Phoenix Star Dawgs

EECS 348 Group Project
Software Requirements Specifications

Version 1.5

Revision History

Date	Version	Description	Author
03/08/2024	1.0	Added section 4: Classification of Functional Requirements	Katie Nordberg
3/14/2024	1.1	Part 1	Ibrahim
3/20/2024	1.2	Part 2	Shravya
3/21/2024	1.3	Part 3	Evans
3/21/2024	1.4	Cleaned up sections 1, 2, and 3	Katie
3/23/2024	1.5	Bring all sections into one document	Katie

Table of Contents

Software Requirements Specifications

1. Introduction

1.1 Purpose

The goal of this is to act as a document that lays out the needs for the Boolean Logic Calculator project. Its purpose is to offer an explicit explanation of how the application should function. This involves defining the tasks for the Boolean Logic Calculator and any restrictions or boundaries that need to be followed throughout the development process.

1.2 Scope

The scope of this document is to document the software side of requirements for this project. This program is designed to interpret and assess Boolean logic input, by users. It has operators like AND, OR, NOT, XOR and more which are described in section 3: Specific Requirements. The program will return true or false using the supported logic operators based on the user's prompt.

1.3 Definitions, Acronyms, and Abbreviations

AND – The AND operation takes two Boolean inputs and produces an output that is true only when both inputs are true.

OR – The OR operation takes two Boolean inputs and produces an output that is true when one input is true, or both are true.

NOT - This operation takes a Boolean value and returns the opposite Boolean value.

NAND - It combines AND and NOT. This returns the inverted value of AND.

NOR- It combines OR and NOT. This returns the inverted value of OR.

XOR – The XOR operation takes two Boolean inputs and produces an output that is true if exactly one of the inputs is true.

Boolean – Boolean expressions take one of two possible values, true or false. Under positive logic, 1 is used to denote true and 0 is used to denote false.

1.4 References

EECS 348 Spring 2024 Project Description Document obtained from Professor Hossein Saiedian.

1.5 Overview

The overview section describes the contents of the SRS and explains how the document is organized. It provides a brief preview of subsequent sections, including requirements, design constraints, and supporting information for the Boolean Expression Evaluator software project. The following sections describe the interfaces for the product and all the operations it will do. It will go over the constraints of the program, functionality and functional requirements.

2. Overall Description

2.1 Product Perspective

2.1.1 System Interfaces:

Behind the scenes, the program will interface with system resources for memory management and error handling. This will be handled by the operating system and not our program.

2.1.2 User Interfaces:

The user interfaces with the program through a command-line interface. The user will enter Boolean expressions to be evaluated through the command line. Finally, the command line will be used to display the calculated results of the evaluations to the user. The interface should be user-friendly and provide clear prompts and feedback.

2.1.3 Hardware Interfaces:

The program does not have direct hardware interfaces as it operates solely in software. However, it relies on the underlying hardware for memory allocation and CPU processing.

2.1.4 Software Interfaces:

The software interfaces with the C++ standard library for input/output operations and string manipulation. Additionally, it may interface with external libraries or modules for specific tasks such as parsing.

2.1.5 Communication Interfaces:

There are no explicit communication interfaces required as the program operates locally on the user's machine without any network interaction.

2.1.6 Memory Constraints:

The program should manage memory efficiently to handle expressions of varying complexity while adhering to memory constraints of the system. This includes proper memory allocation and deallocation to avoid memory leaks.

2.1.7 Operations

The program performs operations such as expression parsing, evaluation, error handling, and user interaction. It also involves managing data structures to represent Boolean expressions and their final evaluation values.

2.2 Product Functions:

- Parse Boolean expressions containing operators (!, $, @, &, |) and Boolean constants such as T for True and F for False.
- Evaluate the parsed expressions according to operator precedence rules outlined in 3.1.
- Handle parentheses to determine the order of evaluation within expressions.
- Recognize and handle Boolean constants within expressions.
- Provide a user-friendly command-line interface for inputting expressions and displaying results.
- Implement robust error handling to manage scenarios outlined in 3.1.

