<Project Name>

Version <1.0>

[Note: The following template is provided for use with the Unified Process for EDUcation. Text enclosed in square brackets and displayed in blue italics (style=InfoBlue) is included to provide guidance to the author and should be deleted before publishing the document. A paragraph entered following this style will automatically be set to normal (style=Body Text).]

[To customize automatic fields in Microsoft Word (which display a gray background when selected), select File>Properties and replace the Title, Subject and Company fields with the appropriate information for this document. After closing the dialog, automatic fields may be updated throughout the document by selecting Edit>Select All (or Ctrl-A) and pressing F9, or simply click on the field and press F9. This must be done separately for Headers and Footers. Alt-F9 will toggle between displaying the field names and the field contents. See Word help for more information on working with fields.]

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| <dd/mmm/yy> | 1.0 | Fill out section 3 | Evans |
| 3/21/2024 | 1.1 | Clean up section 3 | Katie |
|  |  |  |  |
|  |  |  |  |

Table of Contents

1. Introduction 4

1.1 Purpose 4

1.2 Scope 4

1.3 Definitions, Acronyms, and Abbreviations 4

1.4 References 4

1.5 Overview 4

2. Overall Description 5

2.1 Product perspective 5

2.1.1 System Interfaces 5

2.1.2 User Interfaces 5

2.1.3 Hardware Interfaces 5

2.1.4 Software Interfaces 5

2.1.5 Communication Interfaces 5

2.1.6 Memory Constraints 5

2.1.7 Operations 5

2.2 Product functions 5

2.3 User characteristics 5

2.4 Constraints 5

2.5 Assumptions and dependencies 5

2.6 Requirements subsets 5

3. Specific Requirements 5

3.1 Functionality 5

3.1.1 <Functional Requirement One> 6

3.2 Use-Case Specifications 6

3.3 Supplementary Requirements 6

4. Classification of Functional Requirements 6

5. Appendices 6

# Introduction

[The introduction of the **Software Requirements Specification (SRS)** provides an overview of the entire **SRS**. It includes the purpose, scope, definitions, acronyms, abbreviations, references, and overview of the **SRS**.]

[Note: The **SRS** captures the complete software requirements for the system, or a portion of the system. Following is a typical **SRS** outline for a project **using use-case modeling**. This artifact consists of a package containing use cases of the use-case model and applicable Supplementary Specifications and other supporting information.]

[Many different arrangements of an **SRS** are possible. Refer to [IEEE830-1998] for further elaboration of these explanations, as well as other options for **SRS** organization.]

## Purpose

[Specify the purpose of this **SRS**. The **SRS** fully describes the external behavior of the application or subsystem identified. It also describes nonfunctional requirements, design constraints, and other factors necessary to provide a complete and comprehensive description of the requirements for the software.]

## Scope

[A brief description of the software application that the **SRS** applies to, the feature or other subsystem grouping, what Use-Case model(s) it is associated with, and anything else that is affected or influenced by this document.]

## Definitions, Acronyms, and Abbreviations

[This subsection provides the definitions of all terms, acronyms, and abbreviations required to properly interpret the **SRS**. This information may be provided by reference to the project’s Glossary.]

## References

[This subsection provides a complete list of all documents referenced elsewhere in the **SRS**. Identify each document by title, report number if applicable, date, and publishing organization. Specify the sources from which the references can be obtained. This information may be provided by reference to an appendix or to another document.]

## Overview

[This subsection describes what the rest of the **SRS** contains and explains how the document is organized.]

# Overall Description

[This section of the **SRS** describes the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in detail in Section 3, and makes them easier to understand. Include such items as:

## Product perspective

### System Interfaces

### User Interfaces

### Hardware Interfaces

### Software Interfaces

### Communication Interfaces

### Memory Constraints

### Operations

## Product functions

## User characteristics

## Constraints

## Assumptions and dependencies

## Requirements subsets

# Specific Requirements

## 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 interfaceto input Boolean expressions and truth values. The output should be presented in a clear and understandable format.

### **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.

### **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.

### **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.

### **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.

### **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.

### **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.

### **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.

## 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 prZ ovides 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.

## 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.

# Classification of Functional Requirements

[List, usually in a table, all functional requirements and order them by Type (Essential, Desirable, and Optional) or by order of appearance in the document.]

|  |  |
| --- | --- |
| **Functionality** | **Type** |
| ... |  |
| ... |  |

# Appendices

[When appendices are included, the **SRS** should explicitly state whether or not the appendices are to be considered part of the requirements]