Boolean Logic Legends

Boolean Logic Simulator in C++
Software Architecture Document

Version 1.1

Revision History

Date	Version	Description	Author
03/Apr/24	1.0	Initial Document	Sam Grimsley
14/Apr/24	1.1	Completed document with initial architecture description	All group members

Table of Contents

Software Architecture Document

1. Introduction

The *Software Architecture Document* (SAD) serves as a comprehensive overview of the architectural design decisions made for the Boolean Logic Simulator (BLS) to ensure understanding of project goals and requirements.

1.1 Purpose

The purpose of the SAD is to serve as a guide for understanding the architectural design decisions made for the software. Its primary objectives are to document architectural decisions and guide development. This document is intended to be used by developers, stakeholders, and anyone involved with this software. The SAD is used to facilitate alignment with project goals and requirements between everyone and ensures consistency during development.

1.2 Scope

The SAD applies to a boolean logic simulator, addressing all aspects of its architectural design and implementation. This includes defining structural framework, architectural goals and constraints, component interactions, and all other considerations necessary for the development of the boolean logic simulator.

1.3 Definitions, Acronyms, and Abbreviations

- SAD: Software Architecture Document
- BLS: Boolean Logic Simulator
- IDE: Integrated Development Environment
- API: Application Programming Interface
- GoogleTest: A library utilized for unit testing for C++ and C

1.4 References

Title	Date	Source	Link
Boolean Logic Simulator in C++ Software Development Plan (Version 1.2)	Mar 22, 2024	EECS 348 Project Google Drive	EECS 348 Project-Plan

1.5 Overview

Architectural Representation	Outlines what software architecture means for the BLS and lists the views and what elements are needed to understand the architecture.
Architectural Goals and Constraints	Outlines goals and limitations that affect the development of the BLS as well as special constraints.
Logical View	Describes architecturally significant parts for the BLS.
Interface Description	A description of the user-interface of the BLS.

Quality	A description of how the software architecture contributes to the capabilities of the system beyond functionality.

2. Architectural Representation

Software architecture for the Boolean Logic Simulator system is the fundamental structure of the program, defining how it is organized, how different components interact, and how it will evolve over time. This architecture is necessary to ensure the system accurately simulates logic operations and provides a user-friendly interface. It lays out a blueprint for the developers and guides the construction of the system. The architecture is represented through a series of interrelated viewpoints, each focusing on specific aspects of the system.

- Context Viewpoint: The system is designed to simulate Boolean logic operations within a simplified scope, focusing on basic logical operators and user-defined truth values. Users interact with the system through a text-based interface to input Boolean expressions and receive evaluations. The project relies on the C++ standard library for its development. There's minimal interaction with external systems.
- Functional Viewpoint: The system will comprise multiple modules and error handling throughout. There will be a user interface module that enables user interaction likely in a command-line. It's responsible for input collection, and displaying the truth table or any errors to the user. There will also likely be a parsing module that will analyze the input Boolean expressions, converting them to a structured format that can easily be evaluated. The last module will be the evaluation module, which traverses the structured representation created by the parsing module and computes the truth table. This module applies the correct logic for Boolean operations.
- Information Viewpoint: The simulator will utilize data structures such as trees or stacks to represent the parsed expressions. These structures support the efficient evaluation of expressions based on operator precedence and parentheses. It will also manage a mapping of variables to their user-defined truth values, allowing the evaluation of expressions involving these variables.
- Development Viewpoint: The simulation will utilize C++ and its standard library, with a focus on object-oriented programming. The development tools include a C++ compiler, IDE (Visual Studio Code), and possibly a testing framework. The code structure will include well-encapsulated modules with clear interfaces and responsibilities. There will be adequate comments and documentation within the codebase to help with understanding and possible future maintenance.
- Deployment Viewpoint: The runtime environment is designed to run on various platforms supporting the C++ runtime environment. Minimal external dependencies ensure that the simulation will be compatible and easy to deploy.
- Operational Viewpoint: The user interface offers a user-friendly interface for inputting expressions and viewing the results. Error messages are designed to be informative and helpful so users can correct mistakes. The system will be maintained and supported through procedures for reporting bugs and requesting features. Updates will address bug fixes and performance improvements.

3. Architectural Goals and Constraints

Architectural goals and constraints for this project cover several aspects of both the project development and the final product, and may include:

- Modularity: In taking an architectural approach, implementation of the project should remain modular, allowing for several separate elements necessary for the end product to not only be developed simultaneously but be separately modifiable if and when issues arise. This will aid in the development of the project as well as testing and modification, allowing problems to be viewed in a localized environment within the project, rather than as part of the program at large.
- Scalability: Everything created or used for this project should be capable of scaling with the project. There should be no issues that arise due to increased workload or complexity. It is worth noting that there should be minimal issues in this area; as no data is stored and the program is run locally, there is not much scaling possible here.
- Portability: Software should be capable of running on most machines. The simulator should not be hardware-dependent, being able to run on virtually any computer capable of running C++.