2.3 User Characteristics

Users of the program are expected to have basic knowledge of Boolean expressions and the operators involved. They should be comfortable using a command-line interface for input and output.

2.4 Constraints

- The program must be written in C++.
- Memory usage should be optimized to handle expressions of varying sizes without exceeding system limits.
- Error handling should be comprehensive to handle all possible scenarios gracefully.
- The program will use object-oriented programming principles.
- The program will include comments to explain the logic and functionality of the code to further readers and programmers on the team.
- The program will have unit tests to verify the correctness of different operator evaluations and complex expressions.
- The program must provide clear and informative error messages that the user will find easy to understand.

The program will have a user-friendly text-based interface for entering Boolean expressions and viewing the results

2.5 Assumptions and Dependencies

- The program assumes that the input expressions provided by users may not be well-formed and as such our program will test their validity.
- Dependencies include the C++ standard library for input/output operations and possibly external libraries for specific functionalities like parsing.

2.6 Requirements Subsets

Requirements subsets include tasks related to expression parsing, operator precedence, parenthesis handling, Boolean constants recognition, user interface development, error handling, and any additional features or optimizations beyond the specified requirements. These subsets are essential for organizing and prioritizing tasks during development.

3. Specific Requirements

3.1 Functionality

Complex Expression Handling:

The system should be able to handle complex expressions with multiple gates and input/output signals. It should support expressions of significant complexity while maintaining efficient evaluation.

User Interface:

The system should provide a graphical user interface to input Boolean expressions and truth values. The output should be presented in a clear and understandable format.

3.1.1 Operator Support

The program should support the implementation of logical operations for the following Boolean operators:

AND (&): Returns True if both operands are True.

OR (|): Returns True if at least one operand is True.

NOT (!): Inverts the truth value of its operand.

NAND (@): Returns True only if both operands are False (opposite of AND).

XOR ($): Returns True if exactly one operand is True.

These operators should be implemented as functions within the program, allowing for their use in evaluating Boolean expressions provided by the user. Additionally, the program should ensure proper handling of these operators according to their precedence rules and parentheses in the input expressions. The program should accurately implement the behavior of each logical operator as described above. The logical operators should be functional within the context of evaluating Boolean expressions provided by the user. Precedence rules for operators should be followed correctly, ensuring the correct order of evaluation in complex expressions. Parentheses should be properly handled to override operator precedence and ensure the desired order of evaluation. The program should provide clear error messages if invalid expressions or unsupported operators are encountered during evaluation. Tests should be developed to verify the correctness of each logical operator implementation under various scenarios. The program should be structured in an object-oriented manner using C++, adhering to best practices for code organization and modularity.

3.1.2 Expression Parsing

The program should have the ability to parse logical expressions entered by the user. These expressions will be in infix notation which is the typical way math is written. Infix notation has operators between the operands rather than at the end.

While parsing user input, the program must recognize operator precedence including parentheses and nested parentheses.

3.1.3 Truth Value Input

The program must allow users to define truth values for variables within the program. Truth values can take on either true represented by T, or false represented by F. The variables will be used to evaluate further logical expressions entered by the user.

3.1.4 Evaluation and Output

The program must correctly calculate the final truth value of a logical expression. It must take into account operator precedence and parentheses. As the program will support operators with unstandardized precedence (XOR and NAND), we will follow this precedence table:

Parentheses	first to be evaluated
NOT (!)	
XOR ($), NAND (@)	Evaluated from left to right in the expression.
AND (&)	

OR (|)	last to be evaluated

The program must also present the final answer clearly to the user. The final answer will either be True or False.

3.1.5 *Error Handling*

The program will handle the following potential errors:

- Missing parentheses
- Unknown Characters

  Unknown characters are letters or symbols that carry no meaning in our program. This may be undeclared variables or unsupported operators. We make no distinction between these as there is no way to tell the difference between an unknown operand and an unknown operator as while the user may enter an unknown operator, the program could interpret it as an unknown operand with a missing operator or vice versa. Therefore, we will keep the message simple and group unknown operands and unknown operators into a single error event.

