Development and Implementation of a Simulator for Synchronous Sequential Circuit

The Final Year Project Report

Submitted by

Anirban Banerjee (Roll No: 111005049)
Anurag Chatterjee(Roll No: 111005021)
Subhasish Kundu (Roll No: 111005025)

In Partial fulfilment of the degree of Bachelor of Engineering.

Under the Guidance of

Prof. Manas Hira

Date: May 26, 2014

Department of Computer Science and Technology
Indian Institute of Engineering Science and Technology, Shibpur
PO: Botanic Garden, Dist: Howrah,
West Bengal, India - 711103

Acknowledgement

We are highly indebted to Professor Manas Hira for his guidance and constant supervision as well as for providing necessary information regarding the project. We would like to extend our sincere thanks to all others who have helped us. Finally, yet importantly words are inadequate to express our heartfelt thanks to our beloved parents and teachers for their blessings, our friends for their help and wishes for the successful completion of this project.

Date: May 26, 2014

(Anirban Banerjee)

(Anurag Chatterjee)

(Subhasish Kundu)

CERTIFICATE

This is to certify that the project work entitled "**Development and Implementation of a Simulator for Synchronous Sequential Circuit**" being submitted by Anirban Banerjee (Roll No.: 111005049), Anurag Chatterjee (Roll No.: 111005021) and Subhasish Kundu (Roll No.: 111005025), final year students of Bachelor of Engineering (Department of Computer Science and Technology) of Indian Institute of Engineering Science and Technology, Shibpur, is a record of the work which has been carried out under my supervision.

_____
(Manas Hira)
Project Guide

Countersigned by

_____
(S. Das Bit)
Head of the Department

Department of Computer Science and Technology
Indian Institute of Engineering Science and Technology, Shibpur
Howrah, West Bengal
India – 711103

Contents:

Abstract

In a real life system, before a circuit is designed and fabricated, generally a software model of the system is designed to simulate it. The simulation model is used to compare the behaviour of the model to the behaviour of the actual circuit. In this project the model of synchronous sequential circuit is built using formulae of temporal logic. Programs representing the circuit are written in Tempura. An interpreter to parse the Tempura statements has been developed. A graphical user interface is also made that takes these user programs as input and displays the output of the program in the form of visual graphs.

Chapter 1
Introduction

Simulation is one interesting and useful mechanism for identifying the correctness (incorrectness) of a design of a system. In place of a building a real life system or its prototype, in simulation one prepares a **model** of the system. The model, then, is allowed to function and its **behavior** is observed and compared with the expected behavior of the system.

In this project, a graphical simulator for a synchronous sequential circuit is built. For simulating every real life system, it is required to define a model of the corresponding system with the help of a formal, unambiguous language. Temporal Logic has been chosen here for this purpose. Tempura is an executable subset of Interval Temporal Logic. It has the two seemingly contradictory properties of being a logic programming language and having imperative constructs such as assignment statements and also provides a way of directly executing suitable temporal logic specifications of digital circuits, parallel programs and other dynamic systems. Simulator is taking Tempura specification of a synchronous sequential circuit and is producing a graphical output according to the output of Tempura interpreter. A Tempura interpreter has also been built for this purpose.

Chapter 2
Objective

Under this project a simulation **environment** for **synchronous sequential circuits** has been developed. Sequential circuits have been modeled in terms of **formulae** of **temporal logic** (more specifically, **Tempura**, an executable subset of **interval temporal logic**). A sequential circuit is made available to the simulator as an interconnection networks of components where behavior of each of those components is also made available as formulae of Tempura. The simulator is presenting a visual display of the output signal of the circuit with respect to a controllable clock signal.

Chapter 3
Temporal Logic

This chapter contains a brief description of Temporal Logic that has been used in the project. First brief introduction to Temporal Logic and Interval Temporal Logic is discussed. After that, Tempura and its basic operators are discussed.