- Security: No user data should be stored for this program. As such, security will be focused more on aspects of stability, such as ensuring there are no memory leaks.
- Stability: The logic simulator should run effectively when given various forms of input. There should be no crashes, runtime errors, or memory leaks.
- Transparency: Using a modular approach, all architecture should be thoroughly documented and explained through the use of inline comments or separate documentation, thus allowing for more ease of modification should issues arise in the future that need addressing.
- Testability: The program should have several independent features that can be easily tested and evaluated. Furthermore, the design of the simulator should be such that testing can be done using tools such as GoogleTest to analyze program behavior and stability.
- Additional development structure: Additional aspects of note relevant to the project development include team structure, schedule, and other tools. The team structure is divided into team leaders, team administrators, quality assurance engineers, and technical leaders. More information about this can be found in the *Project Plan* document. The project must be completed by the beginning of May. A more detailed schedule may also be found in the *Project Plan* document. Lastly, additional tools of use for this project include GitHub, for code and documentation management and collaboration; Google Docs, for documentation; and VSCode or other code editors, for code writing, testing, and evaluation.

4. Logical View

SubSystem and Packages

- User Interface Package
  - Windows Class used for coordinating the actions of the user
  - Canvas Class handling user interactions and rendering the gate icons
  - Console Class will display messages and notifications in the console
- Logic Simulation
  - Logicgate class helps with evaluating inputs
  - AND, OR, NOT, NAND, XOR, subclasses representing specific classes of the gate types
  - Simulation class that helps simulate the initialization of gates, setting input and also updating the output values if they need to be updated

Architecturally Significant Classes

- Logicgate Class
  - Inputs and outputs, being able to calculate the output given the logic function that has been entered,
- Canvas Class
  - Being able to manage the gate, and the user interaction
  - Having a list of all the possible gate options for this given architecture

Important Relationships

- Canvas class containing the gates from the LogicGate Class, having the logic gates be associated through input-output connections during the simulation

4.1 Overview

User Interface Package:

Description: This package encompasses the user interface components responsible for facilitating user interactions and providing a graphical representation of logic gates.

Contained Classes: Windows, Canvas, Console

Responsibilities: The classes within this package handle user input, render gate icons on the canvas, and display messages to the user.

Logic Simulation Package:

Description: This package focuses on simulating logic gates and evaluating logic expressions based on user input.

Contained Classes: Logicgate, Operators, Simulation.

Responsibilities: The classes within this package represent different types of logic gates and manage the simulation process, including initialization, input setting, and output updating.

4.2 Architecturally Significant Design Modules or Packages

User Interface Package Classes:

- Windows:
  - Description: This class serves as the main controller for user interactions, managing the overall flow of the application.
- Canvas:
  - Description: This class manages the graphical canvas where users input logic data.
- Console:
  - Description: Provides a console interface for displaying system messages and error logs.

Logic Simulation Package Classes/Subclasses:

- Logicgate
  - Description: Abstract class representing a generic logic gate with inputs and outputs.
- AND, OR, NOT, NAND, XOR
  - Description: These are subclasses of the logic gate and represent the specific types of logic gates applicable to the logic gate class.
- Simulation:
  - Description: Controls the logic simulation process, including gate initialization and input setting and conversions.

5. Interface Description

Screen Formats:

- The user interface presents a command-line interface where users input Boolean expressions and view the generated truth table.
- Make sure there is a console output to display messages, status updates, and errors

Valid Inputs:

- Users can input predefined elementary boolean logic operators.
- Users can start, pause, and stop the simulation
- Operands including variables, truth values, and parentheses

Valid Outputs:

- The system outputs the generated truth table, displaying the truth values for variables and the corresponding evaluation results for the expression.
- Error messages for invalid inputs including a description of what error is being made.

6. Quality

Extensibility: Having the simulator support an extension makes it so we can add new features in the future. Make sure our Modular design is easy to understand so that adding new gate types will be easier to implement.

Reliability: The Software should be able to handle errors, if an invalid input is entered it should be able to handle it and print out what is wrong if it is a runtime error, or/and a memory failure. Putting the architecture through different scenarios to test the framework.

Portability: Not using dependencies that are specific for C++ using the standard libraries so that when porting it to a different operating system it will be easy to port without having to overhaul the system.

Safety & Security: Having some sort of access control restriction that way the user is not able to access certain features within the simulation. A good idea but optional is having data encryption if there are sensitive values.