more recent)
./ziburys/simple/profile        identical
./zion/simple/profile   identical
./zuse/simple/profile   different (/Data/2 is more recent)
-- 100 files listed
```

B.2 Profiles comparison report (detailed)

Host: Name of the source file

Headers: Number of headers number included by the source file

Components: Number of components included in the resulting profile

Resources: Number of resources included in the resulting profile

Missing resources: Number of resources present in `mkxprof`'s output, but missing in LCFG application output

Extra resources: Number of resources present in LCFG application output, but missing in `mkxprof`'s output

Differences: Differences in resource values for resources present in both outputs

Table B.1: Evaluation results (Scientific Linux 7 based hosts)

Host	Headers	Components	Resources	Missing /extra resources	Differences
altino	479	58	3083	0/1	3
arta	439	56	2968	0/1	3
babel	507	62	3268	3/1	4
bolt	469	59	3094	0/1	3
carnero	440	56	2968	0/1	5
cranston	389	49	2483	0/1	4
djokovic	479	58	3083	0/1	3
gala	477	61	3212	0/1	3
holywood	478	58	3083	0/1	3
honda	480	58	3083	0/1	5
maguire	479	58	3083	0/1	3
nora	440	56	2968	0/1	3
paeroa	480	58	3083	0/1	3
raven	479	58	3083	2/1	2
sashko	439	56	2968	0/1	3
sofia	477	58	3083	0/1	5
turing	440	56	2968	0/1	3
velma	479	58	3083	0/1	5
white	479	58	3083	0/1	3

Table B.2: Evaluation results (Scientific Linux 6 based hosts)

Host	Headers	Components	Resources	Missing /extra resources	Differences
aciu	416	50	2566	0/1	3
blackwell	330	46	2380	0/1	4
blogvm	373	51	2698	2/1	9
bocian	466	57	2955	6/1	3
chatty	365	50	2687	2/1	3
collins	329	48	2621	37/1	6
ert	592	63	3271	22/1	17
farg	490	58	2986	0/1	3
fimo	425	51	2630	0/1	3
hobgoblin	392	53	2960	56/1	2
horton	414	50	2566	0/1	3
idoru	328	46	2471	0/1	4
innerwick	486	54	2752	0/1	3
ivon	269	42	2208	0/1	2
jerry	436	49	2506	0/1	3
kegan	361	50	2662	2/1	3
kelvin	271	43	2158	0/1	2
lazar	435	47	2508	0/1	3
lindor	381	45	2293	0/1	2
march	552	65	3453	352/1	3
mullo	396	54	3023	2/1	2
neeps	393	44	2296	0/1	2
nix	387	52	2927	56/1	2
oakwood	316	45	2485	0/1	3
pergamon	354	50	2789	0/1	2
reeves	335	47	2426	0/1	3
selma	460	55	2844	8/1	1
skelp	530	57	3006	3/1	5
slatkin	442	57	3090	1059/1	0
stoner	315	47	2557	3/1	2
titan	407	47	2369	0/1	2
tycho	416	55	2940	5/1	2
yak	407	53	2785	20/1	5
yukon	416	50	2566	0/1	3
zamora	447	50	2593	0/1	3
zuse	452	54	2728	0/1	3

Table B.3: Evaluation results (Other hosts)

Host	Headers	Components	Resources	Missing /extra resources	Differences
4man	33	5	144	0/0	0
ard	11	3	76	0/0	0
ardeola	13	4	107	0/0	0
arezzo	11	3	76	0/0	0
artist	15	4	107	0/0	0
aruba	15	4	107	0/0	0
astonmartin	11	3	76	0/0	0
baked	9	3	76	0/0	0
bart	11	3	76	0/0	0
bombay	11	3	76	0/0	0
caley	11	3	76	0/0	0
caxton	13	4	113	0/0	0
charon	22	5	144	0/0	0
claise	11	3	76	0/0	0
cowan	11	3	76	0/0	0
eden	10	3	76	0/0	0
empoli	11	3	76	0/0	0
fox	11	3	76	0/0	0
garbo	9	3	76	0/0	0
hakone	20	5	144	0/0	0
helotrix	20	5	144	0/0	0
hopper	22	5	144	0/0	0
hornet	11	3	76	0/0	0
icarus	15	4	107	0/0	0
lion	9	3	76	0/0	0
luse	17	5	186	0/0	0
lychee	19	5	144	0/0	0
lyng	15	4	107	0/0	0
modin	22	4	130	0/0	1
napa	17	5	186	0/0	0
neave	11	3	76	0/0	0
otaka	18	3	76	0/0	0
pavarotti	11	3	76	0/0	0
pike	11	3	76	0/0	0
pollock	15	4	107	0/0	0
rocklin	11	3	76	0/0	0
salinas	17	5	186	0/0	0
scapa	11	4	149	0/0	0
shiel	11	3	76	0/0	0
sobotka	11	3	76	0/0	0
tarski	15	4	107	0/0	0
vili	20	5	144	0/0	0
yashin	22	5	144	0/0	0
ziburys	15	4	107	0/0	0
zion	15	5	186	0/0	0