Temporal Logic is a useful tool for reasoning about concurrent programs and hardware. It has logical operators corresponding to various time-dependent concepts such as “always” and “sometimes”.

3.1 Interval Temporal Logic

Interval Temporal Logic is a Temporal logic for representing both propositional and first-order logical reasoning. Periods of time can be handled in both sequential and parallel composition. Interval temporal logic deals with finite sequences instead of infinite sequences.

3.2 Tempura

Tempura is an executable subset of Interval Temporal Logic. It has two seemingly contradictory properties of being a logic programming language and having imperative constructs such as assignment statements. Thereby providing a way to directly executing suitable temporal logic specification of digital circuit and parallel programming and other dynamic systems i.e it can be used to write programs that are written in programming language such as C, C++, Java etc, as well as use it to simulate the output of digital circuit before fabrication.

3.3 Basic Temporal Operators

In this section we will be discussing some basic or rather some frequently used temporal operators with some associated examples.

- **Next operator:** The operator 'next' is the Tempura form of the operator O in temporal logic. The term “next …” causes “...” to be executed on the next state. There must be a next state, and the execution of the part within the next operator will occur in the next state.
  Eg. next(I = 0) means that there exits a state after this state and during execution of that state the identifier 'I' will be assigned the value 0. But this will fail if the next state is empty. This operator can also be applied to an expression yielding its value in the next state.

- **Gets operator:** It is similar to assignment operator, only difference being in this case the assignment is done after unit delay.
  Eg I gets J means that the value of I in the next state will be the value of J in the present state. Thus I is assigned the value of J but in the next state instead of in the present state. Another important thing about gets operator is that the LHS of gets must always be an identifier but the RHS can be a constant, or an expression or even an identifier.

- **Always operator:** The operator always is the Tempura form of the [] in temporal logic. The expression that is within the always operator is executed in every state.
  Eg. always(I =J+1) means that the value of I in every state is the value of J plus one i.e I will be assigned a value one more than J. But always(next(I = 0)) this will not work since the next operator implies that there is a state after this, but this can not be executed for the last state.

- **Len operator:** The len construct is used to define any interval length. Thus if len(n) is encountered then it defines an interval of length n(containing n+1) states. So, Len(0) means 'empty' i.e this is the

only state and there is no state after this. Len(1) means 'skip' i.e there is a single interval with two states.

- **Halt operator:** The halt operator defines the termination condition for an interval. The halt operator has an expression as an argument and it evaluates whether this expression is true or not if false then it continues execution but the state at which it is true it executes that state and stops execution of states after that.

  Eg. **I = 0 and I gets I+1 and halt(I ==5).** means that in the 1st state the value of I is 0 and from next state onwards the value of I is incremented by the gets operator by 1 so in 6th state the condition in the halt is true so it will stop after executing that state.
  Thus I = 0 and I gets I+1 and len(5) and the above eg are same as len also specifies an interval of length 5 or having 6 states.

Chapter 4
Proof of Concept

This section contains the proof based on which the implementation is based. The proof starts with a discussion of the weak next operator, how other Temporal operators can be expressed using the weak next operator and how to break a Tempura program into what happens immediately and what happens in the future states.

4.1 The Weak Next Operator

In order for the construct next w to be true on an interval σ, the length of σ must be at least 1. It can therefore be refered as strong next. The related construct weak_next w is called weak next and is true on an interval σ if either σ has length 0 or the sub-formula w is true on σ1 . . . σ|σ|. It can be expressed weak next in terms of strong next:

weak_next w ≡def empty ∨ next w.

The operator weak next provides a concise and natural way to express a temporal logic construct as a conjunction of its immediate effect and future effect.

