A Description Language and Analysis Tool for a Software Development Environment

Mats Anderberg

Kongens Lyngby 2005
IMM-THESIS-2005-08

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 0909-3192

Preface

This master thesis project has been carried out in cooperation with the department of Informatics and Mathematical Modeling (IMM) at the Technical University of Denmark (DTU) in Copenhagen and Ericsson Mobile Platforms (EMP) in Lund. The project constitutes the final part of the requirements for obtaining a degree in Master of Science in Computer Systems Engineering at DTU. The work has been carried out in the period between January 11th 2005 and August 19th 2005. The project was supervised by Le Chi Thu at EMP, and Associate Professor Michael R. Hansen at the department of IMM at DTU.

Mats Anderberg, s031392
Lund, August 2005

Acknowledgments

First of all I would like to thank my two supervisors, Michael R. Hansen at DTU and Le Chi Thu at EMP for their guidance, support and contributions during the progress of this master thesis.

I would also like to thank Bo Johansson for his invaluable thoughts and endless patience when discussing the project with me.

The guys at the tools department at EMP for their support and comments on my work and all other employees at EMP for always having a moment to spare.

My family for always encouraging and supporting my studies and finally my girlfriend, Lina, for your love and for always being so considerate.

Abstract

Ericsson Mobile Platforms in Lund have for configuration purposes of the their software development environment developed a small description language. The purpose of the language is to configure a number of tools present in the environment and provide them with information making them capable of fulfilling their tasks, which involves building software executables, configuration management and delivery packing. The language provides the developer with a higher level of abstraction when configuring these tools.

The language however is not supported by a formal grammar nor any capabilities of checking the language for errors. It is this thesis task to formally define the syntax of the language, as much as possible compatible with the existing language, and to develop a tool capable of performing error checking of the language.

The outcome of this thesis is a formal grammar and semantic definition for the language and a analysis tool providing syntactical and semantical checks as well as analysis possibilities.

Keywords: BNF, EBNF, CFG, DSL, syntax tree, compiler compiler, lexical analysis, syntactical analysis, semantic analysis, formal grammar, visitor design pattern, software development.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Ericsson Mobile Platforms (EMP) develops platforms for mobile phones. The mobile platform is a complex architecture that involves several years of research and development.

Building software for large scale software systems, such as the mobile platforms developed at EMP, is a challenging task and when the number of files the software implementations reside in goes beyond several thousands it is important that the environment responsible for typical activities, such as building the software, configuration management and distributing the software to customers, is able to handle those in an efficient and time-saving manner.

To meet these requirements, at least to some extent, imposed on the software system, EMP has developed a small description language with the purpose of supporting the activities mentioned above. The language is used when setting up the build environment, meaning the software environment supporting the development of executable units, of the mobile platform software. The language is also used by the configuration management and when delivery packing the platform for different releases to customers. Easily put, the language describes for different tools present in the software environment what to build, what to version control and what to delivery pack. The purpose is to release the developer from the burden of writing for example make programs, in the case building the software executable, by hand. Instead the description language can be used to simplify the software development.

In the sense of lifting the abstraction of software development for a particular domain the description language can be considered a Domain Specific Language (DSL). A DSL is language with a small vocabulary dedicated to particular domain or problem, which in the case of the description language is the EMP software development environment. Hence the description language is not a General Purpose Language (GPL) like Java or C but a highly specialized language with small array of possible applications. A small introduction of DSLs are given in appendix B.

1.2 Problem Description

The language was originally designed to be used in somewhat simpler applications but as time has progressed it has evolved to be more complex and hard to maintain. Functionality has been added but not documentation or any error checking capabilities. The language of today is powerful (partly because it provides an interface to the scripting language Perl), but unfortunately there is no formally defined syntax for the language. This means that it is hard to find e.g. errors and difficult for the developer to correctly implement descriptions in the language.

Not having a strictly defined grammar nor a proper tool to syntactically and semantically check the language for errornous constructions causes possible faults in the files to be detected at a very late stage in the chain of actions that lead to the building of an executable or delivery packing of a product. This is not feasible and the purpose of this master thesis is to provide the language with a formal definition and a tool capable of performing checks to pinpoint errornous constructions.

1.3 Objectives

The goal of this master thesis project aims at achieving three major objectives:

A Learn and briefly document the approximate syntax of the current description language (CDL) language in a precise (informal) way. Understand the language and its applications.

B Define a new description language (NDL) as much as possible compatible with the present language and describe it formally.

C Design and implement an prototype analysis tool capable of performing syntactical and semantic checks and analysises based on the NDL.

The objectives of this thesis aims of supplying EMP with a grammar and a tool that supports a more strict language with defined rules and constrained behaviour. Hence the objective is not to design a whole new concept for the description language and its applications but to define the syntax more strictly and to develop a tool adhering to that syntax. The major objective of the thesis could alternatively be expressed as providing EMP with a basic foundation and a new paradigm for checking and evaluating the description files.

1.4 A description language application example

To illustrate how the description language is used, this section gives a concrete example of an application of the language, namely the building of an executable. An example description file is depicted in figure 1.1.

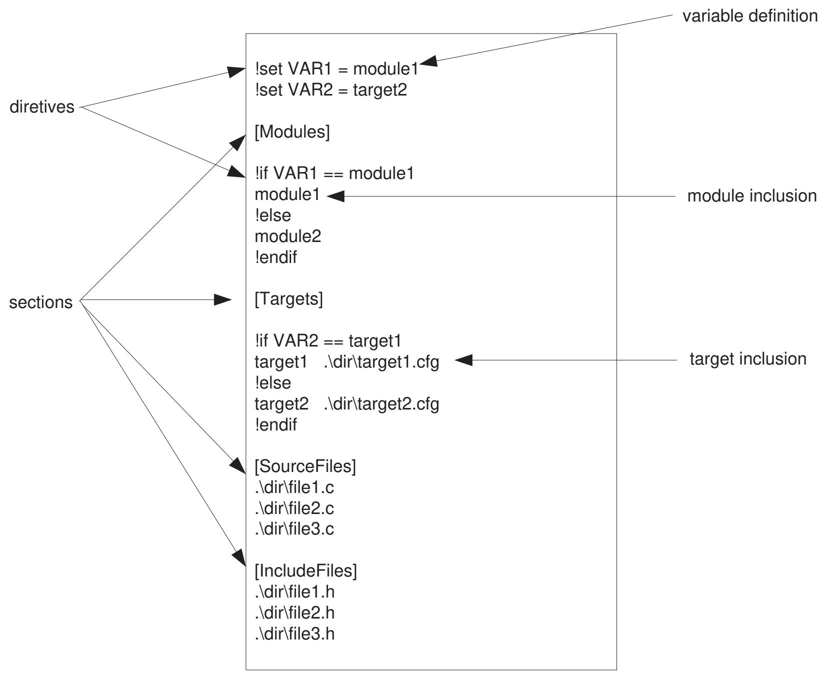

Figure 1.1: An example description file

The description language is made up of *directives*, like variable definition and conditional statements, and *sections*. Each section, which starts with a unique identifier enclosed in square brackets (e.g. "[Modules]" provides a particular functionality.

In the case of the example in figure 1.1 the file defines two variables (VAR1 and VAR2) that is later used in the conditional statements. The "[Modules]" section identifies the inclusion of other description files. The module concept will be further explained in section 2.2. The "[Targets]" section identifies the inclusion of a description file that specifies the configuration of the target, meaning the special compilation settings etc. for a given target (see section 3.2 for more information). A target can for example be Windows or the real processor used in the mobile platform. The two sections in the end of the file, "[SourceFiles]" and "[IncludeFiles]", lists the names of files that is a part of for example the building of an executable unit.

The module and target inclusion identifies a file hierarchy of description files that describes for the build tool what to build and how to build the executable. The file hierarchy for the example file in figure 1.1 is given in figure 1.2.

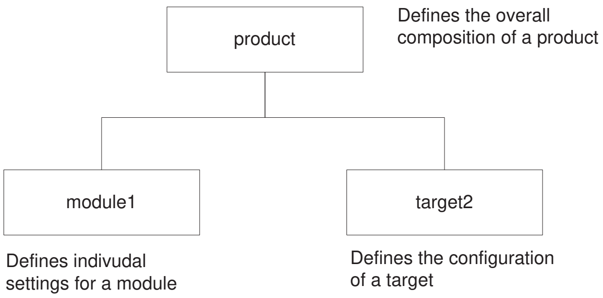

Figure 1.2: The description hierarchy pinpointed by the example

The root description in the file hierarchy is called a product description and typically defines the overall composition of the product (or the executable). When processing these files the build tool is able to extract the information needed to build the executable, which includes the files to be build and their individual settings as well as the configuration of the target.

This example only shows one application of the language and only gives a brief overview of it. The purpose is the give a short introduction to the language without going into to much detail that will hopefully provide a basic understanding that could be useful when reading the rest of the report.

1.5 Report Structure

The report is separated into chapters that map to the objectives described above:

- Chapter 2 describes the domain of the description language, namely the EMP software development environment.
- Chapter 3 covers objective A, including an identification of possible weaknesses of the current language.

- Chapter 4 covers objective B, proposing a new language based on the current one.
- Chapter 5 covers objective C, invoking the new language into an analysis tool.
- Chapter 6 summarizes the thesis and pinpoints further work.

At the end of each section there is a summary section intended to briefly inform the reader of the key information described in the chapter.

The reader is recommended to read the report in a top down fashion since the sections are dependent on one another in the order they appear. Also fundamental knowledge of language theory and compiler techniques are assumed although appendix C provides an brief overview on the fundamentals of compiler and language construction.

Chapter 2

The EMP Software Development Environment

2.1 Introduction

This section aims at briefly presenting the EMP software development environment as seen from the description language point of view. Only the parts of the environment where the description language plays a vital role will be introduced, hence this section will not give a comprehensive and detailed description of the complex structure that forms the EMP software development environment. It will merely present the tools using the description files and how these files interact with the environment.

The description files are in this section regarded as black boxes with information somehow needed by the software development environment, although a brief example is given in the introducton. The detailed description of what the description files actually contain and implement are left to chapter 3 which will thoroughly describe the actual language. For now it is enough to know that the description files holds information that the tools incorporated in the environment needs.

2.2 Environment architecture

The EMP software platform is structured as set of platform modules, each defining its own well-defined functionality of the mobile platform. Examples of different functionality incorporated into these modules are SMS, MMS, data communication etc.

All modules present in the software development environment contain a so called description file as depicted in figure 2.1. Figure 2.1 is oversimplified in the sence that it only illustrates that each module has a description file. It doesn't show the internal structure of the modules nor the different software layers that exist in the platform architecture. A module is basically a number of files (source files, document files, etc.) that together defines the functionality of the module and the description file for the module describes for the environment which files the module contains and how it should be configured.

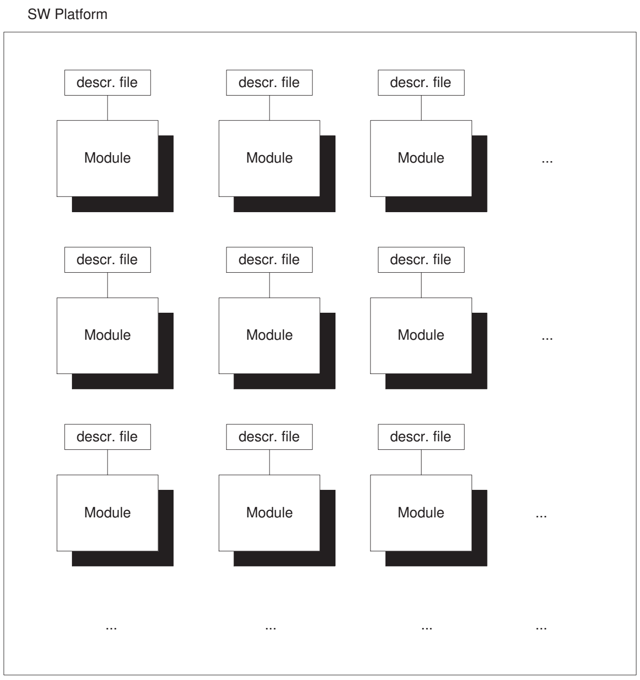

Figure 2.1: Modules and description files

The modules together form something that can be viewed as a file hierarchy where each module have the capability of including other modules and it is the description files responsibility to pinpoint how the modules are hierarchically chained together. The root description of the file hierarchy defines which subsequent description files (and their modules) that forms the specific product. Figure 2.2 gives an example of how the modules and their respective description file forms a file tree.

Depending on how the different description files are included and what kind of functionlity they implement they can be divided into different classes to distinguish between different types of descriptions. Figure 2.2 gives an overall picture

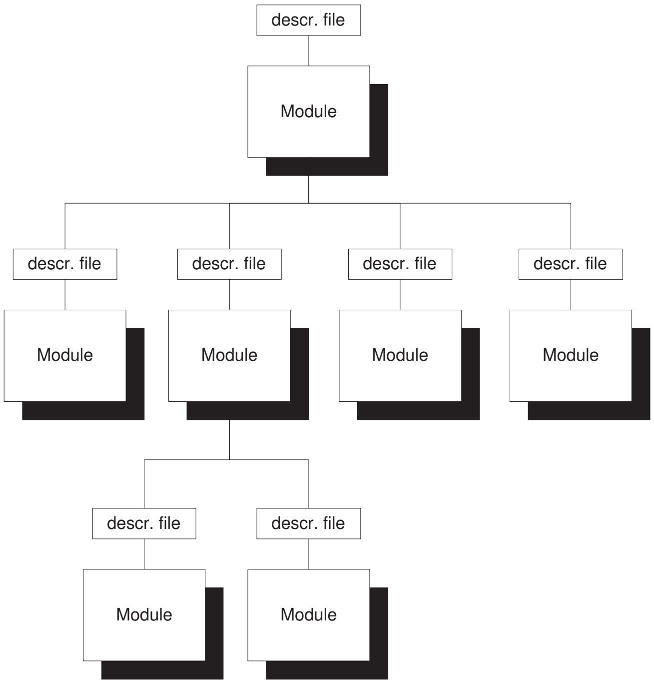

Figure 2.2: Example of description file hierarchy

of the module hierarchy. The notion of the different description file types is introduced in chapter 3.2.

2.3 Tools

The description files mainly participates in three activities of the software development environment, namely:

1 **Software building**, meaning the chain of events leading to generating a software executable.

2 **Software delivery packing**, meaning when the software is packed together and shipped to EMP customers.

3 **Configuration management**, for example version control and configuration of the software.

The tools responsible for performing the activities mentioned above are:

1 **SDE**, Software building

2 **PPZ**, Software delivery packing

3 **CME**, Configuration management

All of these tools use the description files in one way or the other to extract information needed for their respective functionality. The main purpose of the description language is to ”configure” SDE and CME so that these can be used in a wide array of applications without having to modify the tools. Figure 2.3 helps in resolving how the various tools interact with the description files.

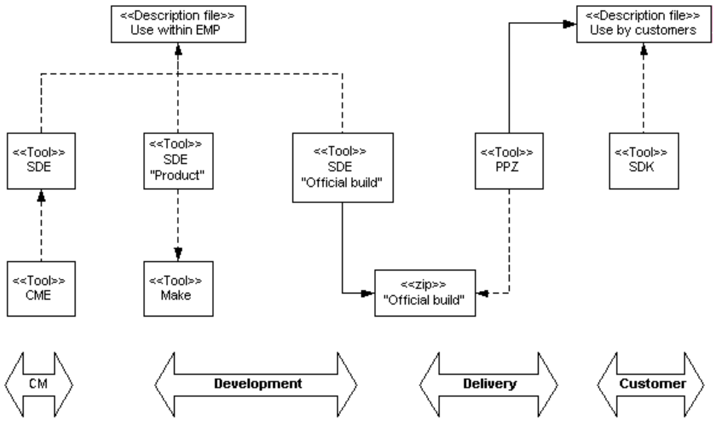

Figure 2.3: Description file interaction with tools.

In figure 2.3 the role of the description files in the software environment is illustrated. As can be seen, almost all interaction with the description files are done via the intermediate usage of SDE. Therefore SDE is not only present in the process of building the software executable but also provides an interface to the configuration management (CME) and the delivery packing (PPZ). The doted lines in figure 2.3 represents usage and the filled lines represent that some kind of output is produced. The subsequent sections will elaborate further on the specific purpose of each tool and and how they interact with the description files.

2.3.1 SDE

SDE handles the vital role of building executable units for the mobile platform products (see also section 1.4). It interfaces compilers, make programs, configuration management systems and debuggers. The main objective for SDE is to provide a tool for software development that can generate executable units both for the different mobile platform CPUs and for simulation and debugging the software on a PC. This main objective and how SDE uses the description files for this purpose is illustrated in Figure 2.4.

Figure 2.4: Generating the make file

The input to SDE are description files hierarchy identifying the entire software platform to SDE. SDE parses these files to extract the information needed to produce the makefile. This basically includes extracting information regarding which files in the platform that should be apart of the build and detailed make rules needed by the make tool. The platform can be built for different software targets and different *variants* of the platform. A variant is basically a subset of the software platform that defines individual settings. Different variants typically only differ in a few source files or settings.

2.3.2 CME

The major CM system for software development within EMP is ClearCase™and CME is a software plugin to that tool. CME uses SDE to get product information from the description files necessary when performing version handling of the platform software.

CME uses the description files to extract file information needed when performing a "freeze" on a software product. A freeze is when files are versioned together to form a well-defined portion of software. CME can extract information from the description files on three different levels of abstraction; 1) the modules for a given product, 2) the files for a given product or, 3) the files in a given module. Hence the CME usage of the description files are not necessarily applied to the entire file hierarchy but can also be invoked on a single module and thereby a single description file. CME usage of the description files are always done through SDE, which provides an interface for the functionality desired by CME.

2.3.3 PPZ

The tool responsible for delivery packing the platform software into a zip-file is PPZ. The zip-file , that is based on the content of the description files, is then released to the EMP customers. PPZ is a perl script that uses SDE to extract the useful information from the description files.

PPZ takes the informations extracted from the description files, via SDE, and transforms the original structure and contents of the description files to fit the purposes of the customer, which is illustrated as the delivery/customer portion in figure 2.3. SDK is a tool used by the customer to extract information from the description files used by the customers. Therefore the SDK includes a version of the SDE tool in addition to the transformed description files.

2.4 Summary

The description language dictates some important aspects of the software development environment for a couple of tools present in the very same. It helps these tools in resolving which files are a part of the software platform and how the software should be built, delivery packed and version controlled.

The platform is composed of a number of modules that each implement a portion of the mobile platform functionality. Each module contain a description file, implemented in the descripition language, responsible for defining the contents and specific configurations for the module. Together the modules form a hierarchy where the description files act as "glue", linking the platform product together.

Chapter 3

The Current Description Language

3.1 Introduction

This chapter describes the current description language (CDL), both in terms of the functionality provided by it and in terms of the detailed syntax and how the files are processed. The disadvantages and possible pitfalls of the language are also investigated and highlighted.

The chapter is organized as follows; Section 3.2 describes the different description file types of the language and the detailed syntax of the language is described in section 3.3. Section 3.4 describes how the files are parsed and processed as of today and section 3.5 highlights identified weaknesses of CDL.

3.2 Description file types

As mentioned in section 2.2 the platform software is made up of modules where each module contains a description file. Each description file adheres to one of five different types of descriptions, namely:

- Product description file, the root file of the description file hierarchy.
- Module description file, the file identifying the configuration and content of a module.
- Target description file, the file specifying the configuration of a target.

- Private description file, developer unique description file capable of overriding settings made in the modules.

- Include description file, description file that gets included via a include directive.

The type a file is given is not a constant property of the file, but depends on the way the file is referenced in the description file tree. This is typically only applied for the product and module descriptions. A file that is a product description in one run could very well be a module description in the next. To simplify the distinction between different types, naming conventions exists. But hypothetically a arbitrary description file can be any of the above listed description types, which is solely dependent on the the way it is referenced. The notion of description file types is depicted in figure 3.1 using a UML notation.

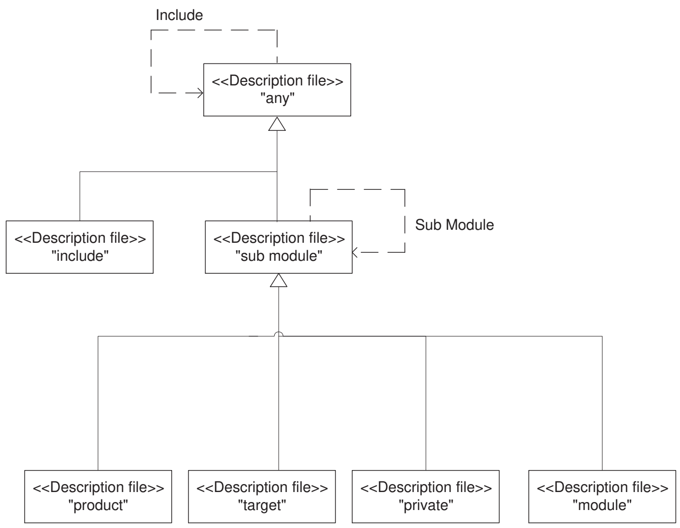

Figure 3.1: Description file types

As illustrated in figure 3.1 a description file can include other description files, either via the Include concept or the sub-module concept. It is important to distinguish between these two notions of inclusion. The Include concept is a logical include and the included file is a part of the file it is being included from. Typically the include file is of the type include description file but could theoretically be any description file type. The sub-module concept is a logical part of the module which is independently version controlled. The sub-module are CM-modules of their own. The inclusion and sub-modules create a description file hierarchy which for a platform contains several hundred description files.

3.2.1 Product description

This is normally the top level file for the product. The product description can either list all the files to be used for the build by itself or it can reference other, via the Sub-module concept software modules descriptions that in turn list the files to be used, see for example the example in section 1.4. You could say that the product description defines the composition of the product.

3.2.2 Module description

This is the top level file in a CM-module. It defines the contents of the module and how the module participates in the building of system. A software module description file typically contains the name of all the files needed to build the module and the individual compilation settings for the files. The fact that the product only references the module description means that it has no knowledge of the modules individual files. This also means that the module has full control of which files to use and the individual settings.

3.2.3 Private description

This type of file is normally unique for each individual developer. They are used to add new files or settings to the main configuration or to override specifications in the main configuration. This facilitates testing without the need to check out or modify the official description files. More than one private description file can be used for the same product.