- Missing Operands
- Empty expressions

The program will also provide informative error messages to aid the user in identifying and resolving errors.

3.1.6 Parentheses Handling

The program will handle input expressions using parentheses. It will support unnecessary but mathematically correct parentheses, i.e. (a+b)+c will not raise an error. It will support nested parentheses, i.e, (a*(b+c)) will not raise an error. The program will determine the order of evaluation of the expression based on the parentheses. The program will evaluate expressions with respect to the most nested expression first and the least nested expression last.

3.1.7 Truth Table

An optional requirement that we would like our program to have is the ability to display a truth table showing the values of each input, each intermediate expression, and the final expression.

3.2 **Use-Case Specifications**

Use Case: Evaluate Boolean Expression

- **Actor**: User
- **Description**: The user provides a Boolean expression, and the system evaluates it, returning the result.
- **Preconditions**: The program is running and ready to receive input.
- **Postconditions**: The system displays the truth value of the evaluated expression.

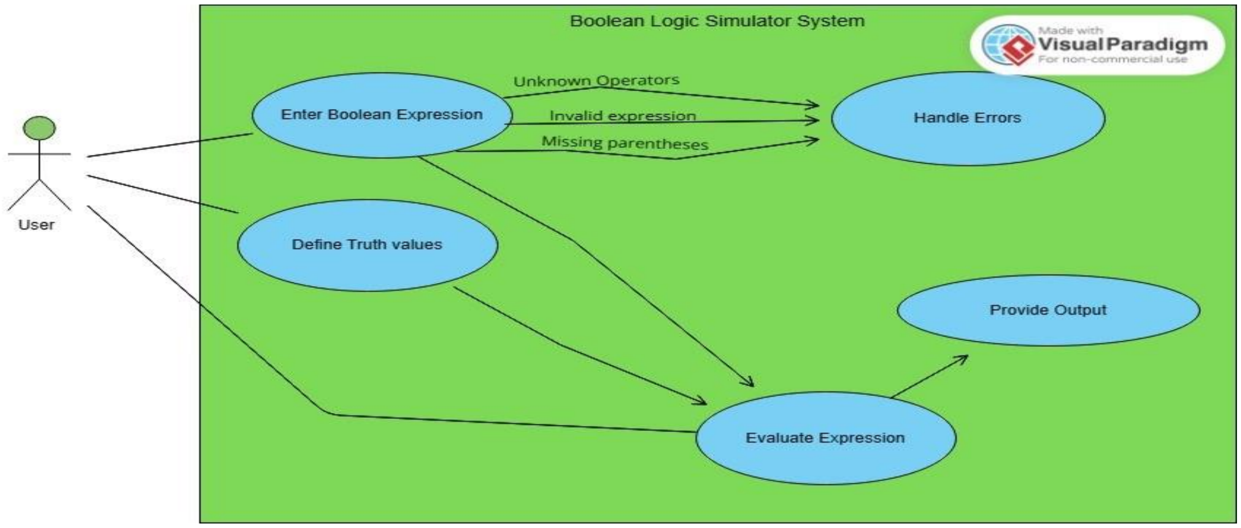
- **Main Flow**:
  1. User inputs a Boolean expression.
  2. System parses the expression and evaluates it.
  3. System displays the truth value of the evaluated expression.
- **Alternative Flows**:
  1. If the expression contains invalid syntax or unknown operators, the system displays an error message and prompts the user to input a valid expression.
  2. If the expression contains unknown operators, the program displays an error message indicating the unsupported operators and prompts the user to correct the expression.

Use Case: Define Truth Values

- **Actor**: User
- **Description**: The user provides truth values for variables in the Boolean expression.
- **Preconditions**: The program is running and ready to receive input.
- **Postconditions**: The truth values are assigned to the variables in the expression.
- **Main Flow**:
  1. User specifies truth values for each variable represented in the expression.
  2. The program assigns these truth values to the respective variables in the expression.
- **Alternate Flows:**
  1. If the user provides an invalid truth value (neither True nor False), the program displays an error message and prompts the user to provide a valid truth value.