4.2 Weak Next Form of Temporal Operators

Here are the ways of expressing common temporal operators using weak_next operator:

1. always w ≡ w ∧ weak_next (always w)
2. e1 = e2 ≡ (e1 = e2) ∧ weak_next (true)
3. e1 gets e2 ≡ always(e1 = e2) ≡ (e1=e2) ∧ weak_next (e1 = e2)
4. halt w ≡ true ∧ weak_next(halt w)
5. len w ≡ true ∧ weak_next(len (w-1))
6. next w ≡ true ∧ weak_next(w)

The weak next form of Temporal constants are:

1. weak_next(empty) = false
2. weak_next(more) = true

4.3 Expressing A Tempura Program In Weak Next Form

From the previous section, it is clear that the temporal operators can be expressed as an expression involving the weak next operator. So any Tempura expression can be expressed using the weak next operator, by aggregating the present state operations and the subsequent operations for all the Temporal operators. Thus, given any Temporal formula one way to execute such a formula is to transform it to a logically equivalent conjunction of the two formulas present_state and weak_next of what_remains:

present_state ∧ weak_next (what_remains).

Here, the formula present_state consists of assignments to the program variables and also indicates whether or not the interval is finished. The formula what_remains is what is executed in subsequent states if the interval does indeed continue on. Thus, it can be viewed as a kind of continuation.

Example of expressing a Tempura program in weak next form

Considering this formula:
(next next empty) ∧ (I = 0) ∧ (I gets I + 1) ∧ always(J = 2 · I)

For the formula under consideration, present_state has the following value:
(I = 0) ∧ (J = 0).

The value of what_remains is the formula
(next empty) ∧ (I = 1) ∧ (I gets I + 1) ∧ always(J = 2 · I)

The following shows the effect of the above transformations before and after each of the three states of execution:

Before state 0:
(next next empty) ∧ (I = 0) ∧ (I gets I + 1) ∧ always(J = 2 · I)
After state 0:
[(I = 0) ∧ (J = 0)]
∧ weak_next [(next empty) ∧ always(I = I + 1) ∧ always(J = 2 · I)]

Before state 1:
(next empty) ∧ always(I = I + 1) ∧ always(J = 2 · I)
After state 1:
[(I = 1) ∧ (J = 2)]
∧ weak_next [empty ∧ always(I = I + 1) ∧ always(J = 2 · I)]

Before state 2:
empty ∧ always(I = I + 1) ∧ always(J = 2 · I)
After state 2:
[(I = 2) ∧ (J = 4)]
∧ weak_next [false ∧ always(I = I + 1) ∧ always(J = 2 · I)]

The execution terminates here since the expression before state 3 is false, which is obtained by a conjunction in which a single operand is false.

Chapter 5
Simulator: The Design

This section contains the steps that have been followed to construct the simulator. In particular the construction of the lexer, the parser, the data structure which is suitable for simulation, algorithm for the Temporal operators on the expression tree, implementation details and the construction of the graphical user interface are discussed.