3.2.4 Target description

This type of description file describe the specifics of a software target (compiler etc.). SDE has a standard set of supported software targets and compilers. A description file for another new target can easily be created. A software module can for instance define a specific software target of its own. Within this module the actual executables for compilers and alike can be stored and CM controlled. Other description files in the hierarchy chain can also replace the definitions in the target description files, completely or in part.

3.2.5 Include description

An include description is the type of description file that gets included via the include concept. This file typically defines settings needed by the module from which it is included. The inclusion is made through the include directive which is described in section 3.3.3.

3.2.6 Priority between file types

SDE defines a specific order in which the file types are prioritized. This means that the information in one file type can override the information stated in another file type. The files types have the following priority:

1. Private description file
2. Product description file
3. Module description file
4. Target description file

Hence the private description has highest priority and the module description lowest. A certain priority also exist within a file. The part of the file that is parsed last is the part that has highest priority. Because the files are parsed from top to bottom a statement in the end of a file can override a statement in the beginning. Because the include description file is a logical part of the file it is being included from it obeys the priority rules that exist within a file.

3.2.7 Description file tree

By referencing other description files using The Sub-module inclusion concept the description files together form a file tree. If it exists the private descriptions is the root of this tree and its sole leaf file is the product description. The product description then references its modules and target descriptions. Target descriptions are typically only invoked from the product descriptions and not from module descriptions. The module descriptions can then invoke other module descriptions and so on. Figure 3.2 depicts and example of how a hierarchy of descriptions files can look like.

Each description file in figure 3.2 have the possibility if including additional include descriptions files via the Include concept which is not shown in the figure but should be kept in mind.

3.3 The language

The current description language does not just contain one uniform context free syntax, but several context dependable syntaxes. CDL can be described as a language with three different levels or steps:

1. Directives

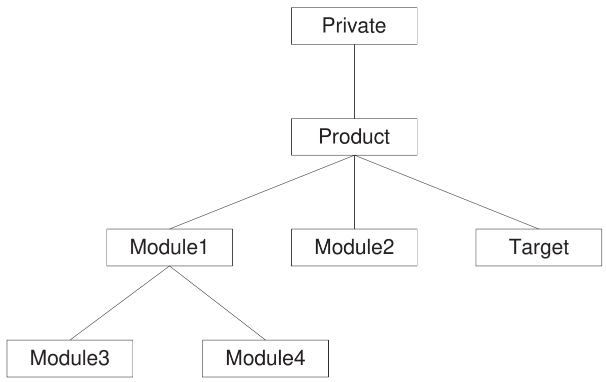

Figure 3.2: Description file tree

2. Sections
3. Section languages

Step 1, Directives, can be considered as a preprocessing step of the language. This step defines a number of directives which main purpose is to define variables and conditional statements that control the flow when the files are parsed. The directives can be considered context free and are applicable in any context. A directive is always proceeded by the "!" character. The directives currently defined by the language are:

- Variable definition
- Variable deletion
- Conditional statements
- Include directive
- Options directive

Step 2, Sections, is concerned with dividing the syntax into several so called sections. Each section defines its own syntax or language with its own specific functionality; step 3. Hence the languages residing in the sections are context dependable.

The syntax of today is dictated by SDE which is responsible for parsing the description files. Step 1 and 2 are parsed line by line and not file by file. Hence the language can be considered as a row-oriented language and thus each statement has to be completed in one line.

It's important to distinguish between the directive part of the language on one hand and the section part on the other hand. CDL can be seen as a language with two phases; 1) the directives are evaluated in a preprocessing step where the variables are given values and the conditional statements defines which rows that are "active", 2) The "active" rows in the sections are extracted and processed. Sections 3.4 describes in more detail how the language is parsed and processed. The following subsections will elaborate further on the syntax of the directives and sections.

3.3.1 Variables

Defining a variable in the description language is accomplished with the use of the *set* directive. The syntax is as follows:

```
!set [Option]* Identifier [= Value]
```

Hence the "!set" string identifying the kind of directive is followed by zero or many options then a variable unique identifier to identify the variable, followed by a possible value. The value assigned to the variable could theoretically be any sequence of characters. The language of today does not support any kind of typed variables and SDE will handle all variables as strings. SDE will match all characters, beginning with the first character after the equality sign (excluding white spaces) and ending with the character proceeding the new line (also excluding white spaces).

Variable states

Basically a variable in the description language can be in one of four different states, depicted as a state chart in figure 3.3.

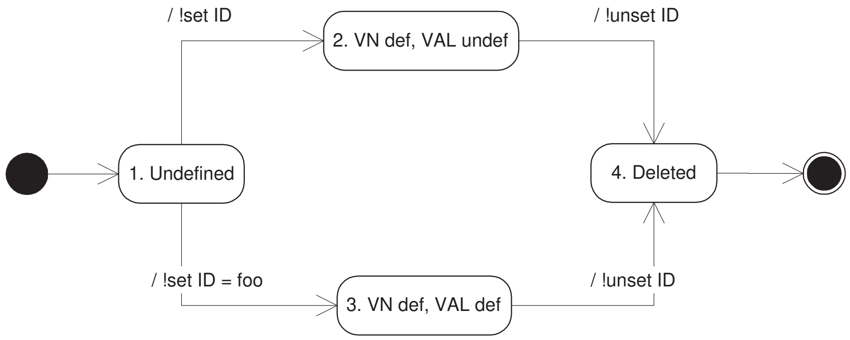

Figure 3.3: Variables states

In figure 3.3 *VN* represents a arbitrary variable identifier and *VAL* represents a possible value assigned to the variable. Figure 3.3 depicts that a variable is in one of the following states; 1) the variable is undefined, 2) the variable name is defined but not the value, 3) the variable name is defined and so is the value and 4) the variable is deleted. The figure implies that if a variable is in the state adhering to number 2, it is not possible with a subsequent use of the set directive also to assign a value to the variable. This is however not the whole truth. SDE defines a number of options to the set directive that can be used to override the logic defined in figure 3.3. These options are:

- -i, irreplaceable set, i.e. from the point the variable is defined and onwards no other definition can override this definition. However a variable that is defined irreplaceable can be overridden if the -i option is used again.
- -f, forced set, i.e. the variable is defined and/or assigned regardless of a possible previous flag, unless it has been set as irreplaceable before.
- -l, the variable is defined locally. The variable scope is the remaining part of the file it has been defined in and in subsequently included files via the include directive (see further section 3.3.3). At the end of the file a possible previous value for the variable is restored. If no previous value has been defined the variable is deleted. The definition is ignored if a variable with the same name previously have been defined with the -i option.
- -a, appends text to an existing variable. This option is necessary due to the fact that a file can be parsed multiple times, and prevents the variable to contain arbitrary duplicates of the appended text.
- -e, the value is first evaluated as perl expression before assigned to the variable.

To further illustrate how these options can be used an example is given:

```
!set VAR = A

!set -f VAR = B

!set -i VAR = C

!set -i VAR = D
```

The final value of the the variable `VAR` is D in the example above. The only way to override the initial definition (A) is to use -f or -i and the only way to override a variable defined with -f (B) is to use -i and finally, the only way to override a a variable defined with -i (C) is to use -i again (D).

Hence it is possible by using the right options to override the behaviour described in figure 3.3. Figure 3.4 gives a more correct view of the states a variables can reside in. In this figure only the options -f and -i are considered. Should all the options be added, the state diagram would be immense and not very intuitive.

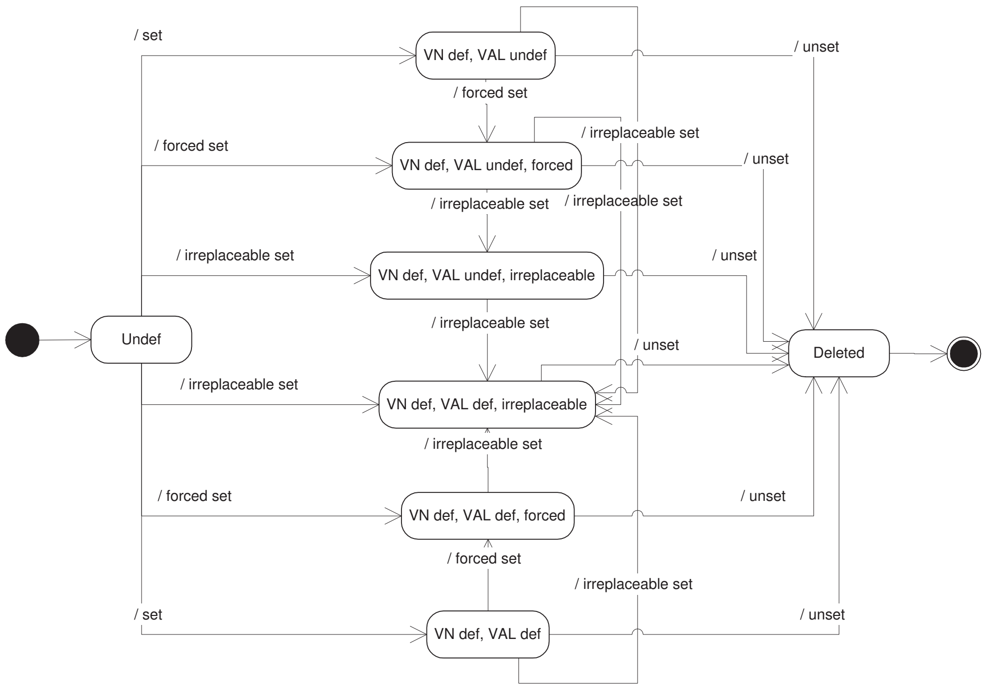

Figure 3.4: Variables states with some override aspects added

However this figure suffices to show the complex logic in defining a variable's scope and controlling the way it can be overridden. In addition with the fact that SDE parses the description files in multiple passes (explained in sections 3.4 cause implications that are obvious. These implications will not be investigated here, but in section 3.5, describing the fallacies and pitfalls the current language produce.

Variable reference

Once a variable has been defined it can be referred to in any context of the description file using the following construct:

```
!set ID = foo
%ID%
```

As seen from the example above a variable reference is a variable identifier enclosed by "%" characters (%ID%). Here %ID% will get the value "foo".

Variable expansion
The variable reference is expanded by SDE in the preprocessing step. SDE allows variables to be referred to anywhere within a description file. Theoretically constructions like this are legitimate:

```
!set IF = if A == 1
!set SET = !set A = 2
!set ENDIF = !endif

%IF%
%SET%
%ENDIF%
```

Here an if statement (conditional statements will be explained in a later section) is declared as variables and then expanded to form the actual directives. The preprocessed file will have the following appearance:

```
!if A == 1
!set A = 2
!endif
```

This feature works fine with the way SDE is currently implemented but may form an intolerable ambiguity when building a syntax tree, which will be investigated further in chapter 4.

Multi line values
A variable defined with a value can consist of several lines (including line breaks) and can be defined using a line-oriented quoting of the value:

```
!set SOME_VARIABLE = << END_OF_DEF
This is one line
This is another line
END_OF_DEF
```

The "END_OF_DEF" identifier can be an arbitrary string that is not part of the definition. When using the variable the full value including the line breaks will be expanded.

Variable deletion
The language also the defines the possibility to delete a previously defined variable. This is accomplished with the use of the *unset* directive which corresponds to the actions leading to the *Delete* state in figure 3.3 and 3.4:

```
!unset ID
```

This construct will cause the variable to be deleted, but also to be blocked for further use. Thus, a variable that has once been deleted can never be defined again with the same variable identifier nor can it be referenced again.

Variable scope
The variable scope is per default defined to be global if the variable has not explicitly been declared as local. A variable having been defined is applicable everywhere in the file hierarchy. As explained earlier the language is a two phase language and it is therefore legitimate to refer to a variable before it has been defined. The preprocessing step in SDE takes care of evaluating all variables regardless of their position in the file tree.

3.3.2 Conditional statements

The language defines the possibility to define conditional statements which makes it possible to control the flow of the file being "executed". The semantics and the syntax resembles the one used in C source files. The following keywords are recognized:

- **!if** logical-statement
- **!elseif** logical-statement
- **!else**
- **!endif**
- **!ifdef** variable identifier
- **!ifndef** variable identifier

The logical statements support comparison between variables and string literals using the operators "==" for equality and "!=" for not equal. The comparisons are textual and not case sensitive. Several comparisons can be grouped together with the logical operators conjunction ("&&") and disjunction ("||"). Consider the example:

```
!if A == 1 || B == 2 && C == 3
...
!elseif D != 4
...
!else
...
!endif
```

Here the string preceding the equality operator is implicitly interpreted by SDE as a variable reference and the one following the operator is a value to SDE. Hence it's legal to refer to a variable on the right side of the comparison but not the left. Using a variable reference on the left side would make SDE first expand the variable and then try to look up the expanded value in the variable database. The variable would not be found because it is a value and has no entry in the table and the comparison would evaluate to false which may not have been the intension in the first place. Hence, the language contain some inconsistency in the use of variables. A string proceeding a comparison operator in an if statement (ifdef and ifndef also for that matter) is implicitly meant as a reference but anywhere else the string has to be encapsulated between "%" characters to take the desired effect.

As seen from the sample code above it is also possible to use the conditional statements elseif and else following and if statement, that resembles the most frequently used programming languages in their semantic and syntax to great extent. The conditional statements are also allowed to be nested.

In conditional statements conjunction has higher precedence than disjunction. The precedence can however be overridden by grouping the logical expressions inside parenthesis.

ifdef and ifndef

The language also defines the conditional statements *ifdef* and *ifndef*, identical to the two directives #ifdef and #ifndef used by the C preprocessor and their semantic and syntax are identical:

- **!ifdef variable name**, the branch is taken if the variable is defined.
- **!ifndef variable name**, the branch is taken if the variable is not defined.

All conditional statements have to be ended with an "!endif" to close the statement. This syntax is also inspired by the C preprocessor. Example:

```
!ifdef VAR
...
!endif

!ifndef VAR
...
!endif
```

3.3.3 Include directive

This directive is once again an inheritance from the C preprocessor and has the following syntax:

```
!include filepath
```

Where *filepath* is a string identifying the file to be included. Files referenced with the include statement are automatically added to the description file from which they are included. The file path can either be absolute or relative to the file containing the include statement. The file path may also contain variable references as parts of, or the whole file path.

3.3.4 Options directive

The *options* directive dictates the activation of specific flags that bear a special meaning to SDE when parsing a file. The syntax is as follows:

```
!options option
```

Where *option* is one of the following:

- **Silent:** Turns of various information messages that are triggered by the product description file.
- **MultipleSections:** Forces SDE to parse the entire file. The default behaviour of the SDE parser is to search a file for a given *section*. Once found the parse is stopped.
- **CheckDescrConditionals:** Makes SDE check that if statements and closing endif's match.

These options all triggers the activation of a behaviour that is per default unactivated.

3.3.5 Sections

Besides the *directives*, that are applicable everywhere, the language defines so-called sections. Each section starts with a tag consisting of a name enclosed in square brackets which is similar to the syntax used in Windows "ini" files. Example:

```
[Variants]
...
[SourceFiles]
...
```

Each section provides different functionality to the environment. The number of different sections defined by the language is approximately 30 but this number is not fixed and sections are constantly added and removed.

Additive or replacive sections
A section or more accurately the information it contains can either be replacive or additive. In a replacive section the information can be overridden or replaced by an identical section in another file provided that the file has a higher priority. Additive means that the information is simply concatenated with other section information residing in identical sections.

Section classes
The sections can be grouped into a number of classes, each defining similar or identical syntax and functionality. The identified sections classes are defined in table 3.1.

Besides the section classes defined in table 3.1 there are a number of miscellaneous sections which have not been able to put in specific class of sections.

File list sections
The sections adhering to this class are used to list file names in the build process for a software product. Examples of file list sections are; "[IncludeFiles]" and "[SourceFiles]" listing header files and compilable source files. Typically these sections merely list the names of the files, each separated by a new line and nothing more:

```
[IncludeFiles]
.\dir\file1.h
.\dir\file2.h
.\dir\file3.h
```

But some sections also exposes the possibility of defining a number of options for the specified file. An example of this is the "[SourceFiles]" section that allows a number of compilation options to be stated on the same line following the name of the file:

```
[SourceFiles]
.\dir\file.c  AVR(=-z9)
```

The compiler option is typically specified with an identifier unique for each target followed by a value enclosed in parenthesis.

The file list sections are additive meaning that the contents of each section are concatenated. However two identical file names defining different options will

- **File list sections:** These sections list files. In addition to the files listed a number of options can be stated (although only for a small number of the file list sections). The options might include different compiler options that is to be mapped to the specified file.
- **File list modification sections:** Each file list section has a corresponding modification section with the intended functionality to remove a file listed in the file list section.
- **Packaging section** These sections also lists files. The files stated in the packaging sections are intended to be exposed to the tool responsible for packing the platform software (PPZ) to customers.
- **Packaging modification sections:** Same functionality as file list modification sections but for the packaging sections.
- **Variant sections:** The product typically produces several different executables and the variants sections identify the names of these and how they can be grouped together etc.
- **Target specific sections:** These sections identify targets specific actions and contains all the information on how to create the makefiles for the build process.
- **Target section:** Identifies the Sub-module inclusion of target description files.
- **Module section:** Identifies the Sub-module inclusion of module description files.

Table 3.1: Section classes

result in the option to the file listed in the description file with the highest priority taking precedence over the other. The typical behaviour when extracting the files is that there are never duplicates of the same file (the key being the actual file name, excluding the path). Hence a file defined twice will result in only one entry in the final list of files, that being the file listed in the file with the highest priority ignoring possibly different paths.

File list modification sections
All file list sections have a corresponding modification section. These section are denoted with a name identical to one of the file list sections preceded with a minus:

```
[-SourceFiles]
```

```
.\dir\file1.c
```

Each of these section can be used to remove a file that has previously been defined in a file list section. The removal of a files listed in these class of sections is done after all file list sections have been examined. The removal is also done per section, meaning that a file listed in a "[SourceFiles]" section can only be removed using the "[-SourceFiles]" section.

Packaging sections
These sections are a complementary to the previously explained file list sections and contain information (files) about what parts of the platform that should be exposed to the EMP customers. What these sections do is to divide the platform logically in three partitions, namely exposed files, forbidden files and other files. The exposed files are such files that always should be available to EMPs customers. The sections exposing files are named "[Packaging_Exposed*] where * is the name of one of the file list sections. Example:

```
[Packaging_ExposedSourceFiles]
...
[Packaging_ExposedIncludeFiles]
...
```

Forbidden files are the opposite of exposed, files that never should be sent to customers. These files should be listed in a section named "[Packaging_ForbiddenFiles]". The third category, other files, consists of the files (the files listed in the file list sections subtracted by the ones listed in the packaging sections) that are neither exposed nor forbidden. When packing the platform the platform packer decides which files that are to be included in addition to the exposed files with respect to the dependencies that exist between the files.

Variants sections
The product typically produces several different executables each differing in a few source files or settings. Each executable is denoted with a unique name called a variant. SDE defines a couple of sections that define the names of the variants and how they can be grouped together when their individual settings are very similar. The variant name is used throughout the description file hierarchy to the control the "execution" of the files with respect to the chosen variant.

Target section
The "[Target]" section is responsible for identifying which target description file to include. The target description file then describes the specifics for how to build an executable unit for that target. The syntax is typically a target unique identifier followed by the path to the target description file:

```
[Targets]
IAR-ARM7   .\dir\iar-arm7.cfg
```

On the encounter of such a statement, provided that the target identifier matches the chosen target, the target description file will be included by SDE.

Target specific sections
The target specific sections are those residing in the target description file pinpointed from the "[Targets]" section and contains some more advanced instructions. The sections defined in this type of file control the actual generation of the target specific makefiles. These sections are an example of replacive sections meaning that can be overruled by another alike section residing in a file with higher precedence.

Module section
The "[Modules]" section identifies the inclusion of module description files. A module can be included from any type of file and the "[Modules]" sections together with the "[Targets]" sections identifies the platform file tree when SDE parses the files. The syntax is a path pointing to the module folder in the file tree followed by a label. SDE will search the module folder for its description file. A label (`CNH160676_R3E`) is string identifying what version of the module that should be included:

```
LD_FuncBlocks_011\cnh160676_level0            CNH160676_R3E
```

There also a "[-Modules]" section in case it is desired to remove a whole module. The syntax is identical to "[Modules]" except the label preceding the path.

Miscallenous sections
The sections and sections classes listed above are not a complete description of the various sections that are possible to use. They are however the most important ones and the reader will not be exhausted with extensive knowledge of every single existing section.

3.3.6 Perl

The description language is highly incorporated with the scripting language Perl. This due to the fact that the SDE parser is implemented in that very language. The description language itself also offers the writer to express arbitrary perl statements, although constrained within a certain context (like the -e option to the set directive). Whether or not this is a recommendable feature will subsequent sections elaborate further on. For now it is just explain what is possible and what is not. As mentioned before the use of perl is allowed in certain predefined contexts of the files,

- In the set directive
- In if statements
- In certain sections

The perl expressions can be either arbitrary or following a strict syntax dictated by a number of predefined functions that follow a strict syntax. Examples of such are functions that return whether or not a variables name is defined and if a given variable contains a another explicitly given substring. The arbitrary expressions, that can be invoked by using the ”-e” option to the set directive, will force SDE to interpret the assigned value as a perl expression and evaluate it as such. The arbitrary perl expressions can also be used in if statements. SDE automatically evaluate the expressions as perl and no options has to explicitly be stated.

The typical usage of perl is in the context of if or set directives. But SDE also defines sections that contain entire perl sub routines.

3.4 Processing of description files

SDE is the tool responsible for parsing and processing the description files in order to obtain information and to produce the output needed to perform the tasks imposed on SDE by the environment. This includes providing CME and PPZ with information enabling them to fulfill their task, namely configuration management and delivery packing, and building software executables.

The SDE processing of a file hierarchy of description files is not performed in one single parse. Instead the file system is parsed in multiple passes and eventually all information needed have been retrieved to form a complete information base of the system. Figure 3.5 gives an overview of how SDE processes the description files.