Use Case: Handle Errors

- **Actor**: System
- **Description**: The program handles errors encountered during expression parsing or evaluation.
- **Preconditions**: The program is running.
- **Postconditions**: The program displays informative error messages for invalid input or situations.
- **Basic Flow**:
  1. During expression parsing or evaluation, if an error is encountered (e.g., invalid syntax, unknown operators), the program generates an informative error message.
  2. The error message is displayed to the user.
- **Alternate Flows**: None.



3.3 Supplementary Requirements

Efficiency:

- The system should be efficient in parsing and evaluating Boolean expressions, ensuring reasonable performance even for complex expressions.

Portability:

- The program should be platform-independent, ensuring it can run on different operating systems (Windows, Linux, MacOS, etc.) without modification. Compatibility with popular C++ compilers should be maintained to facilitate ease of deployment.

Performance:

- The system should be able to evaluate expressions within a reasonable time frame, even for complex expressions with multiple variables and operators. The response time for parsing and evaluating expressions should be minimal to provide a smooth user experience.

Reliability:

- The system should handle invalid input gracefully, providing clear error messages to guide users. Robust error-handling mechanisms should prevent unexpected crashes or errors during expression evaluation.

Scalability:

- The system should be able to handle a growing number of variables and expressions efficiently. It should scale gracefully as the complexity of the logic circuits increases.

4. Classification of Functional Requirements

Functionality	**Type**
1. Operator Support a. AND (&) Support: `n & k` Returns True if both n and k are True.	Essential

	Returns False otherwise.	
	b. OR (|) Support: `n | k` Returns True if at least one of n or k are True. Returns False otherwise.	
	c. NOT (!) Support: `!n` Returns the opposite Boolean value of n.	
	d. NAND (@) Support: `n @ k` Returns True if both n and k are False. Returns False otherwise.	
	e. XOR ($) Support: `n $ k` Returns True if only one of n or k are True. Returns False otherwise.	
2.	Expression Parsing: Parse Boolean expressions entered by the user. Expressions will be given in infix notation. (Infix notation is the common way of writing mathematical expressions.) These expressions will be interpreted respecting operator precedence and parentheses. This requirement is not to evaluate the expression (that is requirement 4). Rather, this requirement describes the need to take in a user inputted expression and determine which operations and truth values must be used.	Essential
3.	Truth Value Input: Allow users to define truth values for each variable. Users will be able to input T or F to represent True or False respectively. These values will be used to define the Boolean value of variables which will then be used in expressions to be evaluated.	Essential
4.	Evaluation and Output: Evaluate expressions according to the operator precedence defined by feature. Then display the Boolean result of the entire expression. There is no single defined precedence for logical operators. Each language defines their own precedence. After researching many ways, we will use the following precedence:	Essential
	Parentheses — first to be evaluated	
	NOT (!)	
	XOR ($), NAND (@) — Evaluated from left to right in the expression.	
	AND (&)	
	OR (|) — last to be evaluated	
5.	Error Handling: Provide a helpful and descriptive message to the user when an error occurs. After an error is discovered and the user is informed, the program will prompt the user to re-enter the expression. This cycle will continue until a valid expression is entered such that it is properly evaluated. The following errors must be handled by the program without crashing.	Essential

a. Missing Parentheses b. Unknown characters (i.e. letters or symbols that carry no meaning in our program. This may be undeclared variables, or unsupported operators. We make no distinction between these as there is no way to tell the difference between an unknown operand and an unknown operator because while the user may enter an unknown operator, the program could interpret it as an unknown operand with a missing operator or vice versa. Therefore, we will keep the message simple and group unknown operands and unknown operators into one error.) c. Missing Operands d. Empty Expression	
6. Parentheses Handling: Expressions should be evaluated with respect to the most nested expression first and the least nested expression last. This feature requires that our program support parentheses (including nested parentheses) in user-entered expressions.	Essential
7. Display a truth table showing the values of each input, each intermediate expression, and the final expression. This is marked as desirable because it is not a requirement of the project. However, it should not be extremely difficult to implement, and it would earn the project extra credit.	Desirable