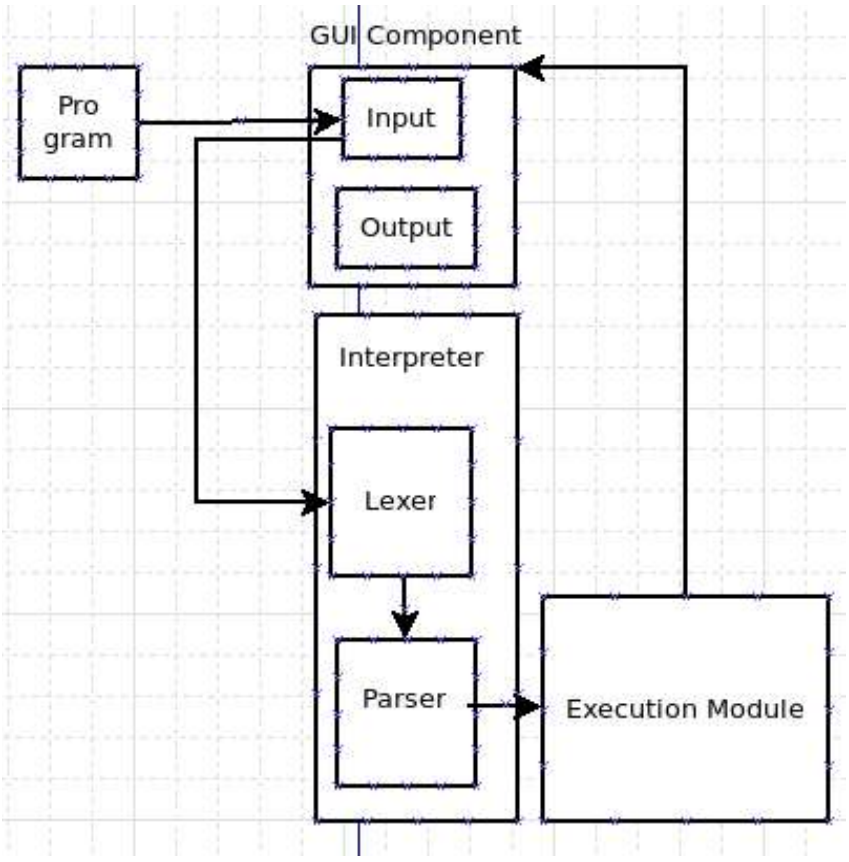


Fig 1. *Execution Flow*

5.1 Interpreter modules

5.1.1. Interpretation Data Structures And Algorithm

There are three Data Structures that are used in the Interpretation Algorithms.
These data structures are :-

1. **Program:** This is a variable that initially contains the Tempura program i.e the expression tree which is obtained by parsing. After the execution of each state, it is transformed to a formula of the form weak-next w, where w describes what should be done in the next state.
2. **Memory:** This is the storage unit which is a list having a separate entry for each variable in the Tempura program. Each entry is a pair with the name of the associated variable and the value of that variable I in that state, analogous to a symbol table.
   Eg < <I,0>, <J,2> >.
3. **Terminate_flag:** This is a boolean variable, which is initialized to false value. The Tempura program places either true or false in it during every state, thus indicating whether or not that state is the final

state. At the end of executing a state the tempura program checks whether this flag is true or false, if its is true then it stops execution else it continues execution of the next state. For example, in a state where the statement empty is encountered, the interpreter puts the value true in the done-flag cell. If the statement more is encountered, the value false is used instead.
Eg O(O(empty)) ∧ (I = 0) ∧ (I gets I + 1) ∧ 2(J = 2 · I)

Before state 0:
Program: O(O(empty)) ∧ (I = 0) ∧ (I gets I + 1) ∧ 2(J = 2 · I),
Memory: < <Terminate_flag , false> >.

After state 0:
Program: weaknext [(O empty) ∧ (I = 1) ∧ (I gets I + 1) ∧ 2(J = 2 · I)],
Memory: < <I,0> , <J,0 >, <Terminate_flag , false> > .

Before state 1:
Program: (O empty) ∧ (I = 1) ∧ (I gets I + 1) ∧ 2(J = 2 · I),
Memory: < <I,0> , <J,0 >, <Terminate_flag , false> > .

After state 1:
Program: weaknext [empty ∧ (I = 2) ∧ (I gets I + 1) ∧ 2(J = 2 · I)],
Memory: < <I,1> ,<I, 2> , <Terminate_flag , false> > .

Before state 2:
Program: empty ∧ (I = 2) ∧ (I gets I + 1) ∧ 2(J = 2 · I),
Memory: < <I,1> ,<I, 2> , <Terminate_flag , false> > .

After state 2:
Program: weaknext [false ∧ (I gets I + 1) ∧ 2(J = 2 · I)],
Memory: < <I,2> , <J,4> , <Terminate_flag , true> > .

Top Level Algorithms :-

```
Begin
prepare_execution_of_program;
loop
   execute_single_state
   exit when Terminate_Flag = true
   otherwise
   advance_to_next_state
end.
```

Explanation:

The algorithm begins and executes the “prepare_execution_of_program” module. This module assigns the Program variable the expression tree that is returned by the parser (details of expression tree is discussed later).