3.4.1 Multiple file processing

A ”start” description file is provided to SDE identifying the root of the file tree to be parsed (this is the typical behaviour, the objective could also be to only process a single file). This file is typically the product description. SDE performs an initial ”dummy parsing” of the entire file tree pinpointed by the root description file. The dummy parsing's main objective is to extract variable information and to build up a variable database for the file tree. The dummy

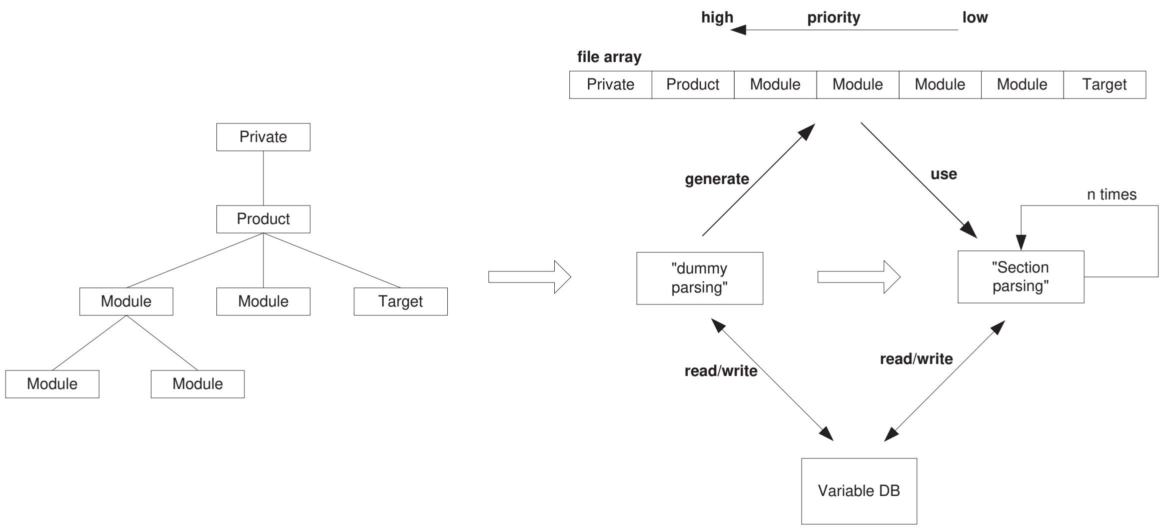

Figure 3.5: SDE description file processing steps

parsing also creates an array of all file paths to the included files from the root file. Hence the structure is no longer a tree but a flat structure organized in such a way that the priority order explained in section 3.2.6 is fulfilled. The array is organized in descending priority order, the file with the highest priority is located first in the array and the file with the lowest priority is located last. By retrieving the files from the array in the order they are arranged the priority order between the file types is ensured. This also implies that a priority order exists within a file type which will be based on their individual location in the file tree and in the order in which they are defined. As mentioned in section 3.3.5 sections can be either replacive or additive. By accessing the array either backwards or from the front the information in sections can be ensured to be replaced or added with respect to the file priorities.

The objective of the "dummy parsing" is to ensure that all variables are defined. This is because a parent file might have variable dependencies to variables defined in a leaf file. Once the "dummy parsing" is completed the following parses will use the created file array and variable database. These parses are illustrated in figure 3.5 as the *section parsing*. The section parsing will continuously update the variable database given the rules explained in section 3.3.1 adhering to the variable definition and deletion. The objective of the section parsing is to extract information from the sections needed by the activity accessing the description file, where activity is implied to be either, build, packaging or CM specific actions. Each scan of the files will search for different sections and retrieve section specific data. SDE has a number of data structures that saves the information extracted from the sections. Once the processing of the description file system is ended these data structures holds the information base of the product (which can also be for a single file).

Because the files are parsed first one initial time and then several subsequent times the variable scope will be global, unless the variable is explicitly defined to be local. A variable defined further down the file tree is accessible in parent files in the next scan of the file array. This feature is a bit unorthodox but probably not unintended. For example the product description may want to include a file depending on a specific kind of target, which is not known until later in the processing. The inclusion is dependent on a target specific variable defined after the product description have been parsed. But considering the complex way the variables can be overridden and the global variables scope in addition to the multiple parsing gives a very unintuitive variable handling and it's difficult to predict a variable's value at a given time. The question is whether the possibilities provided by the current implementation overshadow its disadvantages.

3.4.2 Single file processing

Each individual scan of a description file can be divided into two phases as mentioned in previous section:

- **Preprocessing**
- **Section retreiving**

As mentioned in the previous section some of the *directives* are very much influenced by the preprocessor used in the C programming language. The preprocessing done by SDE includes executing the directives in the language and provides the section retrieving step with a preprocessed description file for further compilation. This step then parses the files again to extract desired information from the sections. Figure 5.4 depicts this.

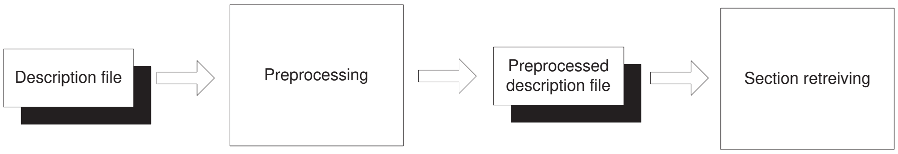

Figure 3.6: SDE description file processing steps

Preprocessing
What the preprocessor step effectively does is to parse the directive part evaluating every conditional statement in the file. Depending on whether or not the

conditionals evaluates to true or false, the branch will be included or excluded from the preprocessed file. The preprocessor also handles the variables defined and expands all references made to one of those. When the preprocessing is done parsing the file the information contained in the file will be reduced and all variable references will be expanded leaving only the "active" rows in the preprocessed file.

Section retrieving
This step retrieves and processes information contained in the *sections*. The section or sections that is to be found is explicitly requested and differs from scan to scan. In each scan of the file array, different sections are extracted to eventually form a complete information base of the system. The data is then either used to generate the make file, provide the platform packer with packaging specific information or CME with file information.

3.5 Fallacies and pitfalls

The description language of today gives great flexibility when implementing descriptions, partly because the powerful perl availability but also due to the fact that there is no strict grammar and very limited error checks. In addition to the complex and unintuitive way the files are processed and parsed the language of today highlights a number of fallacies that makes it feasible to elaborate on a refinement of the current language which regards the following matters:

- **Absence of formal grammar**
- **Handwritten parser**
- **File processing**
- **Variable handling**
- **The extensive Perl availability**

The following sections will elaborate further on each of these weaknesses and the implications they impose.

3.5.1 Absence of formal grammar

The CDL is not formally defined and it is therefore difficult to know how to correctly implement descriptions in the language. Defining a formal grammar would eliminate many of the ambiguities and risks in the current language and would simplify error checking and handling.

3.5.2 Handwritten parser

The SDE parser is handwritten implemented in the Perl scripting language. Having a ad-hoc developed parser makes maintenance of the language very difficult and also produces consistency problems in the sense that modifying the parser has to be done carefully to maintain a correct implementation.

It is therefore feasible to elaborate on another paradigm in how the language is processed and handled. Instead of implementing the parser by hand it would be sufficient if the parser is generated using one of the vast amount of compiler compiler tools available.

3.5.3 File processing

As previously mentioned SDE parses an individual description file several times. After each scan the evaluated file (the preprocessed file) is simply thrown away and no knowledge of the file is kept except the extracted information intended for the particular parse in addition with variable database that is continuously updated. Parsing the files several times is time consuming and it would be more feasible to parse a file one time extracting all information at the same time.

3.5.4 Variable handling

The multiple parsing of the description files and the fact the variables are kept from scan to scan implies that all variables are global if not explicitly stated otherwise. In addition to the overriding possibilities available basically means that every parse can tamper with all variables which in turn imposes a number of possible unwanted "features" in the variable handling. For example consider a variable being defined and assigned a value:

```
!set A = 1
```

Then in a subsequent file the variable is overridden:

```
!set -f A = 2
```

When the first set statement is parsed in a subsequent scan of the file, the assignment will be ignored by SDE because it has not been defined with the proper option and the variable will hold the value "2". This might not be the intention when defining the variable in the first place, at least if the intention is to be able to use the variable with the original value in the file it is defined in.

A similar behaviour can be reproduced by using the unset statement. A variable defined and then deleted will be deleted and unable for use in the next parse as well.

Given the current implementation the only way to make sure that the value you assign to a variable is actually the value used when referring (provided that you refer to it before it is overridden) to it is to, without exception, define the variable as irreplaceable (with -i).

The difficulties imposed by the variable handling is mainly the cause of multiple parsing and the dependencies between them with respect to the variables. As described above this might lead to undesired behaviour but might also be a feature that is needed given the current implementations of the description files. However having a more constrained handling of the variables that limit the risks the current implementation produce is definitely desirable.

3.5.5 The extensive Perl availability

The Perl availability the language offers gives a powerful, but difficult to control, functionality. It is very easy to make mistakes but also increase the capabilities and adds a dimension to the language. However no errors or warnings will be issued if the expressed perl statement contains any syntactical errors. The functionality typically expressed in perl can be replaced by a number of predefined functions implemented by the language. It is therefore recommended that the use of arbitrary perl statements is abolished and replaced by a library of available functions providing a constrained and well-defined functionality. This would provide a more language independent solution instead of the very much Perl dependent solution that exist today.

3.6 Summary

The EMP software development environment uses the CDL to extract information regarding the configuration of some of the tools present in the environment. The language can be said to be a two phase language; phase one being the directives and phase two being the sections. The directives define fundamental constructs such as conditional statements and variables. The sections contain information regarding for example the files a module contains and the individual settings of a target. The directives are evaluated in preprocessing step, leaving a preprocessed file with only the sections remaining which are then extracted to fulfill the needs of the environment.

The descriptions files where the language are implemented in forms a file tree that identifies the configuration of an entire product to the environment. The description files can be one of five different types; 1) private description, 2) product description, 3) target description, 4) module description, 5) include description. The type a file adheres is dependent on how it is referenced in the file tree.

A number of fallacies has been identified in CDL; 1) the file processing is not optimal, 2) the variable handling difficult to understand and may produce unwanted features, 3) the extensive perl availability is hazardous and 4) having no formal grammar and a handwritten parser makes language maintenance difficult and the language difficult to understand. Chapter 4 will take these highlighted weaknesses into consideration when defining the NDL.

Chapter 4

The New Description Language

4.1 Introduction

One of the objectives of this thesis project is to formally define a grammar and semantic for a partly new description language (NDL), which however should be to as much extent as possible compatible with the current one.

Considering the highlighted weaknesses of CDL this chapter defines a new language, in terms of a formal grammar and semantic. This chapter also introduces possible constraints and improvements to the given language. Although one of the requirements imposed on the thesis project is to as much as possible define the new language to be backwards compatible with the existing, some aspects of the current implementation is however considered hazardous and therefore feasible to alter and improve.

This chapter is structured as follows; section 4.2 introduces a number of design decisions based on the weaknesses identified in chapter 3, section 4.3 defines a formal grammar for NDL and section 4.4 the semantic. It is recommended that appendix C.3 is read before section 4.3 because it introduces the formalism and notation of the language definition.

4.2 Design decisions

Chapter 3 highlighted a number of fallacies in the CDL and the implications they impose to the environment:

- No formal grammar
- Handwritten parser
- File processing
- Variable handling
- Perl dependencies

In NDL these items are revoked and alternative solutions are used regarding how the files the language reside in are treated and how the language is defined and implemented. The following sections elaborate on these solutions.

4.2.1 Formal grammar

CDL have no formally defined grammar. NDL however, will have a formal grammar using the EBNF notation described in section C.3. This grammar is however not complete in the sense that a formal definition have been found for every aspect of the language. The grammar for NDL is described in section 4.3.

4.2.2 Handwritten parser

The lexer and parser for NDL is not handwritten as opposed to the parser for CDL. Instead a compiler compiler tool (JavaCC) is used to generate the lexer and parser as well as a tree representation (AST) of the file being parsed. Input to the parser is the formal grammar defined in sections 4.3. The detailed design of the parser is described in chapter 5.

4.2.3 File processing

The description files form a well defined file hierarchy of descriptions that identifies and defines the platform product, which is depicted in figure 4.1.

The product description typically includes module and target descriptions and each modules can invoke further modules. However this structure is not kept when SDE processes the file tree. Instead the tree is transformed into a flat structure, or more accurately an array of the description files present in the product. This is explained in more detail in section 3.4.

However, when defining the new language the file tree structure is assumed to be intact, meaning that the files are processed according to their location in the file tree. Also the feature of parsing the files multiples times, done by SDE, is no longer applicable. The new language is assumed to be parsed only one time, or more correctly, the tool developed as a part of this project performs no multiple

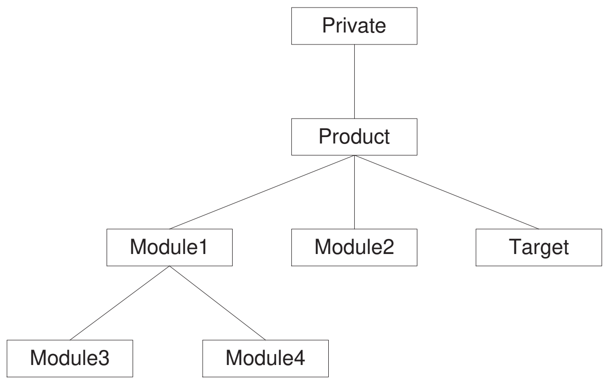

Figure 4.1: Example description file tree

parsing. The tool scans the files a single time and in contrary to SDE, builds an Abstract Syntax Tree (AST), representing the language grammar.

The file processing have no special meaning when defining the formal grammar, but will play a vital role when defining the language semantics. Given the new way of processing the files, the new language is handled in more orthodox manner that resembles the way a traditional compiler works.

4.2.4 Perl dependencies

The current language gives great flexibility that is the extension of being able to invoke Perl expressions, although not entirely context free, in the description files. The new language will forbid the use Perl expressions motivated by the possible dangers and unpredictable behaviour they might produce. Instead, inherited from the current language, a library of predefined functions are available. These functions are implemented by the language and should not have any dependencies to the Perl language.

The Perl abolishing is however only restricted to the if and set directives. It is still possible to use Perl in other contexts.

4.2.5 Variable handling

As an extension of the assumptions made in section 4.2.3, keeping the file tree intact and single parsing, but also the implications imposed by the current variable handling described in section 3.5, a new more intuitive way of handling the variables is feasible to introduce. Instead of all variables being global, the variables are given a more constrained scope adhering to where in the tree they are defined. A variable being defined in *file A* is applicable in that very file and in all leaf files to *file A*. A lot of variables defined in the product description should be regarded as constants by the sub-modules included from the product description. Also the private description typically defines variables that should override subsequently defined ones. Therefore it is feasible to regard the variables as constants when they are propagated down their scope. In other words, a variable is override able in the file it is being defined in but in leaf files the variable is regarded as constant and can not be overridden. This solution might however be to general and occasions might occur where a variable should be able to override by leaf nodes. However for now, this kind of constrained and general variable handling suffices. Figure 4.2 illustrates gives a general depiction on how the variables are handled.

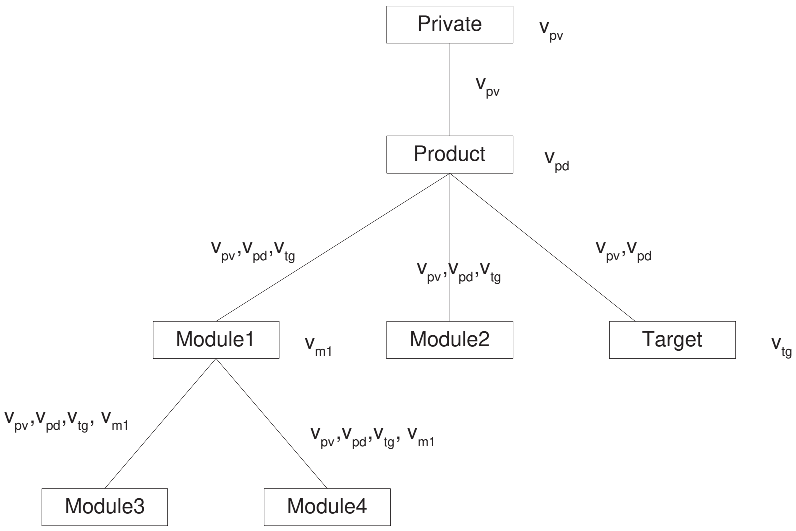

Figure 4.2: Variable handling

In figure 4.2 v_{pv} denotes a set of variables defined in the private description, v_{pd} variables defined in the product description, v_{tg} variables defined in a target description and finally v_m variables defined in a module description. The variables are propagated down to their leaf files. An exception is the target description, these variables are returned back to its parent, motivated by that the target description defines a number of variables that should also be visible in the modules.

To further illustrate the scope a variable is given an simple example is given i figure 4.3. Here, the variables C and D are inherited down to root file of the figure from a the parent file (which is not visible in the figure). In the root file two new variables are defined (A and B). The file invokes two leaf modules and

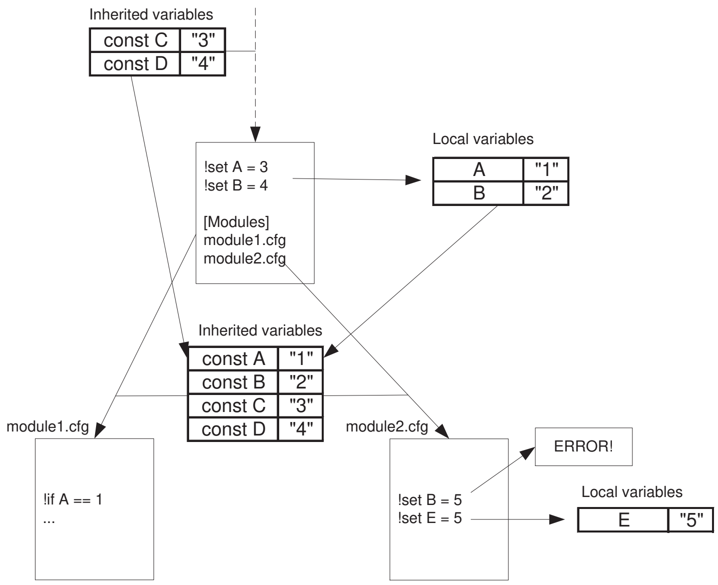

Figure 4.3: Example of how the variables are handled

variables defined in the file (A and B) in addition to the inherited variables (C and D) are merged together and passed to the included modules. In these files the variables are free to use ("!if A == 1" in the figure) but trying to override them is not possible and will produce an error ("!set B = 5" in the figure). Figure 4.3 shows that the inherited variables are regarded as constants and can not be overridden and that variables are only visible in leaf files. If this semantic is violated an error is issued.

As a consequence of the above defined variable handling the options to the set directive are obsolete and looses their meaning. Hence it is no longer possible to define a variable as irreplaceable, forced or local (mapping to the -i, -f, -l flags), it is already implicitly defined by the variable handling semantics. Also given that the files are parsed a single time the append flag (-a) is also obsolete and can be abolished as well as the perl flag (-e) since arbitrary Perl expressions is no longer applicable.

4.3 Grammar

Using the notation introduced in appendix C.3, this section will formally define a grammar for the description language. First the directive part is defined and then the languages residing in the sections are added. Before defining the actual grammar, table 4.1 gives a number of general terminal symbols used universally throughout the grammar.

Terminal	Regular Expression	Description
ID	[A-Za-z0-9-_]+	Identifier, typically identifies the name to a variable
Ref	"%"Identifier"%"	Variable reference
Opt	"-"[iflea]+	Option to set directive
Str	[.]+	String literal
QStr	"""[.]+"""	Quoted string literal
Drive	[A-Za-z]":\"	Identifier denoting the beginning of an absolute file path
RelPath	".\"	Identifying the beginning of a relative file path
NL	"\r\n" — "\r" — "\n"	New line

Table 4.1: General non terminal symbols

4.3.1 Directive grammar

Given the possible directives available in the language described in section 3 the following start symbol for the language can be defined:

Stmt → (Set | Unset | If | Ifdef | Ifndef | Include | Options)*

where each nonterminal on the right hand side corresponds to a directive in the language and *Stmt* is an arbitrary sequence (zero or many) of directives. Here follows the production rules for each of the directives.

Set directive
The set directive can be described by the following BNF productions:

Set → "!set" (Opt)? ID ("=" (Val | Func)+)? NL

Val → (Str | Ref)+

Func → ID "(" Param ("," Param())* ")"

Param → QStr | Ref | ID | Func

Hence the value assigned to the variable, which is optional to define, can be a arbitrary sequence of strings, variable references and functions, where *Func* corresponds to the predefined functions defined by the language described in section 4.2.4. A possible function application will be evaluated and its value concatenated with the rest of the value. In the current language the options to a set statement, here defined as the terminal *Opt*, have a special meaning. In the new language they are allowed by the grammar but simply ignored by the semantical analysis (however resulting in a warning message) since their meaning is obsolete. The grammar for a function also shows that they can be nestled.

If directive
the *if* directive can be described by the following BNF productions.

If → "!if" BExp NL Stmt ("!elseif" BExp NL Stmt)* ("!else" NL Stmt)? "!endif" NL

BExp → OrExp

OrExp → AndExp ("——" AndExp)*

AndExp → (EqExp | Func) ("&&" (EqExp | Func))*

EqExp → (ID ("==" | "!=") Val) | ("(" BExp ")")

The recursive call to *Stmt* from within the if statement shows the nestling capabilities of the construct. An if statement can have zero or many branching elseif's and an optional else before the statement is finalized with the keyword "!endif". The rules defined for the logical expression dictates that conjunction binds tighter than disjunction. Defining the precedence between conjunction and disjunction ensures an unambiguous grammar [1] as opposed to an ambiguous one. In an ambiguous grammar a sentence in the language can be derived with two different parse trees which is highly unsatisfactory. The grammar also allows the boolean expressions to be grouped together in parenthesis to change the precedence of the expression. For example !if A == 1 || (B == 2 && C == 3) gets a different meaning than !if (A == 1 || B == 2) && C == 3. The rules defined above map directly to the current implementation of the language with the exception of not allowing arbitrary perl expressions within a boolean expression.

Given a the above defined rule for an if statement, the following example,

```
!if A = 1
  !set X
!elseif A = 2
  !set Y
!else
  !set Z
!endif
```

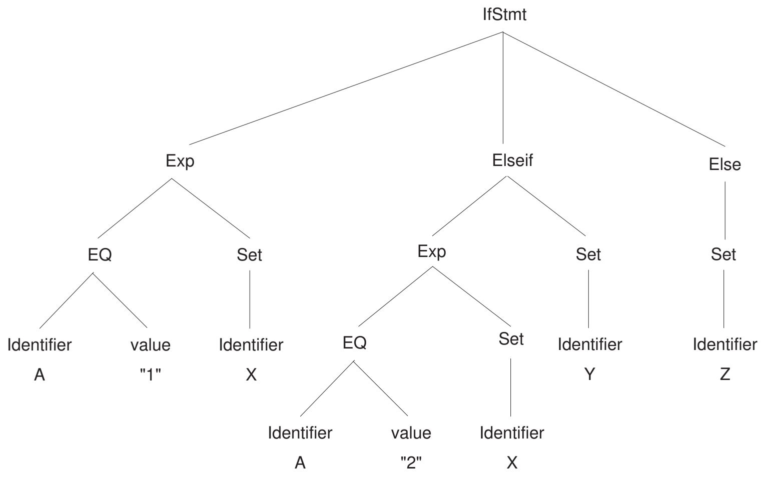

Figure 4.4: AST for an if statement.

would result in the AST depicted in figure 4.4. In figure 4.4 it can be seen how IfStmt has its branching elseif's and else's as children and they in turn define their respective boolean expressions and statements as their children.

Ifdef and Ifndef directive
The *ifdef* and *ifndef* directive can be described by the following BNF productions:

Ifdef → "!ifdef" ID NL Stmt ("!else NL Stmt)? "!endif" NL

Ifndef → "!ifndef" ID NL Stmt ("!else NL Stmt)? "!endif" NL

The grammar for the ifdef and ifndef statements are identical, differing only in keyword identifying the directive ("!ifdef" or "!ifndef"). The rules depicted here map to the current implementation of the language with the exception that SDE allows branching elseif's, whereas the new language do not. That feature is not considered adding any considerable functionality to the language and therefore removed.

Unset directive
The *unset* directive can be described by the following BNF production:

Unset → "!unset" ID NL

This rule is straightforward and maps directly to the language given by SDE.

Options directive
The *options* directive can be described by the following BNF production:

Options → "!options" ("-l")? (("CheckDescrConditionals" ("error" | "fatal")?)
| "MultipleSections" | "Silent") NL

This rule is straightforward and maps directly to the language given by SDE.

Include directive
The *include* directive can be described by the following BNF production:

IncludeStmt → "!include" ("-b")? FilePath NL

FilePath → (Ref | Drive | RelPath)? (Ref | ID)* (File)?

File → ID "." ID

The rule for an include statement is straightforward and maps directly to the current language. Defining a grammar for file paths in the description language is however a bit trickier. A file path can be either absolute or relative and therefore beginning with one of the regular expressions defined by *AbsPath* or *RelPath*. This however not the whole truth. The file path can also be a variable reference, either as a part of the path or the entire file path. The file path typically ends with *File*. This is however only the case when the full file path is written explicitly and the file path is not a directory. The *FilePath* grammar does not constrain the user to only write legal file paths but does resolve some ambiguities and provides the semantic analysis with a tree structure defining the full file path.

Define macro
This is a new directive introduced in NDL. Its objective is to replace the existing possibility of defining multi line values to variable and its syntax is identical with the exception of the new keyword "!define":

```
!define SOME_VARIABLE = << END_OF_DEF
This is one line
This is another line
END_OF_DEF
```

The reason for defining a new directive for multi line variables is that the value could be an arbitrary number of statements of the language. Because the variables are expanded after the syntax tree is build (see chapter 5 for details) it would contain an ambiguity and the expressions has to be re parsed in order to obtain the correct information. Therefore multi line variables are separated from "ordinary" variables by the use of this macro construct. In order to resolve the ambiguity in the AST this construct is handled by a preprocessing step described in more detail in chapter 5.4.

4.3.2 Adding a grammar for the sections

The sections each define a grammar of their own. Each section grammar contains the directives which are allowed anywhere together with a section specific syntax. However a unique grammar has not been defined for every section alone since many of them are very similar or even identical in their syntax. It is therefore feasible to group the sections that define similar syntax together in a class.

The grammar extended with sections results in the following start symbol instead of the one defined in the previous section 4.3.1:

Stmt → Directive | Section

where *Directive* is the previously defined *Stmt* describing the syntax for the directives. *Section* is the non-terminal defining the section part of the grammar. The full section grammar will not be elaborated here. Instead a pseudo grammar is introduced giving a general view of the real grammar. The non-terminal *Section* can be described as choice between the identified section classes:

Section → SectionClass1 | SectionClass2 | SectionClass3 | ...

where *SectionClass* is the non-terminal denoting the start of a new section followed by a number of section specific statements which also includes possible directives:

SectionClass → ("[SectionName1]" | "[SectionName2]" | "[SectionName3]" | ...)
(SectionStmt)*

SectionStmt → SectionSyntax | Directive

SectionName denotes the tag identifying the name of the section. Each section class contains one or many sections which all have identical syntax, here denoted with *SectionSyntax*.

Defining the grammar this way forces the parser to only parse constrained BNF productions adhering to a section class. As opposed to having a more general grammar where all section specific syntax is allowed everywhere, this solution eliminates ambiguities in the resulting AST and errors are discovered at an earlier stage. However, the start symbol (*Stmt*) for the grammar implies that a recursive calls to *Stmt* from within a conditional statement can be a any section and the syntax it defines. This should however not be allowed. Each section should only allow its specific syntax and the directives. However this has not been able to express with the BNF notation but is constrained by the tool which will described further in chapter 5.

It has not been possible to define a grammar for the entire language. Some sections have been found to be impossible to define a strict syntax for. Instead each line in these sections are kept intact in the AST, except for possible directives which are parsed as usual.

The detailed grammar for the sections will not be elaborated further in this report. The full BNF is however depicted in appendix E and it's left to the reader to further examine it.

4.4 Semantics

The semantics of a language is concerned with specifying the meaning, or behaviour of a program. Having a well defined semantics can simplify the implementation and reveal ambiguities in the language. This section defines a formal semantic, where applicable and feasible, for the description language.

Given the grammar and syntax defined in the previous section the following meta-variables, that will be used to range over *constructs* of each categories, are identified:

x will range over variables, **Var**

s will range over values, **String**

b will range over boolean expressions, **Bexp**

S will range over statements , **Stmt**

The meta-variables can be primed or subscripted. The BNFs for the individual constructs are given in section 4.3.

A **definition** consists of a finite subset X $\subseteq$ **Var** and a function:

$\delta : x \mapsto \textbf{String} \cup \{ \underline{\text{undef}} \}$

δ(y) = $\underline{\text{undef}}$, means that y is defined, but y's value is not defined.

A **state** $(\nu,(X,\delta))$ consists of:

1. A finite set $\nu \subseteq \mathbf{Var}$ of illegal variables where, $\nu \cap x = \emptyset$
2. A definition $\delta : x \mapsto \mathbf{String} \cup \{ \underline{\mathrm{undef}} \}$

The execution of a statement S changes the **state**:

$$\mathbf{Eval} [\![\text{ S }]\!] : \mathbf{State}_{error} \rightarrow \mathbf{State}_{error}$$

where **State** is a set of states and $\mathbf{State}_{error}$ denotes a state resulting in a possible error:

$$\mathbf{State}_{error} = \mathbf{State} \cup \{ \underline{\mathrm{error}} \}$$

Eval is strict in the sense that $\mathbf{Eval} [\![\text{ S }]\!]$ maps error to error.

Before defining the semantics for each individual statement in the language, lets define the semantics for executing an arbitrary sequence of statements:

$$\mathbf{Eval}[\![S_1,\ S_2\ ]\!](\nu,(X,\delta)) = \mathbf{Eval}[\![S_2]\!](\mathbf{Eval}[\![S_1]\!](\nu,(X,\delta))$$

where S_1 and S_2 are two arbitrary statements of the language.

4.4.1 Directive semantic

A value s can be expressed as given in the following BNF production rule:

$$\text{s} \rightarrow \text{c} \mid \%\text{x}\% \mid \text{ss}$$

Here c denotes string expressed literally and %x% a reference to a variable. A value can be concatenated in arbitrary long sequences of these terminal symbols. The evaluation of a value can be expressed as follows:

$$\mathbf{EString}[\![s]\!] : \mathbf{State}_{error} \rightarrow \mathbf{String}_{error}$$

And for the individual rules:

$$\mathbf{EString}[\![c]\!](\nu, (X, \delta)) = c$$

$$\mathbf{EString}[\![\%x\%]\!](\nu, (X, \delta)) = \begin{cases} \underline{error} & if\, x \notin X \\ \delta'(x) & if\, x \in X \end{cases}$$

$$\textbf{EString}[\![s_1 s_2]\!](\nu, (X, \delta)) = \textbf{EString}[\![s_1]\!](\nu, (X, \delta))\,^\wedge\,\textbf{EString}[\![s_2]\!](\nu, (X, \delta))$$

Now that a semantic for values have been defined it is possible to define a semantic for the *set* directive:

$$\textbf{Eval}[\![\text{set x}]\!](\nu,\ (X, \delta)) = \begin{cases} \underline{error} & if\, x \in \nu \\ (\nu, (X \cup \{x\}, \delta')) & if\, x \notin \nu \end{cases}$$

$$\begin{array}{l} \text{where,} \\ \delta'(y) \ = \ \delta(y)\, for\, y \in X \backslash \{x\} \\ \delta'(x) \ = \ \underline{undef} \end{array}$$

and for an assignment:

$$\textbf{Eval}[\![\text{set x = y}]\!](\nu,\ (X, \delta)) = \begin{cases} \underline{error} & if\, x \in \nu \\ (\nu, (X \cup \{x\}, \delta')) & if\, x \notin \nu \end{cases}$$

$$\begin{array}{l} \text{where,} \\ \delta'(y) \ = \ \delta(y)\, for\, y \in X \backslash \{x\} \\ \delta'(x) \ = \ \delta'(y) \end{array}$$

Hence executing the *set* directive will result in an error provided that the variable is already defined and belongs to ν, otherwise the variable will be given a **String** or an undef value and added to δ resulting in a change of the **State**. The opposite applies for the *unset* statement:

$$\textbf{Eval}[\![\text{unset x}]\!](\nu, (X, \delta)) = \begin{cases} \underline{error} & if\, x \notin X \\ (\nu \cup \{x\}, (X \backslash \{x\}, \delta')) & if\, x \in X \end{cases}$$

Given the semantic for the *set* statement, defining the semantics for the *ifdef* statement is straightforward.

$$\textbf{Eval}[\![\text{ifdef x \{S\}}]\!](\nu, (X, \delta)) = \begin{cases} \textbf{Eval}[\![S]\!](\nu, (X, \delta)) & \text{if } x \in X \\ (\nu, (X, \delta)) & \text{if } x \notin X \end{cases}$$

And equivalent for *ifndef*:

$$\textbf{Eval}[\![\text{ifndef x \{S\}}]\!](\nu, (X, \delta)) = \begin{cases} (\nu, (X, \delta)) & \text{if } x \in X \\ \textbf{Eval}[\![S]\!](\nu, (X, \delta)) & \text{if } x \notin X \end{cases}$$

Returning to the grammar for a *BExp* given in the previous chapter and slightly revising it, results in the following BNF:

$$b \rightarrow v = s \ \mid \ v \neq s \mid b_1 \,\&\&\, b_2 \ \mid \ b_1 \parallel b_2$$

Note that the operators used for the equality and not equality are substituted but still bear the same meaning. Given the above defined BNF, it's possible to define the semantics for **Bexp**:

$$\mathbf{EBool}[\![v\ =\ s]\!](\nu,(X,\delta)) = \begin{cases} \mathbf{true} & \text{if } \delta(v)\ =\ \mathbf{EString}[\![s]\!](\nu,(X,\delta)) \\ \mathbf{false} & \text{if } \delta(v)\ \neq\ \mathbf{EString}[\![s]\!](\nu,(X,\delta)) \end{cases}$$

$$\mathbf{EBool}[\![v\ \neq\ s]\!](\nu,(X,\delta)) = \begin{cases} \mathbf{true} & \text{if } \delta(v)\ \neq\ \mathbf{EString}[\![s]\!](\nu,(X,\delta)) \\ \mathbf{false} & \text{if } \delta(v)\ =\ \mathbf{EString}[\![s]\!](\nu,(X,\delta)) \end{cases}$$

$$\mathbf{EBool}[\![b_1\ \&\&\ b_2]\!](\nu,(X,\delta)) = \begin{cases} \mathbf{true} & \text{if } \mathbf{Eval}(b_1)\ =\ \mathbf{true} \text{ and } \mathbf{Eval}(b_2)\ =\ \mathbf{true} \\ \mathbf{false} & \text{if } \mathbf{Eval}(b_1)\ =\ \mathbf{false} \text{ or } \mathbf{Eval}(b_2)\ =\ \mathbf{false} \end{cases}$$

$$\mathbf{EBool}[\![b_1\ \ \|, b_2]\!](\nu,(X,\delta)) = \begin{cases} \mathbf{true} & \text{if } \mathbf{Eval}(b_1)\ =\ \mathbf{true} \text{ or } \mathbf{Eval}(b_2)\ =\ \mathbf{true} \\ \mathbf{false} & \text{if } \mathbf{Eval}(b_1)\ =\ \mathbf{false} \text{ and } \mathbf{Eval}(b_2)\ =\ \mathbf{false} \end{cases}$$

where **EBool** denotes the evaluation of a boolean expression:

$$\mathbf{EBool}[\![b]\!] : \mathbf{State}_{error} \rightarrow \mathbf{Bool}_{error}$$

Given the semantics for boolean expressions, defining the semantics for *if* statements is straightforward:

$$\mathbf{Eval}[\![\text{if } b\ \{S\}]\!](\nu,(X,\delta)) = \begin{cases} \mathbf{Eval}[\![S]\!](\nu,(X,\delta)) & \text{if } \mathbf{Bexp}[\![b]\!]\ =\ \mathbf{true} \\ (\nu,(X,\delta)) & \text{if } \mathbf{Bexp}[\![b]\!]\ =\ \mathbf{false} \end{cases}$$

The semantics for an *elseif* branch is identical to an *if* statement. However the dangling elseif's and else branches are not executed if one of the preceding conditionals have evaluated to true.

The *include* statement involves importing a file and executing its contents with the so far obtained illegal variables ν and the variable definitions δ. The semantics for the statement can be expressed as given in the following somewhat informal rule:

$$\mathbf{Eval}[\![\text{include file}]\!](\nu,\ (\text{X},\ \delta)) = \mathbf{Eval}(\text{S})(\nu,\ (\text{X},\ \delta))$$

where S denotes the contents of file.

The example below illustrates how the above defined semantics for the *set* and *unset* statements is applied. In the example, "{...}" denotes the set of illegal variables and "[...]" denotes the map from legal variables to values.

	$\{\}$	$[]$
set X		
	$\{\}$	$[X \mapsto \text{undef}]$
set Y = A		
	$\{\}$	$[X \mapsto \text{undef}$ $Y \mapsto \text{"A"}\ ]$
set Z = %Y%		
	$\{\}$	$[X \mapsto \text{undef}$ $Y \mapsto \text{"A"}$ $Z \mapsto \text{"A"}\ ]$
set Z = %Z%%Z%C		
	$\{\}$	$[X \mapsto \text{undef}$ $Y \mapsto \text{"A"}$ $Z \mapsto \text{"AAC"}\ ]$
unset Y		
	$\{Y\}$	$[X \mapsto \text{undef}$ $Z \mapsto \text{"AAC"}\ ]$

4.4.2 Section semantic

The most interesting part of the section grammar to define a formal semantic for are the sections listing files, simply because the main part of the description files syntax reside in such sections. As seen from the BNF in appendix E each row in these section define a file. Some sections also defines the possibility of expressing zero or many file specific options to the file. In addition to the file listing sections there are sections that provide functionality to override a file listed in a file list section, the file list modification sections. A file listed in a file list modification section modifies the contents of a previously defined file list section in the sense that it is removed from the list of defined. Hence the semantic for these two types of sections resembles the semantic for the set and unset directive to great extent. But instead of having a variable name mapping to a value we have a file mapping to a possible file option and instead of blocking a variable for further use, a file is removed.

A file listed in a file list section, simplifying the grammar slightly, can be expressed as a string s followed by another string s, the first string mapping to the file path and the second string mapping to a possible options to the file. It should be mentioned that the majority of the sections merely list files without any options but the semantic is expressed assuming options to all sections which gives a more general semantic. If the option is empty, then the entry in the file list is also empty.

IncSec → File | File IncSec

File → s NL | s s NL

The evaluation of a file listed in a file list section, here denoted with **EIncSec**, will result in the file path and the option being added to a list of file descriptions, **fileDescr***:

$$\mathbf{EIncSec}[\![s_1]\!](\nu, (X, \delta)) = < (\mathbf{EString}[\![s_1]\!](\nu, (X, \delta)), \text{""} >$$

$$\mathbf{EIncSec}[\![s_1, s_2]\!](\nu, (X, \delta)) = < (\mathbf{EString}[\![s_1]\!](\nu, (X, \delta)), \mathbf{EString}[\![s_2]\!](\nu, (X, \delta))) >$$

$$\mathbf{EIncSec}[\![File\, IncSec]\!](\nu, (X, \delta)) = < \mathbf{EIncSec}[\![File]\!](\nu, (X, \delta)) \,^\wedge\, \mathbf{EIncSec}[\![IncSec]\!](\nu, (X, \delta)) >$$

$$\mathbf{EIncSec}[\![IncSec]\!] : \mathbf{State} \rightarrow \mathbf{fileDescr*}$$

where

$$\mathbf{fileList} = \mathbf{fileName} \times \mathbf{Options}$$

In extension of having the possibility to list a file it is possible to subsequently remove the file from the list. The grammar is simply a string denoting the file to be overridden:

ModSec → File | File ModSec

File → s NL

The semantics for a file listed in a file list modification section is simply removing it from the list of files. Hence:

$$\mathbf{EModSec}[\![s]\!] : \mathbf{fileList*} \rightarrow \mathbf{State} \rightarrow \mathbf{fileList*}$$

and

$$\mathbf{EModSec}[\![\underline{del} s]\!]\, \mathbf{fileList*} = \mathbf{fileList*} \backslash \{s\}$$

where *del* denotes the deletion of a file s from the list of files (**fileList**).

Finally an example to illustrate the defined semantic for file list sections and file list modification sections. As in the previous example "{...}" denotes the set of overridden files and "[...]" denotes the list of files.

	{}	[]
[SourceFiles] .\dir\file1.c .\dir\file2.c AVR=(-z9) .\dir\file3.c	{}	[.\dir\file1.c , ”” .\dir\file2.c , ”AVR=(-z9)” .\dir\file3.c , ””]
[-SourceFiles] .\dir\file1.c	{.\dir\file1.c}	[.\dir\file2.c , ”AVR=(-z9)” .\dir\file3.c , ””]

The file list modification sections are always evaluated last, hence would ”[-SourceFiles]” in the example have been followed by another ”[SourceFiles]” listing file.c again it would still not be not be part of the file list.

4.5 Summary

This chapter has described the new description language (NDL) which is based on the current implementation (CDL). The syntax of CDL is to great extent kept as it is in NDL. The main difference is that NDL has a formal definition of the grammar whereas CDL does not. In other words NDL implements a more strict syntax with defined rules and constrained behaviour. Another main difference between CDL and NDL is that the latter has a whole new way of processing the files and parsing the language. Where CDL is implemented in a handwritten parser, NDL uses a compiler compiler tool generated parser. Also the way the files are processed differ between the two languages. When it comes to the semantic of the language it is practically identical with one key difference; the variable semantic is changed regarding the rules concerned with the variables scope rules.

The main differences between the old and the new language are summarized in table 4.2.

Hence the functionality provided by CDL is kept intact. The key difference is the way the languages are defined and handled.

CDL	NDL
No formal grammar	Formal grammar
Flat file structure	File tree intact
Extensive Perl availability	Constrained Perl availability
Hand written parser	Generated parser
Global variables	Constrained variable scope
File parsed multiple times	File parsed one time

Table 4.2: Main differences between CDL and NDL

Chapter 5

The Analysis Tool

5.1 Introduction

One of the objectives of this master thesis project is to develop a analysis tool capable of performing syntactic and semantic analysis on the new language in addition to different analysis on individual description files or a file tree. This chapter describes the detailed design and implementation of that tool.

One of the prerequisites when implementing the tool is that it should be partly generated with the help of a compiler compiler tool as opposed to the handwritten parser that exists today. A compiler compiler, as the name implies, is a tool that support the development of a compiler. Such a tool typically takes a formal grammar and generates the parser and possibly also the syntax tree. This simplifies the development process significantly and reduces many risk that having a handwritten parser produce. An overview of how the chosen compiler compiler tool works is given in appendix D.

This chapter is structured as follows; section 5.2 describes the main objectives of the analysis tool, section 5.3 gives an overview of the analysis tool design, section 5.4 to section 5.6 describe the individual parts of the tool, which includes a preprocessor, front end and back end.

5.2 Tool objectives

The main objectives of the analysis tool is to issue warnings and errors messages where the language is violated regarding its syntax and semantic. These messages can be categorized into three different levels of errors:

1. **Lexical errors**, are thrown by the lexer whenever the JavaCC token

manager detects a problem. See examples of errors in section 5.5.1.

2. **Syntactical errors**, are thrown by the parser whenever it detect a problem in the input token stream that violate the grammar of the language. See error examples in section 5.5.2.

3. **Semantic errors**, are issued by the semantic analysis when constructs that do not follow the language semantic are found. See error examples in section 5.6.3.

Another objective of the tool is to implement different analysis of the description files. The analysis incorporated into this version of the tool are:

- **File analysis**, analysis of the file names that reside in the file list sections. A more detailed description is given in section 5.6.5.

- **Variable analysis**, analysis of the variables that reside in the description files. A more detailed description is given in section 5.6.6.

All three parts of the tool issues errors and warnings when constructs that violate the defined syntax or semantic is found. The lexer and parser in the preprocessor and front are both capable of finding lexical error and syntactical error whereas the semantic analysis pinpoints constructs that violate the defined semantic. A lexical error is

5.3 Tool architecture

The analysis tool architecture resembles the design of a compiler to great extent. The general notions behind the design of compiler is introduced in appendix C.2. and figure C.1 shows the typical phases. The analysis tool is however not a complete compiler. It is not of interest to translate the source text into machine code and therefore the back end part of a compiler will not be further elaborated here. The part of interest when implementing the analysis tool is mainly the *front end* part, with the exception of the *intermediate code generation* phase. Revising figure C.1 to fit the purposes of the analysis tool, figure 5.1 depicts the phases applicable.