Next the loop is started. As execution begins the loop it executes the “execute_single_state” module where the expression tree is executed, the identifiers whose value changes in this state are changed in the symbol table or the memory is updated and the Program's associated weak next form is generated which is going to be executed in the next state if there exists any.

After executing the “execute_single_state” module it checks the Terminate_Flag. If its value remains false then its executes “advance_to_next_state” module where the Program variable is assigned the weak next of the Program obtained in this state and the loop is executed again. If the value of the Terminate_Flag changes

to true during “execute_single_state” which means this is the last state and further execution will not be done then it bypasses the “advance_to_next_state” module and ends.

5.2 Simulator Module (Module between Interpreter and GUI)

Here algorithm for simulation is done. In the algorithm how the interpreter and the simulator are co-ordinating is discussed.

5.2.1. Simulation Data Structures and Algorithm

program is assigned the expression tree of the input code

```
if termination condition is not set in program

        ERROR

else

    Loop till termination condition met

        {present_state variable values, what_remains}:
            ExecutionRoutine(program);

        PaintRoutine(present_state variable values);

        Program is assigned with what_remains

    end
```

Chapter 6
Simulator: The Theory behind Implementation

6.1. Lexer Construction

This sub-section discusses the steps required behind the construction of the lexer which is the first component of the interpreter. In particular, the theory behind lexer generators and the lexer generator used in the project, jflex is discussed. The lexer implementation in the project is then discussed.

6.1.1 Brief Theory Of Lexer Generators And Jflex

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens, i.e. meaningful character strings. A program or function that performs lexical analysis is called a lexical analyzer, lexer, tokenizer or scanner. The specification of a programming language often includes a set of rules, the lexical grammar, which defines the lexical syntax. The lexical syntax is usually a regular language, with the grammar rules consisting of regular expressions; they define the set of possible character sequences that are used to form individual tokens.

Lexers are often generated by a lexer generator. These generators are a form of domain-specific language, taking in a lexical specification – generally regular expressions with some markup – and outputting a lexer. These tools yield very fast development, which is particularly important in early development, both to get a working lexer and because the language specification may be changing frequently. Further, they often provide advanced features, such as pre- and post-conditions which are hard to program by hand.

JFlex is a lexical analyzer generator (also known as scanner generator) for Java, written in Java.

The features of Jflex are:

- Predefined character classes
- Fast generated scanners
- Better platform independence

6.1.2 Lexer Implementation In Project

The following is a code snippet of the lexer specification:

```
/*building up a string constant */
\'                      {
                          stringConstant=new StringBuffer();
                          yybegin(SS);
                        }
/* state SS */
<SS> {
     [^\'\n]*           { stringConstant.append(yytext());}
     \'                 {
                          yyparser.yylval=new ParserVal(new StringConstantNode(stringConstant.toString()));
                          yybegin(YYINITIAL);
                          return Parser.STRING;
                        }
     <<EOF>>             {yybegin(YYINITIAL); return Parser.END;  }
     }

/* temporal operator */
"always"                  { return Parser.ALWAYS; }

/* arithmetic operator */
"+"                       { return Parser.PLUS ;  }

/* logical operator */
"not"                     { return Parser.NOT;    }
```

Here is an explanation of the above code snippet:

'\' denotes on encountering an open quote (the \ is needed as an escape character), a new StringBuffer object (a string object that is optimized to be resized in Java) is created and the lexer's state is changed to SS. In the state SS on encountering a character other than a closing quote and a new line character, it is to be appended to the StringBuffer object and is represented by the regular expression [^\\'\n]*, the ^ indicating complement of the language with characters end quote and new line. The * indicating repetation 0 or more times. On encountering a closing quote, the string value of the StringBuffer object is passed to the StringConstantNode's constructor and this is the value of the token that is returned by the lexer. The lexer goes to the initial state. A string token is returned by the lexer. On encountering the end of the file, the lexer goes to its initial state and returns an end token. Other regular expressions are self explanatory, when a string of characters a,l,w,a,y,s is encountered, the always token is returned. Similarly when the + character is seen by the lexer, the plus token is returned by the lexer.