An elaborative and detailed architecture sketch is illustrated in figure 5.2. The phases needed to implement the analysis tool is mainly the typical front end part of a compiler. However this notion have been revised when defining the design of the analysis tool. Here the front end part have been split into one front end and one back end extended with a preprocessing step. The front end part takes care of parsing and tree building while the back end is responsible

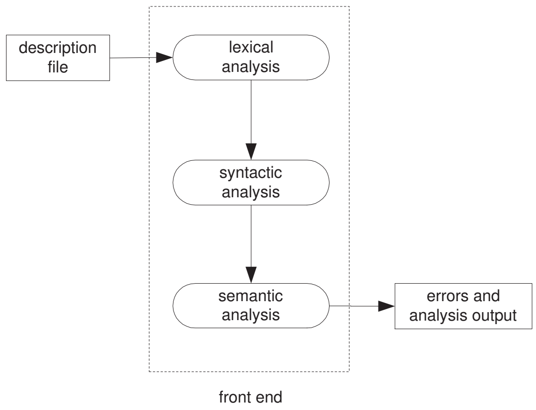

Figure 5.1: Phases in the analysis tool mapping to a typical compiler

for the semantic analysis. Hence the notions from section C.2 is slightly revised but not the overall meaning.

The analysis tool is made up of three separate parts. The input description file is first passed to a *preprocessor* responsible for performing some initial parsing of the file to resolve some constructs available in the language. The preprocessed file is then send to the *front end* which implements the actual lexer and parser. The front end builds an AST representation of the file which is passed on to the *back end*. The back end implements the semantic analyser but also different analysis of the file.

All three parts of the tool issues errors and warnings when constructs that violate the defined syntax or semantic is found as described in section 5.2. The lexer and parser in the preprocessor and front are both capable of finding lexical error and syntactical error whereas the semantic analysis pinpoints constructs that violate the defined semantic but also a file and variable analysis.

5.4 Preprocessor

As described in chapter 3 and further revised in chapter 4 the language offers a macro directive identified with the keyword "!define" which makes it possible to assign multi line values to a macro which can then be referred later in the file. The value can theoretically consist of any type of expression adhering to the description language. However having the actual parser taking care of this type of construct would be messy. It would be more sufficient if the parser only handles the expanded macro value and don't have to bother about the macro syntax, generally because the parser not knowing what actually resides

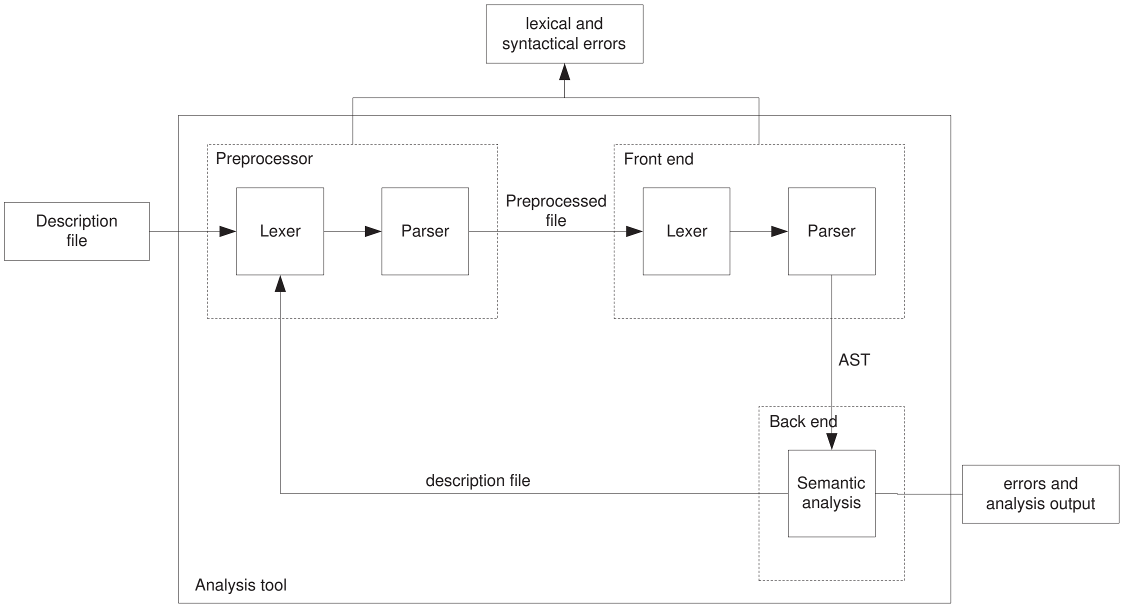

Figure 5.2: Analysis tool architecture

in the macro causes an insufficient ambiguity in the AST. Therefore it is feasible to introduce a pre parsing step or a preprocessor that can handle the macro constructs and expand them before the main parser continues processing the file. Figure 5.4 shows the functionality provided by the preprocessor step.

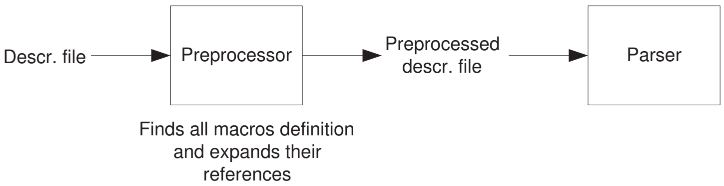

Figure 5.3: The preprocessor step

The preprocessor takes a description file as input and produces a preprocessed file which is passed on to the front end. The preprocessor searches the file for macro definitions and macro references. The macro values are put in a list and on the encountering of a macro reference the macro value is extracted from the list and replaces the macro references. Hence in the preprocessed file all macro occurrences are removed and replaced with blank lines and all references with their corresponding value. An example is depicted in figure 5.4.

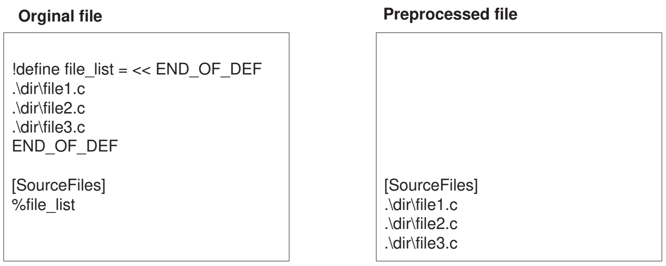

Figure 5.4: A preprocessed file and corresponding non preprocessed file

Since the references might be expanded with multi line values the line numbering of the original file might be corrupted in the preprocessed file and would result in an possible error pointing to the wrong line in the original file. This has been solved by the preprocessor instrumenting the code with a line offset construct representing the number of lines in the macro, which is then handled by the front end. The line offset integer is simply subtracted from the actual line number in the front end resulting in the correct line being referred to when finding an error. An example is illustrated in figure 5.5.

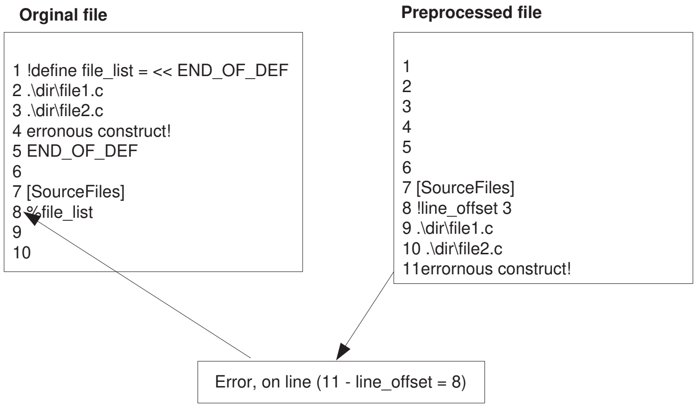

Figure 5.5: The solution to corrupted line number

The language also offers the possibility of dividing long lines into several using a special continuation mark denoted by "!\" which is placed last on the line resulting in the following line being concatenated with the preceding one. This language construct is also handled by the preprocessor, simply concatenating all occurrences of continuing lines and replacing them with blank lines.

5.5 Front End

The front end part of the analysis tool contains the actual parser of the language together with a lexer. The front end parses the file following the rules dictated by the grammar and generates an AST representation of the file contents.

The lexer and parser are generated with the help of a compiler compiler. The compiler compiler chosen to generate the core of the analysis tool is the Java Compiler Compiler (JavaCC) [2]. JavaCC is an open source project and the components it generates are pure java code. The motivation behind the choice of this particular compiler compiler is that it is open source but the fact that it is Java based is also a factor, because EMP is planning to move their software development environment to the Eclipse platform which also is Java based and therefore making the analysis tool integration with the software environment easier. It is recommended to read appendix D which gives an overview of JavaCC before continuing.

The grammar described in chapter 4 resides in a perfect theoretical world. The real world is however not quite that utopian and applying the grammar straight on to the compiler compiler would not suffice. The grammar has to be modified to some extent to fit the real world. The subsequent sections will elaborate on the implementation of the front end and the different problems arising when trying to apply the grammar into the tool.

5.5.1 Lexical analysis

The big issue when defining the tokens for the lexer has been to match a string value to a variable correctly since it can basically be any sequence of characters and there are no defined delimiters denoting the start and end of the string.

A first approach was simply to define a regular expression matching anything until a new line is encountered:

```
TOKEN:
{
  < VALUE: (~["\r","\n"])* >
}
```

The "~" character tells the lexer to match anything but the characters enclosed in parenthesis. This solution however proved to be very dangerous because it is very likely that the lexer will match every line in the file as a VALUE token. To solve this the JavaCC notion of lexical states can be used. Every time an assignment operator is found corresponding to a set statement the lexer is forced to switch to another lexical state (e.g. IN_VALUE) where only the token VALUE is applicable:

```
<IN_VALUE>
TOKEN:
{
  < VALUE: (~["\r","\n"])* > : DEFAULT
}
```

This solution however implies that further lexical states have to be introduced because moving to the IN_VALUE state should only be performed when the assignment operator is found in a set statement. Also the value is not always simply a string. It can also be a variable reference or a function which the above given token definition would match but leaving no trace in the resulting AST of such occurrences, resulting in a unwanted complexity when resolving the actual value. Having many lexical states also makes the reading of the grammar file significantly more difficult and debugging harder. Another setback using an extensive amount of lexical states is the rapid growth of the generated parser in terms of its size, which at least for some of the intermediate solutions elaborated on in this thesis gave a unrecoverable memory overflow error in the compiler. It is therefore feasible that the final solution uses as few lexical states as possible and avoids the "match everything" token defined above.

Therefore, instead of defining one single token for the matching of a value, many tokens are used that each define a single character. Each character matched by the lexer is then concatenated together by the parser to form the original string. When implementing such iteration over tokens, the JavaCC feature of writing pure Java code productions is very sufficient. When the parser encounters an assignment the suitable Java method is invoked taking care of the value parsing, described by the following somewhat simplified code:

```
void matchValue() {
  Token t;
  String s;
  while(true) {
    t = getToken(1);
    if(t = NL) {
      stringLiteral(s);
      break;
    }
    else if(t == VARIABLE_REF) {
      stringLiteral(s);
      s = "";
      variableReference();
    }
    else if(t == FUNCTION_IDENTIFIER) {
      stringLiteral(s);
```

```
      s = "";
      function();
    }
    else {
      s = s + t;
    }
    getNextToken();
  }
}
```

The lexer defines an API that enables access to to the token sequence directly. Examples of such API routines are `getToken(int index)` and `getNextToken()` used in the code sample above. `getNextToken()` consumes a token from the input stream while `getToken()` merely peeks at index-th token from the current token ahead in the token stream. The keywords denoted by capital letters in the `matchValue()` method are the identifying names for defined tokens. The method iterates over the token stream until a new line is found (NL). If a variable reference or function is found the suitable BNF production rule is invoked, otherwise the token is considered a part of a string and concatenated with the previous token.

This solution avoids using lexical states as well as tokens matching everything. The `matchValue` method is also invoked when making comparisons in a boolean expression to a conditional statement.

However it should not be regarded as an disadvantage to use lexical states since they provide a very powerful feature to the lexer. The recommendation is not to use to many of them. The lexers final solution also has lexical states, more precisely two, not counting the DEFAULT state. When finding a section that has no defined grammar it is desired to match a whole line as one token and thereby using "match everything" token. To disable the possibility of the parser matching other expressions in the file as such a token it is sufficient to divide the matching of section syntax into two states. The DEFAULT states matches the section syntax where a grammar is defined and IN_MISCSECTION matching the rest of the sections. However the directives should still be parsed in these "miscellaneous" sections. Therefore the tokens needed in the directives are also applicable in the IN_MISCSECTION state. Whenever a miscellaneous section is found the lexer forces a switch to IN_MISCSECTION. In IN_MISCSECTION, besides the tokens needed by the directives, the following token is applicable:

```
<IN_MISCSECTION>
TOKEN :
{
  < MISC_STMT : ~["[","!","%"] (~["\r","\n","%"])* >
}
```

This token matches everything until new line except those tokens beginning with "!" denoting the beginning of a directive, "[" denoting the beginning of a section or "%" marking the beginning of variable reference. Once a new section is found, depending on the type of section, the lexer either moves to the DEFAULT state or stays in the IN_MISCSECTION state.

The second lexical state implemented in the lexer is called COMMENT_NA, which is short for comment not applicable. Comments in the description language are normally denoted with the "#" character which marks the start of the comment. However the comments are not entirely context free. Actually SDE only considers a comment to be a comment if it is stated in the beginning of a line. Comments in the new description language are defined as a SKIP token, which means that comments will simply be skipped by the lexer wherever they are stated and will not be passed to the parser. This is however not feasible considering that the "#" character should not always be matched as the start of the comment. This is considered particularly important when matching a value or the arguments to a file option. Therefore when parsing these rules the parser forces the lexer to switch to new state, COMMENT_NA, where the comment token is not applicable and thereby not matching the "#" character as a beginning of comment. However anywhere else in the grammar comments are applicable. Being able to force a switch in the lexer from the parser implies that the lexer and parser are not two sequential steps but rather execute in parallel. The lexer is however always ahead of the parser. Forcing a switch of lexical state from the parser should therefore be considered with care because the lexer might be well ahead of the parser owing to lookahead and the switch to the new lexical state might not be applied to the token one had in mind.

The implementation of the rest of the lexer is pretty straightforward. Since the regular expressions for the tokens are matched in their order of occurrence in the grammar file it is important that tokens are defined in specific order to ensure the correctness of the lexer. The order in which the tokens are defined are described below:

1 Keywords

2 Skip tokens

3 Identifiers

4 New line

5 Single characters

6 "Match anything" token

Hence all keywords in the language, such as "!if" or "[SourceFiles]" are put in the beginning since it is imperative that the language keywords are not matched as anything else.

Error reporting

Whenever the token manager in JavaCC detects a problem it throws an exception of type `TokenMgrError`. This occurs if a token does not obey any of the rules dictated by the lexer, for example if the token contains a character that distort the meaning of the token. For example:

```
!s&et A
```

will cause the lexer to print the following error message:

```
Lexical error at line 6, column 7.  Encountered: "e" (105), after : "&"
```

5.5.2 Syntactical analysis

The syntactical analysis or the parser is concerned with transferring the grammar rules defined in section 4.3 to the JavaCC grammar file.

Avoiding left recursion
Although the grammar was considered not have any left recursion and no rewriting of the production rules are necessary it could still be interesting in elaborating on a grammar that would. Consider the revised and more compact writing of the BNF production for a boolean expression:

$$b \rightarrow v = s \mid v \neq s \mid b \,\&\&\, b \mid b \,\|\, b$$

This BNF would result in JavaCC throwing an error that the grammar contains left recursion because the non-terminal b contains a recursive reference to itself that is not preceded by something that will consume tokens. The parser produced by JavaCC works by recursive descent. Left-recursion is banned to prevent the generated subroutines from calling themselves recursively ad-infinitum. The left recursion is prevented by using the rules defined in section 4.3. These rules have no recursive calls to themselves and thus no left recursion.

Alternative approaches
Before defining the final solution of the description language parser, a couple of other approaches where tried out but finally rejected. A first approach was to use two separate parsers, one for the directives and one for the sections. The directive parser simply left the section syntax intact only analysing the directives. The section code where then extracted from the generated AST and fed back to the section parser. The section parser was initialized with different lexical states depending on the type of section to be parsed. The section AST was then added to the original AST build by the directive parser. This approach is illustrated in figure 5.6.

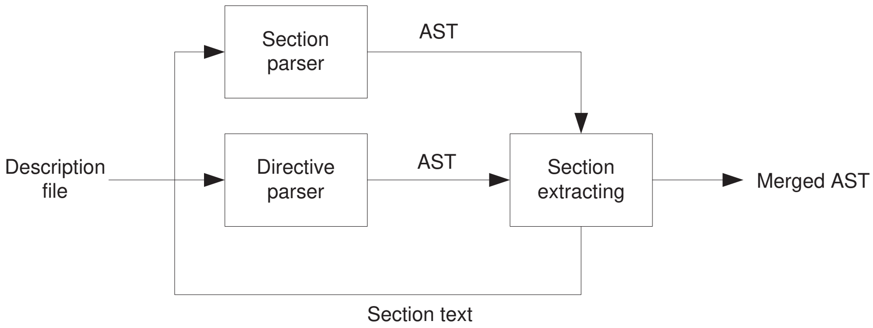

Figure 5.6: The approach of having two parsers

This approach however is a bit redundant since the section parser still have to parse the directives residing in the sections. So both parsers had to implement the same grammar to some extent, namely the directive grammar.

Also a more general grammar was elaborated that simply allowed the section specific syntax to be context free, meaning that all special syntax residing in the sections could be written anywhere and the analysis whether or not an expression belonged to the right section was left to back end to resolve. Having a general grammar gives a fairly easy parser and but does imply that a lot of work should be done by the semantic analysis in the back end.

Final solution

Although having a general grammar for the parser would provide an easily implemented parser, it's disadvantages was still considered to overshadow it's advantages. Having a grammar that constrains the parser to only consider the section specific syntax in it's right context is the final and chosen solution since it can resolve a lot of errors on a early stage, releasing the back end from the burden of having to check every single expression with respect to to its context.

The solution is however not as trivial as the more general one. If the possible statements in the grammar were free from recursive calls it would be fairly simple. Then each section could invoke a BNF production that only allowed the desired syntax, illustrated in the sample code below for a file list section:

```
void Section() :
{}
{
  ...
  <INCLUDEFILES> <NL> FileListStmt()
  ...
}
```

```
void FileListStmt() :
{}
{
  (FileList() | Directive())*
}

void FileList() :
{}
{
  FilePath() <NL>
}
```

However problem arises when `Directive()` is a conditional. The branch to the conditional could basically be any statement in the grammar, which is not suitable. The available BNF productions that can be invoked from the conditional somehow have to be forced to be only the ones allowed for the section that is being parsed. To solve this a state for each section class is introduced, which are not to be confused with the notion of lexical states. The state, which is just an integer constant, is passed on to every directive that is invoked from the applicable section class production. If the directive is a conditional and a branch is found the state is used to force the parser to only parse the productions available for the given section. Below the example from above is extended with the notion of section states:

```
void Section() :
{}
{
  ...
  <INCLUDEFILES> <NL> FileListStmt()
  ...
}

void FileListStmt() :
{}
{
  (FileList() | Directive(FILELIST))*
}

void FileList() :
{}
{
  FilePath() <NL>
}
```

Here an integer, FILELIST, denoting that the section is a file list section is passed on to the directive production:

```
void Directive(int sectionState) :
{}
{
    IfStmt(sectionState)
  | IfdefStmt(sectionState)
  | IfndefStmt(sectionState)
  | SetStmt()
  | UnsetStmt()
  | IncludeStmt()
  | OptionsStmt()
  | LineOffset()
  | <NL>
}
```

Each conditional statement (if, ifdef and ifndef) calls a Java code production with the section state integer, which acts as a switch invoking the appropriate BNF production depending on the given state:

```
JAVACODE
void sectionStmt(int sectionState) {
  switch(sectionState) {
    case FILELIST: FileListStmt(); break;
    ...
  }
}
```

This way the parser is forced to only allow the section specific syntax for the given section. The implementation is not entirely straightforward since the used Java code productions (like `sectionStmt`), are considered a black box by JavaCC which somehow does its task and the normal regular expression constructs can't be invoked on them, for example constructs like repetition and optional expansion. These implementation work arounds will however not be further elaborated here.

The rest of the parser implementation is pretty straightforward mapping almost without exception to the grammar defined in chapter 4.

Error reporting

Whenever the parser detects a problem it throws an exception of type `ParseException`. This occurs if the token stream provided from the lexer to the parser does not obey the rules dictated by the grammar. An example is if the the boolean expression to an if directive is corrupt:

```
!set = A
```

Here the variable identifier is clearly forgotten and the parser will print the following error message in the case of the above construction:

```
Encountered "=" at line 4, column 6.
Was expecting one of:
    <OPT> ...
    <IDENTIFIER> ...
```

The error message is informing us that the parser is expecting either an option (¡OPT¿) or an identifier to proceed the assign character ("="). Another example is if for example the section name is misspelled:

```
[SorseFiles]
```

This will produce the following error output from the lexer:

```
Encountered "[" at line 2, column 1.
Was expecting one of:
    <EOF>
    "!if" ...
    "!set" ...
    "!unset" ...
    "!ifdef" ...
    "!ifndef" ...
    "!include" ...
    "!options" ...
    "!line_offset" ...
    "[SourceFiles]" ...
    "[IncludeFiles]" ...
    "[BuildFiles]" ...
    "[LinkFiles]" ...
    ...
```

where the parser is telling us that another token has to be used, namely the ones dictated by the grammar production rule.

5.5.3 Building the AST

With the use of JJTree it is possible to, besides the generated parser, to produce an AST representation of the parsed file as explained in section D.4. Since it is not desired to produce a parse tree node for each nonterminal in the grammar the default behaviour of JJTree has been overridden and only nonterminals where it is explicitly stated generates a node.

As explained in section D.4, JJTree provides a very simple way of generating AST representations by simply invoking the "#" character followed by the name of the node with a possible argument. Implementing the AST has been straightforward. Given below is an example of description file implementation and the AST it produces following the rules implemented in the grammar file.

```
!set A = 1
!set B = 2

!include .\file.cfg

!if A == 1 && B != 2
  !set C = 1
!elseif A == 1 || B == 2
  !set C = 2
!else
  !set C = 3
!endif

[SourceFiles]
.\dir\file1.c
.\dir\file2.c AVR=(-z9)

[MakeRules]
copy #path $(INC)#name#ext >nul
```

results in the AST depicted in appendix F. Every children is denoted by the node being indented relative to its parent. As can be seen every section is a child directly to the start symbol and all syntax residing in the section is regarded as children to that section. The "[MakeRules]" section, which is such a section where no formal grammar has been defined, is simply left untouched.

5.6 Back End

The back-end is the part that traverses the AST and outputs error messages for syntactically and semantically incorrect constructions in the description file as described in section 5.2. The back-end is also responsible for producing analysis result when applicable. Figure 5.7 depicts a general notion of the back end.

Input to the analysis tool back-end is the generated AST, which is traversed a

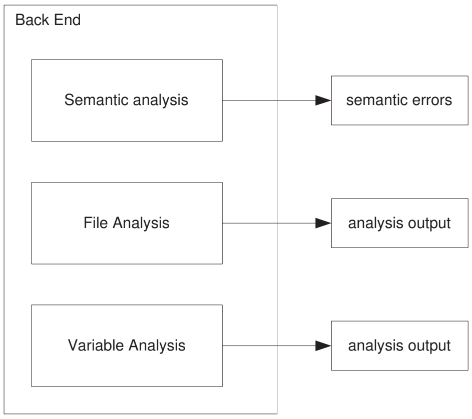

Figure 5.7: Block diagram depicting the back end

number of times to produce the analysis tool output. For this purpose a design pattern called the visitor pattern technique is suitable. The back end is easily extendable with new back end applications, which handle the AST differently. The different back ends each produce different output, i.e. their task is to check different semantic aspects of the language.

Besides the visitors classes the back end also contains a number of help classes. A class diagram of the back end is depicted in figure 5.8.

5.6.1 The Visitor Pattern Technique

In the context of JavaCC and JJTree, the parser parses the description file and returns a handle to a node that is the root of an AST; that AST represents the input parsed. After the parser and tree building are done, the AST then typically needs to be traversed and processed in one or more ways. Each of these operations involves walking the AST, starting at the root and executing suitable code for each type of AST node encountered. It is here the *visitor design pattern* [7] comes in handy. A visitor is an object that "visits" the AST and does something useful with the information in the AST. The visitor design is an elegant and flexible way to implement such operations on the AST. The different back end applications in the analysis tool are implemented according to the visitor pattern technique. The visitor pattern technique is suitable when it is desirable to perform some operation on every object in a structure or a collection of objects (the AST) and JJTree provides support for this. By invoking the proper option in the grammar file JJTree will generate the visitor design support:

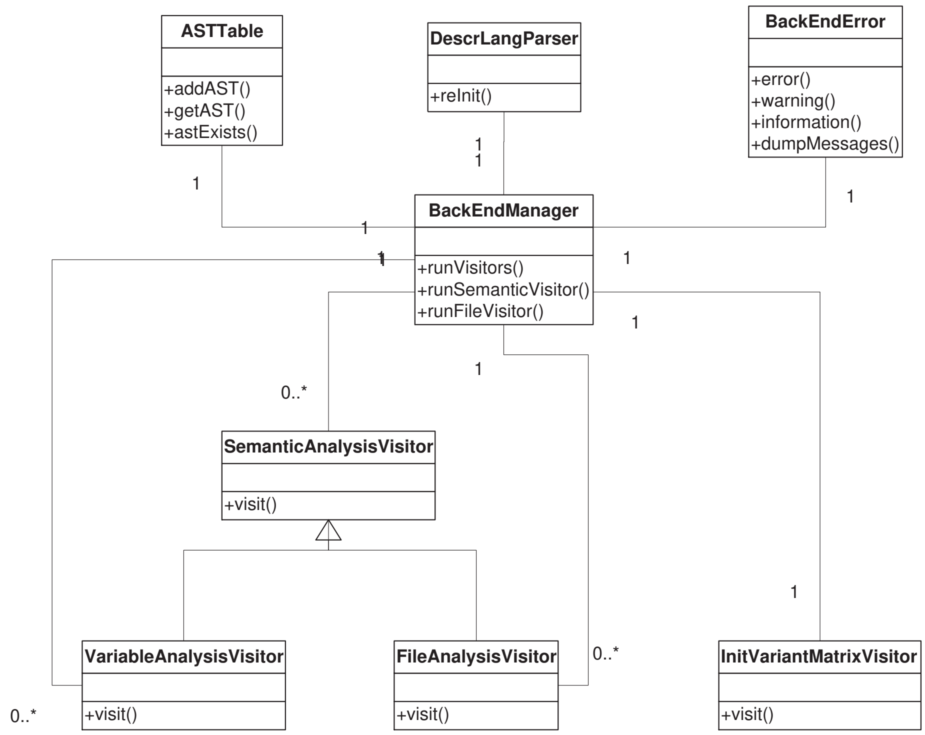

Figure 5.8: Class diagram depicting the back end design

```
Options {
  VISITOR = true;
}
```

When this option is set to true, JJTree will automatically do two things:

1 Insert a `jjtAccept` method into each of the AST node class definitions that it generates.

2 Generate a Visitor interface (a standard kind of Java interface) with an empty method for each type of AST node used in the grammar.

When implementing a back end application (i.e. visitor) it must implement the automatically generated visitor interface and the visitor implemented must also contain a handwritten `visit()` method to deal with each type of AST node that the grammar generates. This is illustrated in the figure 5.9.

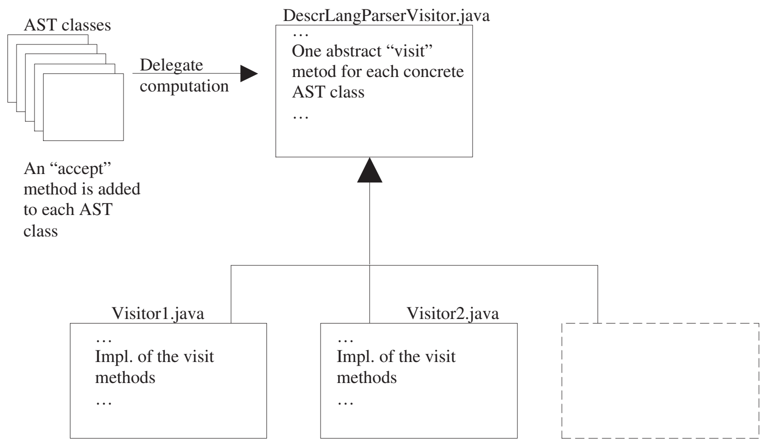

Figure 5.9: Vistor pattern overview

Figure 5.9 illustrates that it is possible to define an arbitrary number of visitors that operate on the elements of an object structure without modifying the classes of the elements on which it operates.

Once the parser has parsed the file provided to it, it passes back a handle to the root node of the created AST, which can be stored using the construct:

```
ASTStart ast = parser.Start();
```

`ast` now holds an instance of the root node of the newly created AST. It is now possible to instantiate the suitable visitor:

```
DescrLangParserVisitor fv = new FooVisitor();
ast.jjtAccept(fv,null);
```

The visitor object is passed to the jjtAccept method of the AST root node. The second argument, null, is not used. The call to jjtAccept is telling the node to "accept" the indicated visitor. The automatically generated jjtAccept method looks like this:

```
/** Accept the visitor. **/
public Object jjtAccept(DescrLangParserVisitor visitor, Object data) {
  return visitor.visit(this, data);
}
```

As can be seen the method just does a call back to the DescrLangParserVisitor itself:

```
visitor.visit(this, data);
```

The method passes to the visitor's visit method a handle to the AST node itself (`this`), and a handle to the Object data, which in the case of the analysis tool always will be **null**. What the jjtAccept() method in ASTStart() effectively does is send a message to the indicated Visitor, saying "here is a handle to me (which will be of type ASTStart), do whatever you are supposed to do with an AST node of my type". The visitor is a whole collection of overloaded visit() methods, one for each AST node type. The particular visit() method in the Visitor that will be invoked is the one whose first argument matches the type of calling node. The way the visitor is instantiated and how it traverses the AST is depicted in the sequence diagram in figure 5.10.

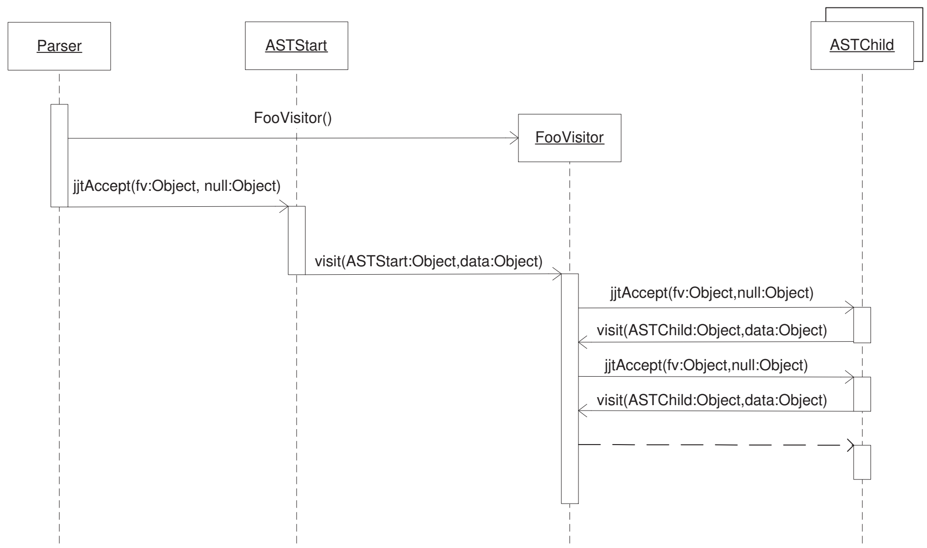

Figure 5.10: Traversing the AST with a visitor

The root node of the AST, ASTStart, which is the obvious starting point invokes its `visit()` method in the visitor by passing a reference to itself. This visit method will iterate over all its children and accept them (depicted in the figure 5.10 as the object ASTChild), forcing their respective `visit()` methods to be invoked. Each node in the AST will recursively traverse its possible children the same way until all nodes have been visited. Here follows an example of a visit method:

```
public Object visit(ASTInclude node, Object data) {
```

```
  Node n;
  String file = "";
  for(int i = 0; i < node.jjtGetNumChildren(); i++) {
    n = node.jjtGetChild(i);
    if(n instanceof ASTFilePath) {
      file = (String)n.jjtAccept(this,null);
    }
    else {
      n.jjtAccept(this,null);
    }
  }

  return data;
}
```

This is the visit method for the node built when an include directive is parsed. The method iterates over its children and when finding a children of type ASTFilePath the node is accepted and its visit method is invoked, implementing logic to resolve the file path to be included and returning it to ASTInclude.

5.6.2 Recursive analysis

It is desired that the analysis tool is able to perform checks on the entire file hierarchy of description files. In order to obtain this functionality all files have to recursively be searched for constructions that include other files (i.e. either via the Sub-module concept or the include directive).

The AST is therefore traversed (with a suitable visitor), starting with the AST build from input file to the parser, in search for include clauses. If a module or target include is found the parser will be re instantiated from inside the visitor with the file to be included in the file tree. The parsers parses the file, builds the AST, and then a new visitor will be instantiated traversing the new AST in the same fashion for inclusions. This procedure will repeat itself until all files have recursively been searched and an AST have been build for all the files in the file hierarchy, identified by the input file.

The behavior is slightly different when it comes to files included via the include directive. These files are considered a logical part of the files from which they are included. Therefore, they are parsed they same way as above (by re instantiating the parser) but their AST is not passed to a new instance of the visitor. Instead the AST is added to the AST being traversed as a child to the node where the include is found.

A visitor resumes traversing its AST once all its leaf files have been traversed and searched for include statements. An example of how the module and target includes are handled is illustrated in the sequence diagram figure 5.11.

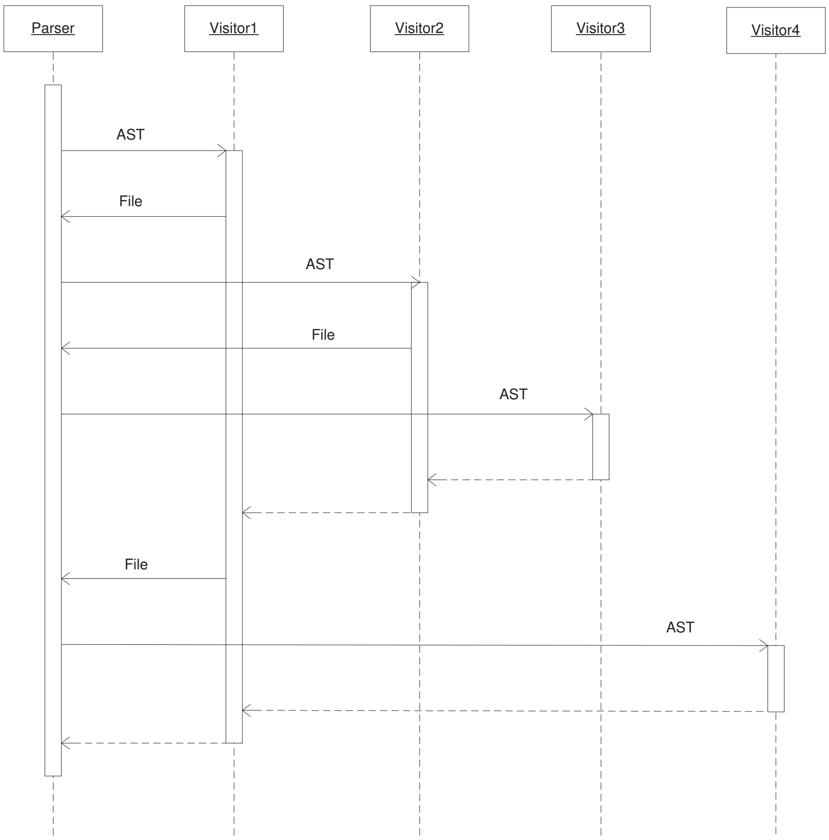

Figure 5.11: Building the file tree

In figure 5.11 *File* is an arbitrary module or target description found in the AST being traversed by the visitor.

It should be mentioned that this version does not constitute any checks regarding circularities in the file tree, meaning checks to find if file includes another file that recursively includes the initial file which will cause an infinite loop of inclusions as illustrated in figure 5.12.

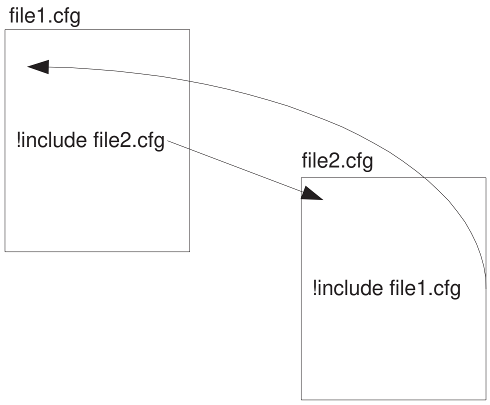

Figure 5.12: Infinite loop of inclusions

5.6.3 Semantic analysis

The semantic analyser is implemented in a visitor named SemanticAnalysisVisitor (`SemanticAnalysisVisitor.java`). This visitor besides implementing checks to find errors violating the language semantic, also implements the recursive analysis to identify the file tree as described in section 5.6.2. Whenever an include file is found the parser is invoked ending with a AST being build which is either added to the existing or left separate.

To be able to ensure that the semantic rules are followed as well as being able to spot other errornous constructions in the files the SemanticAnalysisVisitor have to able to resolve and evaluate the nodes of the AST. This includes evaluating conditionals, resolving file paths, executing predefined functions and storing variables and resolving them when referenced. It is feasible that all visitors incorporated in the analysis tool also can implement this functionality. Therefore all additional visitors can extend SemanticAnalysisVisitor and thereby accessing the functionality without having to implement it themselves. Wherever it is feasible to override the implementation of the SemanticAnalysisVisitor the visitor can simply implement it's own version of the suitable `visit` method for the AST node.

Handling variables

As explained in section 4.2.5 the variable scope is considered to be from the point of definition and further down the file tree. In the file the variable is being defined it is considered over ridable while in a leaf file it is constant. SemanticAnalysisVisitor therefore implements two tables inhabited by the variables, to ensure the variable scope and semantic. One being the `localVariables`, obviously taking care of the local and over ridable variables. On the first encounter of the variable it is put in this table. The other table is called `constantVariables` and all variables in this table are considered constant and may not be overridden. When a module or target include is found the contents of `localVariables` are merged into `constantVariables` and send as an argument to the included module or target ensuring that the variables are visible in the leaf files.

The chosen data structure for the variable tables is hash table with the variable name as the key and the variable value as the value. If the variable is defined without a value it is set to an empty string. Using a hash table ensures easy lookup once the a variable is referenced.

SemanticAnalysisVisitor also implements a third table; `illegalVariables`, mapping to the variables having been blocked by the unset directive. If a variables is unset it is removed from the tables containing legal variables and put in `illegalVariables`.

Whenever a variable is referenced or used, the visitor checks according to the variable semantic if the construction is correct, otherwise an error is issued. Figure 5.13 gives an example on how the variables are handled.

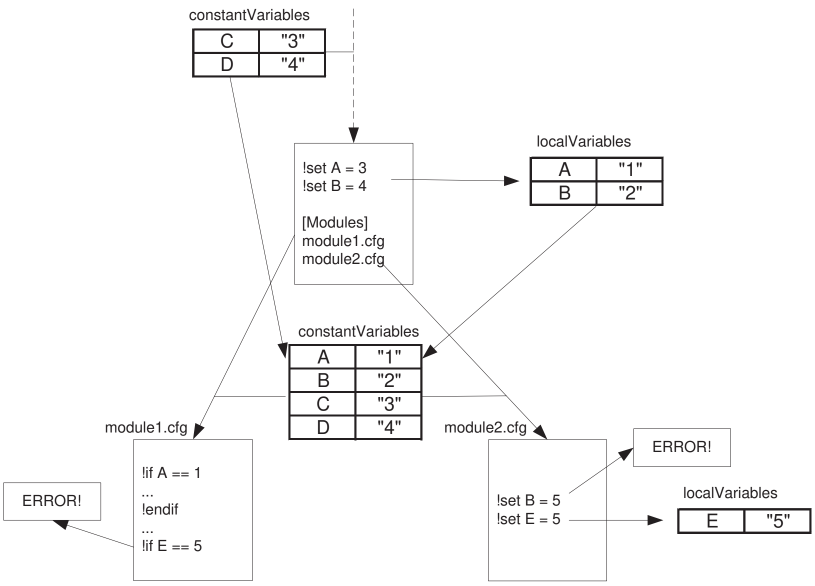

Figure 5.13: Example on how the variables are handled

Figure 5.13 shows how the two variables tables are merged together before they are propagated down the file tree. The figure also shows constructions that produces an error.

Evaluation of conditionals

The evaluation of an if statement is pretty straightforward mapping completely to the semantic. The boolean expression is made up of either comparisons between variables or a predefined function which can be grouped together with the logical operators "&&" or "||" denoting logical AND and OR. The evaluation evaluates the individual comparisons and functions before possibly "ANDing" or "ORing" them together. When eventually all individual comparisons and functions have been logically evaluated the result is returned back to the parent, being either an if or an elseif node. If the conditional evaluates to true the branch will be taken otherwise a possible proceeding conditional will be considered and evaluated.

The evaluation an ifdef or ifndef statement is even simpler. The variable that should be examined with respect to its existence or non existence is looked up in the variable table. If the variable is defined the ifdef statement will evaluate to true and its branch will be taken, and correspondingly vice versa for ifndef. If the variable is not defined the scenario is the opposite.

Predefined function library

The SemanticAnalysisVisitor implements the library of predefined functions available in the language. The functions available as of today are described in table 5.1.

`IsDefined(var)`	Returns true if var is defined, otherwise false.
`FindSubString(string, substring)`	If the given string contains substring the function evaluates to true.
`GetVariableList(expr, separator, prefix)`	The functions returns the list of all variables matching the given expr separated by the given separator
`ThisFileInfo(fileinfo)`	Returns file info for the file being parsed (drive and/or directory and/or filename and/or suffix)
`TrimBS(string)`	Removes trailing backslash from the given string
`SwitchStringVal(var, matchlist)`	Returns a new value to var according to the old value and the match in matchlist

Table 5.1: Predefined function library

By extending the SemanticAnalysisVisitor, the predefined variables become available for additional visitors. It should be mentioned that these functions are only a subset of the functions possible to use in the description files. However they are the most commonly used and its easy to implement additional functions when suitable and desired.

Error reporting

The semantic analysis differentiates between three different types of messages, namely errors warnings and information messages, which are also described in section 5.6.9. The errors are typically such where the variable handling semantic is violated. For example if a variable is referenced before it is defined the semantic analysis will produce the following error message:

```
ERROR: Variable A cannot be resolved, at line 7 in file
C:\dir\test.cfg
```

As can be seen from the above example the error type, description and location is clearly stated. Another example where an error is issued regards the consistency between file list sections and packaging sections. A file listed in a packaging section should always be listed in file list section as well. If any inconsistency is found between these two kinds of sections an error is issued:

```
ERROR: File file1.c is exposed to packer but not listed in any
filelist section, at line 15 in file C:\dir\test.cfg
```

The warning messages are those that identifies constructs that are regarded harmless but should be kept in mind. For example if an option is used in connection with the set directive (remember that the NDL does not apply any variable options but they are still allowed by the grammar) the following warning message will be issued:

```
WARNING: Option to !set directive is obsolete and will be ignored,
at line 3 in file C:\dir\test.cfg
```

The intention of the information messages are to inform the user of constructs that the tool can't handle. The only information message implemented in this version of the tool is when a special make macro ($(AUTO_DIR)) is used in a file path. This macro is not known by the analysis tool and therefore an information message is issued:

```
INFORMATION: The macro $(AUTO_DIR) is set by make and can't be
resolved by this tool, at line 15 in file C:\dir\test.cfg
```

5.6.4 InitVariantMatrix

A variant variable is normally always provided to SDE when parsing the file tree. The variant variable constrains SDE to only consider certain parts of the file tree, that is the parts applicable for the given variant. Together with a target variable, which pinpoints which target that is to be included, the variant forces

SDE to extract information that is needed for the explicitly given variant and target. However not all targets can be build with all variants. Figure 5.14 (this figure does not apply a real case from EMP but is strictly hypothetical) shows how the variant and targets form a matrix, where the gaps illustrates that this variant target combination is not applicable.

Variant/Target	Target1	Target2	Target3	Target4	Target5
Variant1	x	x		x	
Variant2	x	x	x	x	x
Variant3	x	x	x		x
Variant4	x	x	x	x	x
Variant5	x	x			
Variant6	x	x		x	x
Variant7	x	x	x	x	x
Variant8	x		x	x	x
Variant9	x	x	x		x
Variant10	x	x	x	x	x

Figure 5.14: Variant target matrix

The analysis tool offers the user to state the variant and target, via a command line option, which is to be analysed. The tool will set the variables variant and target (which are to be considered as constant) to the values entered on the command line. The variant or target on the command line can be one ore many. Optionally the keyword *all* can be stated for the variant and/or target indicating for the tool that all variants and/or targets should be analysed. However as seen in figure 5.14 not all targets are applicable for all variants. Therefore the analysis tool first (if the all keyword is stated for either the variant or target) extracts the correct variant target matrix by first extracting all variants from the variants section and then for each variant resolve the applicable targets (the target inclusion is always dependent on the variant). This is done by the InitVariantMatrixVisitor (`InitVariantMatrixVisitor.java`) which is executed before any other visitors are executed.