6.2 Parser Construction

This sub-section discusses the steps required behind the construction of the parser which in addition to the input file takes in the tokens returned by the lexer. In particular, the theory behind parser generators and the parser generator used in the project, byaccj is discussed. The parser implementation in the project is then discussed.

6.2.1 Brief Theory Of Parser Generators And Byaccj

Parsing or syntactic analysis is the process of analysing a string of symbols in computer languages, according to the rules of a formal grammar. A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process. The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyzer, which creates tokens from the sequence of input characters and these tokens are sent to the parser where they are the terminal symbols of the grammar rules.

A compiler-compiler or compiler generator is a tool that creates a parser from some form of formal description of a language and machine. The earliest and still most common form of compiler-compiler is a **parser generator**, whose input is a grammar (usually in BNF) of a programming language, and whose generated output is the source code of a parser often used as a component of a compiler.

BYACC/J is an extension of the Berkeley v 1.8 YACC-compatible parser generator. A “-J” flag is added which will cause Byacc to generate Java source code, which if compiled properly will produce a LALR grammar parser.

Features of Byacc/J:

- BYACC/J is coded in C, so the generation of Java code is extremely fast.
- The resulting byte code is small -- starting at about 11 kbytes.
- Syntax is similar to classic YACC grammars, only the action routines need to be in Java.
- No additional runtime libraries are required. The generated source code is the entire parser.

6.2.1 Parser Implementation In Project

The following is a code snippet of the parser specification:

```
%%
/* Rules start here */
program:
non_empty_expression_end_list {$$=$1; RootNodeOfExpressionTree=((NodeType)$$);}
;

non_empty_expression_end_list:
program expression END    {$$=new BasicNode(new AndOperator(),(NodeType)$1,(NodeType)$2);}
| expression END  {$$=$1;};

/* EXPRESSIONS */
expression:
  LBRACE expression RBRACE {$$=$2;}
| LPAREN expression RPAREN {$$=$2;}

/* BOOLEAN OPERATORS */
| expression AND expression {$$=new BasicNode(new AndOperator(),(NodeType)$1,(NodeType)$3);   }

/* TEMPORAL OPERATORS */
| expression GETS expression {$$=new BasicNode(new GetsOperator(),(NodeType)$1,(NodeType)$3);  }
| ALWAYS expression { $$=new BasicNode(new AlwaysOperator(),(NodeType)$2); }
| HALT expression { $$=new BasicNode(new HaltOperator(),(NodeType)$2); }
| LEN expression { $$=new BasicNode(new LenOperator(),(NodeType)$2); }
```

Here is an explanation of the above code snippet:

The action routines that are executed once the rules are satisfied construct the nodes of the expression tree that is constructed as the parsing goes on. The grammar rules are specified in BNF. The action routines are contained within the braces. The upper case strings in the grammar rules are the terminal symbols that are returned by the lexer. The LHS of the first production rule contains the root of the generated expression tree and hence the corresponding action routine assigns the value of the RHS (contained in $$) to the variable RootNodeOfExpressionTree. The second production rule creates a conjunction of all expressions. The remaining production rules shown, define the grammar for the various types of expressions, an expression within matching brackets, a boolean expression for and operator and expressions for the temporal operators: gets, always, halt and len.

6.3 Data Structure For Program Simulation

This sub-section discusses the data structure that is used to represent the tempura program. In particular, the version of the Expression Tree data structure used in the project is discussed and also why it is suitable for simulation purposes.