Further visitors iterates over the matrix examining one variant target combination for each iteration until eventually all variant target combinations have been examined.

5.6.5 FileAnalysisVisitor

FileAnalysisVisitor, residing in the file `FileAnalysisVisitor.java`, takes care of analysing the description files with respect to to the files listed in the sections listing files (i.e. file list sections and packaging sections and their respective modification section). The visitor inherits from SemanticAnalysisVisitor and thereby accessing all the functionality provided by that visitor. FileAnalysisVisitor overrides SemanticAnalysisVisitor's visit method for ASTFilePath, to implement its own version. Every time the visit method for ASTFilepath is visited, FileAnalysisVisitor checks if the current section is any of the ones mentioned above. If so the file path is saved together with the information regarding the section it resides in. When all description files AST representation have been traversed, possibly several times if more than one variant target combination is to be examined, the file analysis result (the file analysis is preformed by the class FileAnalysis) is written to a file. The file is a simple text file and the information it contains is organized with respect to the file name. For each file name every section it has been found in is listed and for each section every variant target combination it adheres to is stated. Here follows a portion of how a example analysis result file looks like:

```
...
fileA.c     c:\dir\
[SourceFiles]                  Variant1:Target1,Target2
                               Variant2:Target2,Target3
[Packaging_ExposedSorceFiles]  Variant1:Target1
                               Variant2:Target3

fileA.h c:\dir\
[IncludeFiles]                 Variant3:Target1,Target2
...
```

The result file lists every file found in the description files. For each file its full path is stated to the right. Then for each section the file is found in, the variant target combinations are presented on the right. The file analysis can be performed independently of the semantics of the language and hence possible to perform on current implementations of the description files.

5.6.6 VariableAnalysisVisitor

VariableAnalysisVisitor, residing in the file `VariableAnalysisVisitor.java`, takes care of analysing the files with respect to variables defined in them. Every variable is saved with information regarding its variable name and value and the location in the file tree. Whenever the variable is overridden or referenced information regarding its possible new value and the location is saved. The data structures that holds the saved variable information are then analysed (by the class VariableAnalysis) and the result is written to a text file:

Here follows a portion of how a example analysis result file looks like:

```
...
------------------------------------------------------
```

```
VARIABLE: A
module.cfg, line: 52      Value: 1
module.cfg, line: 102     Value: 2

REFERENCED
module.cfg, line: 66      Value: 1
module.cfg, line: 152     Value: 2

------------------------------------------------------
...
```

The result file lists the variable identifier (A) followed by the file and line number where it is defined and possibly overridden, together with the value (1 and 2). After that follows a list of every location the variable is referenced together with the value it is expanded to. The variable analysis is performed with the new semantics for variables and hence can not be performed with expected result on current implementations of the language.

5.6.7 BackEndManager

The visitors are not instantiated directly from the parser. Instead they are invoked from a class named BackEndManager (`BackEndManager.java`) which acts as an interface between the parser and the visitors. When the parser is done parsing the input file provided to it, it instantiates the BackEndManager and passes the build AST to it together with information regarding how the file is to be analysed (provided as command line options when running the analysis tool executable). BackEndManager then instantiates the suitable visitors. Whenever a new file needs to be parsed the parser is invoked from this class and then subsequently a new visitor instance is created and the newly created AST is passed to it. Hence BackEndManager works as a hub in the back end, instantiating new vistors and invoking the parser when needed.

5.6.8 ASTTable

As the files sometimes are analysed several times and it would be highly unnecessary to parse the file again every time, their AST representation is saved the first time they are parsed. ASTTable (`ASTTable.java`) implements a hash table where the ASTs are put together with the file as key depicted in figure 5.15.

Every time a file is to be analysed and hence possibly parsed, ASTTable is first checked if the AST already have been build for the given file. If so the AST can be invoked directly preventing the file to be parsed a second time. The individual ASTs in the table can have possible references to another element in the table, containing the AST for the referenced file as illustrated in figure 5.15.

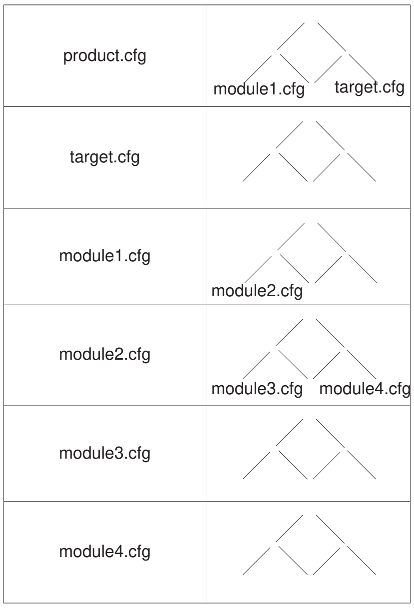

Figure 5.15: ASTTable

5.6.9 BackEndError

This class is responsible for gathering and reporting the errors the visitors find and output them to the user. The analysis tool differentiates between three types of error output; errors, warnings and information.

- Errors are faults that it is imperative to fix to guarantee a correct result once the file is used.
- Warnings are such that are not vital to fix to ensure correct behavior but still considered recommendable to correct.
- Information's aims to inform the user of constructions that the analysis tool can't handle. Examples of such are when "$(AUTO_DIR)" is used in file paths. This macro is set by make and can't be handled by the analysis tool.

5.7 Summary

The analysis tool introduces a whole new way of parsing and processing the description files. The files are parsed with compiler compiler tool generated parser as opposed to the handwritten parser that exists today for parsing the current implementation of the description language. The analysis tool consists of three major parts, namely:

- Preprocessor
- Front End
- Back End

The preprocessor is responsible for parsing and evaluating the macro directive (!define). The front end implements the actual lexer, parser and tree building (generating the AST). The back end implements the semantic analysis but also a file analysis as well as a variable analysis. The back end takes the AST generated by the front end and traverses it using the visitor design pattern.

Chapter 6

Conclusion

The objective of this master thesis has been to, based on the current implementation of a language (CDL) propose a "new" language (NDL) that is to as much extent as possible compatible with the existing one. The requirement of backwards compatibility implies that developing the new description language is not concerned with specifying a new and better concept for the application of the language. Instead the focus of this thesis has been to analyse the current syntax and to transfer and fit that syntax into a formal definition using a formal notation such as BNF. Hence the the language is not "new" in the sence that it provides a whole new concept for the description language and how it is applied. The language is only new in the sence that the syntax of the language is constrained into a formal definition. In addition to having a formal grammar, NDL utilizes a completely different way of parsing and processing the language.

Setion 6.1 highlights the accomplishments of the thesis. The main differences between the languages, if its feasible to talk about to different languages, are presented in section 6.2 as well as a discussion conserning these. Section 6.3 lists a number of matters that should be considered if the work this thesis has provided shall prevail. This includes a list of further work and the integration of NDL into the environment.

6.1 Status

To sum up, this master thesis has fulfilled the major objectives imposed on the project, namely;

1. Briefly understand and describe CDL in terms of its syntax, semantic and application. This however does not constitute a complete description of the language.

2. CDL is constrained as a result of defining a formal grammar resulting in a new language (NDL). Where it has been considered feasible the language semantic have been slightly revised.
3. An analysis tool is developed capable of performing syntactical and semantical checks of NDL as well as analysis of the description files.

Hopefully the design of the analysis tool is such that it is easily extendable with new functionality, such as new syntactic and semantic checks or functionality that as of today is done by SDE such as generating the makefile or extracting information needed by CME.

6.2 CDL vs. NDL

This sections elaborates on the main differences between CDL and NDL which regard three different aspects; 1) syntax, 2) semantic and 3) how the language is processed and parsed. The differences are highlighted together with a discussion regarding the right and wrong of the two languages.

6.2.1 Syntax

The syntaxes of the current and new language are practically identical, although some minor differences exist, mainly because the current language offers higher degrees of freedom concerning the possible ways of writing the different statements and expressions of the language whereas NDL does not. When specifying the grammar for NDL the current description files have been examined and formed a base when defining the formal grammar. Therefore NDL is to great extent identical to the syntax of CDL.

An important property of CDL is that it is easily extandable with new sections without having to intrude to much on the tools using the language. This is because CDL offers great flexibility when writing the descriptions in the language mainly because there is no formal definition of language. This implies that it is easy to write descriptions "correct" meaning that the CDL parser does not complain on errornous constructs.

The parser of NDL will however pinpoint slightest error in the description file as a result of having a formal grammar constraining the language to well-defined rules.

There is point of having the language very general, as in CDL, and allowing a high degree of freedom when writing descriptions. But it also makes the arise of errors very easy. It is therefore considered feasible that using a more constrained language, as in NDL, is the recommended choice, although extending the language with new constructs might be messier.

If given more free hands when defining the syntax a more constrained way of defining the beginning and end of an expressions is feasible. For example all strings in the language that should be considered literally can be encapsulated in quotes. For example the variable values:

```
!set A = "this is literal string"
```

This would greatly simplify the parsing of the language since it is easier to know when the string actually begins and ends. Also the end of a section could be stated with a sufficient delimiter denoting a clear end of the section syntax.

6.2.2 Semantic

The only semantic difference between CDL and NDL is the variable handling. Whereas CDL uses global variables, NDL utilizes a confined variable scope. The current usage of global variables must be considered a unsatisfying implementation. One of the initial ideas was to make the modules independent with well-defined interfaces between those. This is not entirely the case today as a result of the extensive use of global variables. The variable handling used by NDL does not solve the problem with dependent modules but imposes a more constrained and more intuitive variable handling.

6.2.3 Language processing

Whereas the syntax and semantic of the languages does not differ that much, the way the language is parsed and processed does. Whereas the CDL parser is handwritten the NDL one is generated from a compiler compiler. Whereas CDL description files are parsed multiple times to extract the vital information, the NDL parser parses the files a single time generating an AST that instead can be traversed a number of times. Whereas CDL parser hardly implements any error checks, the NDL parser implements both lexical and syntactical checks as well semantical. This very useful because it implies an earlier discovery of errornous constructions in the development process which might for example include such constructs that produce errors when delivery packing the software such as inconsitency between file list sections and packaging sections (see section 5.5.2).

Having a handwritten parser imposes a bunch of implications (such as maintenance and consistency issues). A generated parses from a formal grammar removes these implications and provides a safer way of parsing the files. An AST representation is also considered more feasible for extracting information and is a more common way when processing and analysing a language.

6.3 Further work

The outcome of this thesis is a prototype analysis tool and its important to stress that it is far from complete. The grammar the tool incorporates has to be carefully examined and the tool has to be extensively tested. This section highlights the status of the tool as of today and also a discussion regarding the integration and migration aspects if the tool is to be used in the environment.

There is number of items that as of today has not been resolved, both in the grammar and in the analysis tool, motivated by lack of time. These unresolved items are listed below.

- **Incomplete grammar:** The grammar of NDL is incomplete meaning that a definition has not been found for the full language. Some aspects of the language have been left out on purpose, for example make specific parts, but some parts simply haven't been formally defined because of their sheer complexity. An example of such a part is a section where it is possible to use patterns using regular expressions to define file paths.
- **Indented sections:** The CDL parser does not allow sections to be indented. If so they are simply disregarded. The analysis tool does not incorporate such a check because the handling of white spaces and tabular are ignored by the lexer and thus the parser has no knowledge of them ever existing. There are workarounds but then the whole parser has to be redesigned.
- **Circular dependencies:** The analysis tool does not implement checks that can find circular dependencies, meaning that files should not be able to include each other in such a way that an infinite loop of recursive inclusions occur.
- **Negation of boolean expressions:** In CDL it is possible to negate the boolean expressions by using perl syntax. This is not implemented in NDL yet.
- **Additional boolean comparisons:** By using perl syntax in conditional statements it is possible to make additional comparisons between variables beyond the equal and not equal clauses, such as greater or less than. This is however not commonly used since the variables are strings and one can argue whether or not this feature should be inherited into NDL given that the language does not allow typed variables.
- **Predefined functions not complete:** The predefined functions library does not implement all possible functions available in CDL. Also since the perl availability is abandoned some functionality provided by using perl is lost. It is however easy to invoke new functions into the language to fill the gaps.

- **Environment variables:** In CDL it is possible to access environment variables. As of today this feature is not applicable in NDL.
- **Predefined variables:** SDE implement a number of predefined variables that accessible in the language. Most of these predefined variables are not incorporated into NDL.
- **More sophisticated analysis:** The variable and file analysis done by the analysis tool are very simple. It is therefore feasible to provide them with more sophisticated behaviour both in terms of how the result is presented as well the result it self. This could include for example to generate the result into HTML-format with hyper links to increase the readability of the result and to provide a deeper variable analysis that could analyse the many variable dependencies.
- **Narrow-minded view of the language:** This thesis has only focused on the in-house applications of the language, meaning how the language is applied by EMP in the development process. No focus at all have been taken to the customer application of the language (see the customer portion of figure 2.3) which might impose other requirements on the language.

The majority of these matters can be solved fairly easy while other has to carefully be considered before implementing a solution.

6.3.1 Migration and integration

Since CDL and NDL are not entirely identical in there syntax and semantic the analysis tool can not be invoked directly to the existing description files. In other words; NDL is not backwards compatible with CDL. Before the tool can be integrated with the environment the files has to be run through the tool and thereby migrating the current implementations to the new language by correcting the errors pinpointed by the lexer, parser and semantic analysis. After all errors pinpointed by the tool have been corrected, of course carefully considering whether or not the error detected by the tool is to be considered as an actual error, it is important to ensure that the files adhering to NDL provide the same functionality as the old ones.

Appendix A

Terminology and Abbreviations

AOL
Application Oriented Languages
API
Application Programming Interface
AST
Abstract Syntax Tree
BE
Back End
BNF
Bachus Naur Form
CC
Compiler Compiler
CDL
Current Description Language
CM
Configuration Management
CME
Configuration Management Environment, a plug in to the CM tool ClearCase used at Ericsson
CFG
Context Free Grammar
Description language
The language this thesis investigates
Description file
Text file implementing the description language

EBNF
Extended Bachus Naur Form
EMP
Ericsson Mobile Platforms
NDL
New Description Language
FE
Front End
GUI
Graphical User Interface
GPL
General Purpose Languages
NFA
Nondeterministic Finite Automaton
PPZ
Script responsible for packing software
SDE
Software Development Environment
SDK
Software Development Kit
Target
A compiler for a specific processor
Variant
A well-defined portion of the platform software

Appendix B

Domain Specific Languages

As mentioned in the introduction the description language can be considered as a DSL [5] [6], given that has a small vocabulary and is highly dedicated to a particular domain, namely the EMP software development environment. The term Application Oriented Languages or Little Languages can also be used.

The major benefits of using a DSL are the following:

- **Easier to learn and program**, Because of appropriate abstractions and declarative formulations, a DSL program is concise and readable than its GPL counterpart. Hence it also easier to learn and easier to program and development time is shortened and maintenance is improved.
- **Protected interfaces**, DSLs provide focused, protected interfaces toward a systems functionality. This allows security and integrity to be maintained during system configuration and reconfiguration. Only the parts a given user should have access to need to be made available.
- **Programming in terms of the application**, There is always a gap between the formulation of a problem to be solved and the formulation of the solution. The larger the difference is between these two, the harder it is to do what you want. The DSL concept lets languages be adapted more to the user than the computer.

The characteristics of a DSL are:

- specialized notation
- narrow domain of application
- implementation efficiency is usually secondary

- easier to optimize than a GPL
- enables the user to describe the problem in terms of the application

Appendix C

Compiler Theory

C.1 Introduction

This chapter gives a brief overview of the fundamentals behind the construction of a compiler and the formalism and notation used when defining the language grammar, preparing the reader for the chapters that describe the actual grammar and semantics of the language (see chapter 4) as well as the design of the analysis tool (see chapter 5). Also the compiler compiler tool of choice is introduced, briefly describing how it works.

C.2 Compiler theory

The typical phases of a compiler are depicted in figure C.1.

The *lexical analysis* (lexer) reads in a sequence of characters (the input description file) and produces a sequence of tokens. The rules used to break the sequence of characters into a sequence of tokens are supplied as the set of regular expressions defined by the grammar. These tokens are then passed to the *syntactic analysis* (parser) for further processing. The syntactic analyzer or the parser uses a set of grammar productions to decide whether or not the sequences of characters follow the rules described by the grammar. The parser produces a tree representation (AST) of the file being parsed, which is provided to the *semantic analysis*. The semantic analysis contains functionality for traversing the AST and to produce error output and/or analysis results.

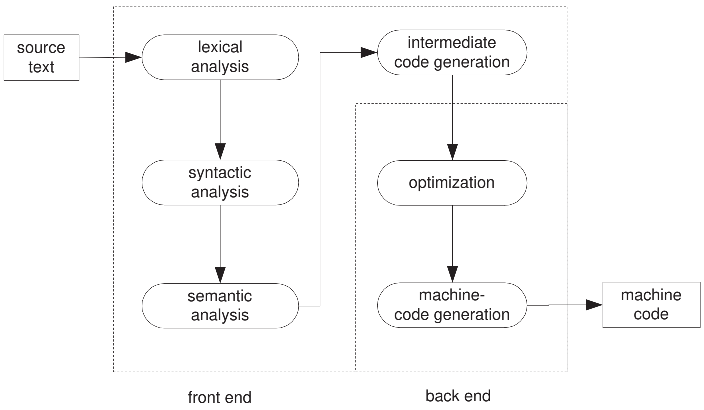

Figure C.1: Typical phases in a compiler

C.3 Formalism and notation

When defining the NDL the notion of context free grammar (CFG) is used, from which it is possible to produce an abstract syntax tree (AST) representing the defined grammar. The construction of the AST involves the steps depicted in figure C.2.

The construction involves two isolated steps, one being the *lexical analyser* (lexer) and the other being the *syntactical analyser* (parser). The lexer takes a stream of characters, representing the input description file, and produces a stream of symbol or tokens, discarding white space and comments between the tokens.

The parser is responsible for parsing the produced tokens, following the rules dictated by the context free grammar. The context free grammar describes how the nodes may combined to form a syntax tree. A syntax tree consists of two kind of nodes, namely

- *terminal nodes* - instance of a token, always leaf in the tree.
- *nonterminal nodes* - other nodes in the tree, usually inner nodes.

A context free grammar G can be defined as the tuple $G = (N, T, P, S)$ where

N - a finite set of nonterminal symbols.

T - a finite set terminal symbols (tokens).

P - a finite set of productions that describes a legal sequence of childs to nonterminal.

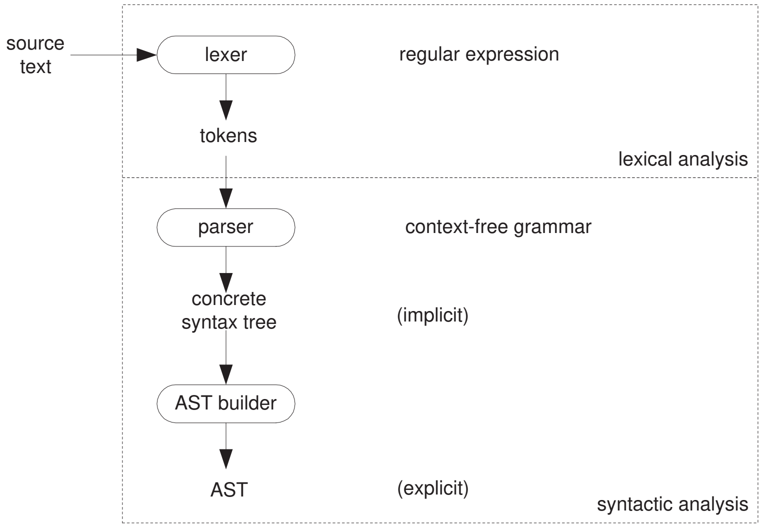

Figure C.2: The steps involved in constructing the AST

S - the start symbol (one of the nonterminals in N).

The terminal symbols or tokens can be defined as *regular expressions*. A CFG could be used but it would be overkill. CFG's have the expressive power to describe recursive structures but tokens do not have any internal recursive structure and iterations is sufficient. Regular expressions is therefore sufficient for describing tokens.

Figure C.2 depicts an intermediate step producing a *concrete syntax tree* before finally building the AST. The concrete syntax tree has all tokens present in the tree, whereas in the AST redundant information is removed and only nodes and tokens that are important are kept.

The notation used for the description language CFG is Extended Backus Naur Form (EBNF). EBNF is, as the name implies, an extension of Backus Naur Form (BNF). A BNF is a shorthand for writing several productions for the same nonterminal in the same rule. EBNF is BNF extended with the notation of regular expressions that can be written in the right hand side of the rule. Using EBNF gives a very compact grammar and is the standard notation for describing the syntax of programming languages. The BNF notation can be expressed as follows:

$$X \rightarrow \gamma \mid \delta$$

where X is a nonterminal and γ and δ are arbitrary sequences of terminal and nonterminal symbols. The terminals and nonterminals on the right hand side of the rule can be written using regular expressions. The notation described in table C.1 is used for the regular expressions.

a	An ordinary character stands for itself.
ϵ	The empty string.
$M \mid N$	Alternation choosing from M or N.
$M \cdot N$	Concatenation, an M followed by an N.
MN	Another way to writ concatenation.
$M*$	Repetition, zero or more times.
$M+$	Repetition, one or more times.
$M?$	Optional, zero or one occurrence of M
$[a - zA - Z]$	Character set alternation.
.	A period stands for any single chapter except new line.
$"a. + *"$	Quotation, a string in quotes stands for itself literally.

Table C.1: Regular expression notation

Now we have a tool for describing the description language formally as a CFG using the EBNF notation. The grammar is defined in section 4.3.

Appendix D

JavaCC

D.1 Introduction

A compiler compiler, as the name implies, is a tool that support the development of a compiler. Such a tool typically takes a formal grammar and generates the parser and possibly also the syntax tree. This simplifies the development process significantly and reduces many risk that having a handwritten parser produce.

JavaCC [2] is a compiler generator that accepts language specifications in BNF-like format as input. The generated parser contains the core components of corresponding compiler of the specified language, which includes a lexical analyzer and a syntactical analyzer. JavaCC also provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included in the distribution of JavaCC), actions, debugging, etc. JJTree act as tree building preprocessor to JavaCC and has been used when building the AST. Figure D.1 shows the overall structure of a parser generated by JavaCC and JJTree.

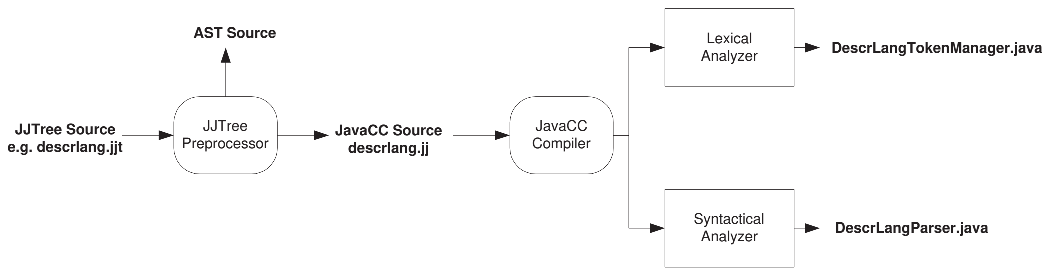

Figure D.1: Generation of JavaCC parser

The JJTree preprocessor takes the language specification (*.jjt) extended with tree building constructs and generates the AST and a JavaCC grammar file (*.jj). JavaCC compiles the generated grammar file and generates the syntactical analyzer (parser) and lexical analyzer (lexer). Note that JavaCC could also be used without JJTree. In that case the handwritten grammar specification is provided directly to JavaCC.

The JavaCC source file includes both the lexer as well as the grammar specification in the same file. The grammar file has the following outline:

options
”PARSER_BEGIN” ”(” ¡IDENTIFIER¿ ”)”
java_compilation_unit
”PARSER_END” ”(” ¡IDENTIFIER¿ ”)”
(*java_code_production* | *regexp_production* | *bnf_production* | *token_manager_decls*)*
¡EOF¿

The grammar file starts with a list of options (which is optional). They are followed by a Java compilation unit enclosed between ”PARSER_BEGIN(name)” and ”PARSER_END(name)”, where name is the name given to the generated parser. The contents between the PARSER_BEGIN and PARSER_END keywords is a regular Java compilation unit (a compilation unit in Java lingo is the entire contents of a Java file). This may be any arbitrary Java compilation unit so long as it contains a class declaration whose name is the same as the name of the generated parser. Hence, in general, this part of the grammar file looks like this:

```
PARSER_BEGIN(parser_name)
. . .
class parser_name . . . {
  . . .
}
. . .
PARSER_END(parser_name)
```

Typically the compilation unit is responsible for instantiating the parser and lexer with the file to be parsed. After this follows a list of productions that can either be a *regexp_production* defining the lexer, a *bnf_production* defining the parser, a *java_code_production* or a *token_manager_decls*. The java code production is way to write Java code for some productions instead of the usual EBNF expansion. This is useful when there is a need to recognize something that is not context-free or for whatever reason is very difficult to write a grammar for. The token manager declarations are declarations and statements accessible from within lexer.

D.2 Lexical Analysis

The lexical analyzer in JavaCC is called *TokenManager*. The TokenManager is used to group characters from an input stream into tokens or symbols according to specific rules, which in the case of the analysis tool is according to the regular expressions defined for the nonterminals in chapter 4. Each specified rule in TokenManager is associated with an expression kind:

SKIP:	Throws away the matched expression. Suitable to use with white space and comments.
MORE:	Continue taking the next matched expression to build up a longer expression.
TOKEN:	Creates a token using the matched expression and sends it to the parser.
SPECIAL_TOKEN:	Creates a token with the matched expression and optionally sends it to the parser, which is different from **TOKEN**.

Table D.1: Expression kinds of JavaCC tokens

A token, associated with one of the given expression kinds, is expressed with an identifier followed by a regular expression describing how the token should be matched. The notation for the regular expressions is basically the same as the one defined in table C.1. A token to match an identifier, which should start with a letter followed by an arbitrary number of letters and digits can be expressed as follows:

```
TOKEN :
{
    < IDENTIFIER: <LETTER> (<LETTER> | <DIGIT>)* >
  | < #LETTER: ["A"-"Z"] >
  | < #DIGIT: ["0"-"9"] >
}
```

The TokenManager is a state machine that moves between different lexical states to classify tokens. Each lexical state contains an ordered list of regular expressions corresponding to one of the types described in table D.1; the order is derived from the order of occurrence in the input file. All regular expressions that occur in the grammar are considered to be in the *DEFAULT* lexical state, if not explicitly stated differently. The *DEFAULT* state is the standard state and the one the TokenManager per default starts off in when initialized. It's the optional to force the TokenManager to move to another state depending on the token being matched. The feature of having lexical state has proved to be very sufficient when constructing the analysis tool lexer.

A token is matched as follows: all regular expressions in the current lexical state are considered as potential match candidates. The TokenManager consumes the maximum number of characters from the input stream possible that match one of the regular expressions. That is, the TokenManager prefers the longest possible match. If there are multiple longest matches (of the same length), the regular expression that is matched is the one with the earliest order of occurrence in the grammar file. Hence regular expression in the beginning of the grammar file has higher precedence than a regular expression stated further down. For the purpose of matching tokens, the TokenManager applies a nondeterministic finite automaton (NFA).

The TokenManager is in exactly on state at any moment. At this moment, the TokenManager only considers the regular expressions defined in this state for matching purposes. After a match, one can specify an action to be executed as well a new lexical state to move to. If a new lexical state is not specified, the TokenManager remains in the same state.

To illustrate the the way the TokenManager handles lexical states the following example is provided [3]. Figure D.2 depicts a state machine of the example lexical analyzer.

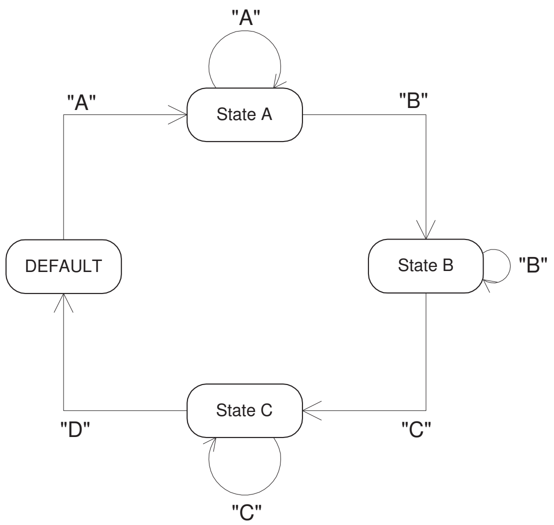

Figure D.2: State machine features in lexical analyzer.

When the analyzer starts it is in the *DEFAULT* state which waits for input. If the input is a character "A", it moves to state *A*. Then from state *A*, if the input is the character "A", it remains in the same state. However, when the input is character ”B” or ”C”, the system moves to the corresponding states. If the machine is facing an unspecified situation, such as hitting a character ”E” in state *C*, it generates a lexical error. The following code segment implements a portion of the *TokenManager* for the example lexical analyzer in JavaCC:

```
TOKEN :
{
  < A: "A" > : State_A    //Switch to State A if input character is "A"
}

<State_A>
TOKEN :
{
  < A_Again: "A" >        //Stay in State A if input character is "A"
  < B: "B" > : State_B    //Switch to State B if input character is "B"
}

<State_B>
TOKEN :
{
  < B_Again: "B" >        //Stay in State B if input character is "B"
  < C: "C" > : State_C    //Switch to State C if input character is "C"
}

<State_C>
TOKEN :
{
  < C_Again: "C" >        //Stay in State C if input character is "C"
  < D: "D" > : DEFAULT    //Switch to DEFAULT if input character is "D"
}
```

Besides the generated TokenManager, JavaCC generates a couple of other files that act as help classes to the lexical analyzer:

Token.java	is a class representing tokens. Each token object has an integer field kind that represents the kind of token and a String field image, which represents the sequence of characters from the input file that the token represents.
SimpleCharStream.java	is an adapter class that delivers characters to the lexical analyser.
TokenMgrError.java	is a simple error class. It is used for errors detected by the lexical analyser and is a subclass of `Throwable`.

D.3 Syntactical Analysis

The syntactic analyzer is a top-down (recursive-descent) LL(k) parser as opposed to bottom-up parsers generated by YACC-like tools [4]. This type of parser uses k number of lookahead tokens to generate a set of mutually exclusive productions, which recognize the language being parsed by the parser. By default, JavaCC syntactical analyzer sets k to 1, but it is possible to override the number of lookahead tokens to any arbitrary number to match productions correctly. A recursive descent parser allows the use of more general grammars and have a bunch of other advantages such as being easier to debug, having the ability to parse to any non-terminal in the grammar, and also having the ability to pass values (attributes) both up and down the parse tree during parsing.

LL(k) parsers allow only *right recursion* in the BNF production rule. Consider the trivial example of adding an arbitrary number or integers together. This can be expressed with the following BNF:

$$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{I}$$
$$\mathrm{E} \rightarrow \mathrm{I}$$

where I denotes an integer value 0-9. The recursive call to E as the first right-hand-side symbol in an E-production is called *left recursion* and grammars with left recursion can not be LL(1). The syntax must be reconstructed so that the parser can recognize the production correctly with limited amount of lookahead tokens. Therefore, sequences of tokens that generate mutually exclusive situations in the production should be placed in the beginning of each possible case, and thereby providing right recursive productions to the parser:

$$\mathrm{E} \rightarrow \mathrm{I}\ (\mathrm{E'})+$$
$$\mathrm{E'} \rightarrow + \mathrm{I}$$

Fortunately the description language grammar defined in chapter 3 has been defined so that it is entirely free from left recursion from the start and no rewriting of the production rules have been necessary. The approach of right recursion is however not always trivial to implement and the conversion should be considered with care.

Using the JavaCC notation for the example expressed above would result in the following grammar code:

```
void E() :
{}
{
  <I> (E'())+
}

void E'() :
{}
{
  "+" <I>
}
```

Each non-terminal mapping to the left hand side (corresponding to *E* and *E'* in the example) of a BNF production rule is written exactly like a declared Java method. Since each non-terminal is translated into a method in the generated parser, this style of writing the non-terminal makes this association obvious. The name of the non-terminal is the name of the method, and the parameters and return value declared are the means to pass values up and down the parse tree. The non-terminals on the right hand side (*E'()* in the example) are written as method calls, so the passing of values up and down the tree are done using exactly the same paradigm as method call and return. The right hand side of a BNF production can also be a terminal corresponding to a regular expression. This is denoted in JavaCC as either a quoted string (”+” in the example) or a reference to token defined in the lexical analyzer (¡I¿ in the example). Basically JavaCC uses the same notation for regular expressions as described in table C.1.

Overriding the default lookahead of 1 can be expressed by explicitly telling the parser to lookahead for more tokens with the use of the LOOKAHEAD(k) keyword:

```
void A() :
{}
{
  (LOOKAHEAD(2) A() | B())
}
```

Besides the generated parser, JavaCC generates a couple of other files that act as help classes to the syntactical analyzer:

***Constants.java**	is an interface that defines a number of classes used in both the lexical analyser and the parser. "*" denotes name given to the generated parser.
ParseException.java	is simple error class. It is used for errors detected by the parser and is a subclass of exception and hence of Throwable

D.4 AST

The tool JJTree, which is a part of the JavaCC distribution, provides tree building capabilities to the generated parser. JJTree is basically a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source code. The output of JJTree is run through JavaCC to create the parser.

By default JJTree generates code to construct parse tree nodes for each nonterminal in the language. This behaviour can bee modified so that only nonterminals of specific interest generate AST nodes. The AST nodes are organized in a tree structure to form the AST. JJTree will generate a Java class for every node defined in the grammar. The nodes are stored in files named AST*.java, where the "*" is the name of the node specified in the grammar. Once these files are generated, JJTree will not regenerate them. Hence these files these files can implement handwritten, besides the code generated by JJTree, methods that enables storing and retrieving data from the node.

JJTree defines a Java interface `Node` that all AST nodes must implement (via `SimpleNode`). The interface provides methods for operations such as setting the parent of a node, and for adding children and retrieving them depicted in figure D.3.

The methods implemented by `Node` provides the following functionality:

```
public interface Node {
  /** This method is called after the node has been made the current
  node.  It indicates that child nodes can now be added to it. */
  public void jjtOpen();

  /** This method is called after all the child nodes have been
  added. */
  public void jjtClose();

  /** This pair of methods are used to inform the node of its
  parent. */
  public void jjtSetParent(Node n);
```

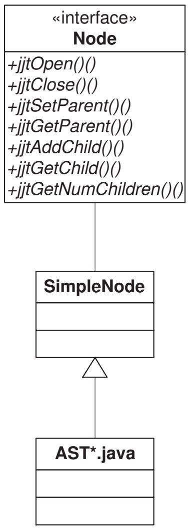

Figure D.3: The Node interface

```
  public Node jjtGetParent();

  /** This method tells the node to add its argument to the node's
  list of children.  */
  public void jjtAddChild(Node n, int i);

  /** This method returns a child node.  The children are numbered
  from zero, left to right. */
  public Node jjtGetChild(int i);

  /** Return the number of children the node has. */
  int jjtGetNumChildren();
}
```

The class SimpleNode implements the Node interface, and is automatically generated by JJTree if it does not already exist. This class can be used as a template or superclass for the node implementations, or can be modified to suit.

Although JavaCC is a top-down parser, JJTree constructs the parse tree from the bottom up. To do this it uses a stack where it pushes nodes after they have been created. When it finds a parent for them, it pops the children from the stack and adds them to the parent, and finally pushes the new parent node itself.

JJTree provides decorations for two basic varieties of nodes, and some syntactic shorthand to make their use convenient:

1. A definite node is constructed with a specific number of children. That may many nodes are popped from the stack and made the children of the new node, which is then pushed on to the stack itself. A definite node is notated like this:

   `#ADefiniteNode(INTEGER EXPRESSION)`

   A definite node descriptor expression can be any integer expression.

2. A conditional node is constructed with all of the children that were pushed on the stack within its node scope if and only if its condition evaluates to true. If it evaluates to false, the node is not constructed, and all of the children remain on the node stack. A conditional node is notated like this:

   `#ConditionalNode(BOOLEAN EXPRESSION)`

   A conditional node descriptor expression can be any boolean expression. There are two common shorthands for conditional nodes:

   a. Indefinite nodes

      `#IndefiniteNode` is short for `#IndefiniteNode(true))`

   b. Greater-than nodes

      `#GTNode(>1)` is short for `#GTNode(jjtree.arity() > 1)`

D.5 Summary

This chapter has introduced the typical ideas behind a compiler. The same ideas has been used to great extent when designing the analysis tool in chapter 5. Also a notation for formally defining a language grammar, EBNF, has been provided. The compiler compiler of choice when implementing the analysis tool is JavaCC. JavaCC is a java based recursive descent compiler compiler available as open source. JavaCC is capable of generating the lexer as well as the parser and optionally also a AST representation of the file being parsed.

Appendix E

New Description Language BNF

This appendix contains the formal definition of NDL using BNF notation.

Stmt → Directive | section | NL

Directive → If | Ifdef | Ifndef | Set | Unset | Include | Options

If → "!if" BExp NL Directive ("!elseif" BExp NL Directive)* ("!else" NL Directive)? "!endif" NL

Ifdef → "!ifdef" ID NL Directive ("!else NL Directive)? "!endif" NL

Ifndef → "!ifndef" ID NL Directive ("!else NL Directive)? "!endif" NL

BExp → OrExp

OrExp → AndExp ("——" AndExp)*

AndExp → (EqExp | Func) ("&&" (EqExp | Func))*

EqExp → (ID ("==" | "!=") Val) | ("(" BExp ")")

Set → "!set" Opt ID ["=" (Val | Func)+] NL

Unset → "!unset" ID NL

IncludeStmt → "!include" ("-b")? FP NL

Options → ”!options” [”-l”] ((”CheckDescrConditionals” (”error” | ”fatal”)?)
| ”MultipleSections” | ”Silent”) NL

ID → [A-Za-z0-9-_]+

Val → (Str | QStr | Ref)+

Str → [.]+

QStr → ”"”[.]+”"”

Ref → ”%”ID”%”

Opt → ”-”[iflea]+

Func → ID ”(” Param (”,” Param)* ”)”

Param → QStr | Ref | ID | Func

FP → (Ref | Drive | RelPath) (Ref | ID)* (File)?

Drive → [A-Za-z]”:\”

RelPath → ”.\”

File → ID ”.” ID

NL → ”\r\n” | ”\r” | ”\n”

section → FileListSec | SourceFileSec | BuildFileSec | BuildDirectoriesSec |
SourceFileOptSec | VariantSec | RelatedVariantSec | VariantSetsSec | Official-
BuildSec |
ModuleSec | OutputNameSec | SystemNameSec | MiscSec

FileListSec → (”[LinkFiles]” | ”[IncludeFiles]” | ”[DocumentFiles]”
| ”[DependantFiles]” | ”[DataFiles]” | ”[ArchiveFiles]” | ”[BuildDirectories]”
| ”[-LinkFiles]” | ”[-IncludeFiles]” | ”[-SourceFiles]” | ”[-DocumentFiles]”
| ”[-DependantFiles]” | ”[-DataFiles]” | ”[-ArchiveFiles]” | ”[-BuildFiles]”
| ”[-SourceFiles]” | ”[Packaging_ExposedDocumentFiles]” | ”[Packaging_ExposedDataFiles]”
| ”[Packaging_ExposedSourceFiles]” | ”[Packaging_ForbiddenFiles]”
| ”[Packaging_ExposedBuildFiles]” | ”[Packaging_ExposedLinkFiles]”
| ”[Packaging_ExposedIncludeFiles]” | ”[-Packaging_ExposedDocumentFiles]”
| ”[-Packaging_ExposedDataFiles]” | ”[-Packaging_ExposedSourceFiles]”

| "[-Packaging_ExposedBuildFiles]" | "[-Packaging_ExposedLinkFiles]"
| "[-Packaging_ExposedIncludeFiles]") (FileList)*

FileList → (FP NL) | Directive

SourceFilesSec → "[SourceFiles]" (SourceFile | Directive)*

SourceFile → FP (SourceFileOpt)*

SourceFileOpt → ID "=" "(" Val ")"

BuildFilesSec → "[BuildFile]" (BuildFile | Directive)*

BuildFile → FP (BuildFileOpt)*

BuildFileOpt → ("DO" | "DEP" | "OUT") "=" "(" Val ")"

BuildDirectoriesSec → "[BuildDirectories]" (BuildDirectory | Directive)*

BuildDirectory → FP ("CLEAN" | "CLEAN_TREE") NL

SourceFileOptSec → "[SourceFiles_Options]" (SourceFileOpt | Directive)*

SourceFileOpt → Pattern (SourceFileOptOpt)+

Pattern → ?

SourceFileOptOpt → ("+")? SourceFileOpt

VariantSec → "[Variants]" (Directive | Variant)*

variant → ID

RelatedVariantSec → "[RelatedVariants]" (Directive | RelatedVariant)*

RelatedVariant → (ID ";" | (ID ";")* (ID | "*")) NL

VariantSetsSec → "[VariantSets]"(Directive | VariantSet)

VariantSet → ID ":" ("@")? ID (":" ID)? (("@")? ID (":" ID)?)*) NL

OfficialBuildSec → "[OfficialBuild]" (OfficialBuild | Directive)*

OfficialBuild → ID ":" ID (":" "KEEP")?(":" "REBUILD")? (";" ID (":"

"KEEP")?(":" "REBUILD")?)* NL

ModuleSec → "[Modules]" (Module | Directive)*

Module → FP ID NL

OutputNameSec → "[OutputName]" (OutputName | Directive)*

OutputName → (ID | Ref) NL

SystemNameSec → "[SystemName]" (SystemName | Directive)*

SystemName → (ID | Ref) NL

MiscSec → ("[MakeDefines]" | "[MakeInit]" | "[MakeRules]"
| "[MakeActions]" | "[IncludePatterns]" | "[MakeIncludeFiles]"
| "[MakeSourceFiles]" | "[MakeDependencies]" | "[MessageFilter]"
| "[ExplicitMakeRules]" | "[ArchiveFiles]" | "[DefaultOptions]"
| "[Name]" | "[IAR-AVR_NO_AUTOSEG]" | "[ToolCommands]"
| "[IAR-ARM_LINK_PAR]" | "[OfficialBuild_Options]"
| "[Packaging_Translate]")
(MiscSecStmt)*

MiscSecStmt → ([.]+ NL | Directive)

Appendix F

Example of generated AST

This appendix contains a sample AST generated from the source code in section 5.5.

```
Start
  Set
   Identifier: A
   Value
    String: 1
  Set
   Identifier: B
   Value
    String: 2
  Include
   FilePath
    RelativePath
    String: file.cfg
  If
   EQNode
    Identifier: A
    Value
     String: 1
   AndNode
    NENode
     Identifier: B
     Value
      String: 2
   Set
    Identifier: C
```

```
    Value
     String: 1
   Elseif
    EQNode
     Identifier: A
     Value
      String: 1
    OrNode
     EQNode
      Identifier: B
      Value
       String: 2
    Set
     Identifier: C
     Value
      String: 2
   Else
    Set
     Identifier: C
     Value
      String: 3
   Endif
  Section: [SourceFiles]
   FileList
    FilePath
     RelativePath
     String: dir\file1.c
   FileList
    FilePath
     RelativePath
     String: dir\file2.c
    SourceFileOption
     Identifier: AVR
     String: -z9
  Section: [MakeRules]
   Text: copy #path $(INC)#name#ext >nul
```

Bibliography

[1] Appel A. W. *Modern Compiler Implementation in Java* Second Edition, pp 42–44, UK 2002.

[2] Java Compiler Compiler^{TM}(JavaCC^{TM}) - The Java Parser Generator.
https://javacc.dev.java.net/

[3] Succi G., Wong R. W., *The Application of JavaCC to Develop a C/C++ Preprocessor* University of Calgary

[4] The Lex and Yacc Page
http://dinosaur.compilertools.net/

[5] Johansson B., Storlind R. *Generic Environment for Application Oriented Languages* ABB Corparate Research

[6] Domain-Specific Languages
http://compose.labri.fr/documentation/dsl/

[7] Visitor Support in JavaCC/JJTree
http://www.xrce.xerox.com/people/beesley/VisitorJJTree.html

[8] Kompilatorteknik, 5 p, Lund Institute of Technology
http://www.cs.lth.se/EDA180/2005/