6.3.1 Expression Tree

In general, expression trees are special kinds of binary trees. The leaves of a binary expression tree are operands, such as constants or variable names, and the other nodes contain operators. However, in the expression tree data structure used here each node can have at-most five operands. The leaves here are objects that represent the various types of constants supported by the language, identifier nodes, whose values are determined by look up from the symbol table and the various types of operands that are supported by Tempura language. The non-leaf nodes are objects that represent expressions which have expressions or leaves as its children.

6.3.2 Expression Tree and Simulation

The representation of the Tempura program using the expression tree data structure as is done in the project gives the following advantages:

1. The expression tree can be constructed during the parsing stage itself. Hence, the construction of the data structure does not require an extra pass over the Tempura program.

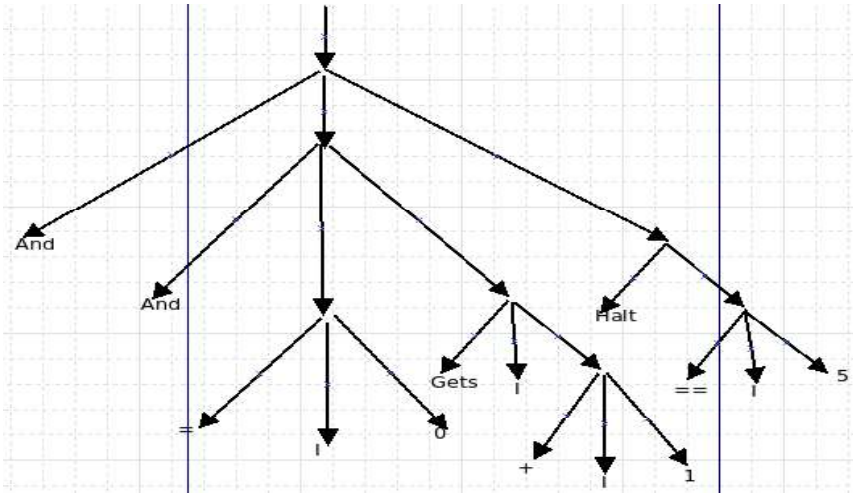
2. Since a Tempura program needs to be converted to its weak next form, a data structure that can be easily modified will be suitable. Here the expression tree is suitable since a sub-expression tree that represents a single Tempura formula can be independently modified into its weak next form and hence the whole program can be converted to its weak next form.

Here is an example:

Tempura program:

```
/* run */define test_2() =
{ exists I:
  {
     I=0 and I gets I+1 and halt(I==5)
  }
}.
```

The pictorial representation of the expression tree:



The expression tree expressed in XML:

```
−<CompoundNode>
    <DefineOperatornode> </DefineOperatornode>
    <Identifiernode> test_2</Identifiernode>
  −<CompoundNode>
      <LambdaOperatornode> </LambdaOperatornode>
    −<CompoundNode>
        <ExistsOperatornode> </ExistsOperatornode>
        <Identifiernode> I</Identifiernode>
      −<CompoundNode>
          <Andoperatornode> </Andoperatornode>
        −<CompoundNode>
            <Andoperatornode> </Andoperatornode>
          +<CompoundNode></CompoundNode>
          +<CompoundNode></CompoundNode>
          </CompoundNode>
        −<CompoundNode>
            <HaltOperatornode> </HaltOperatornode>
          −<CompoundNode>
              <IsEqualTooperatornode> </IsEqualTooperatornode>
              <Identifiernode> I</Identifiernode>
              <Intconstantnode> 5</Intconstantnode>
            </CompoundNode>
          </CompoundNode>
        </CompoundNode>
      </CompoundNode>
    </CompoundNode>
  </CompoundNode>
```

6.4. Algorithm for Tempura Operators on Expression Trees:

This section discusses the algorithms for the operators on the expression trees:

6.4.1 Generic Algorithm of Operators:

The execution of the Operators will follow a generic algorithm. Different operator will have small operator specific changes but the overall algorithm will be same.
The diagram below shows pictorial description of the generic algorithm

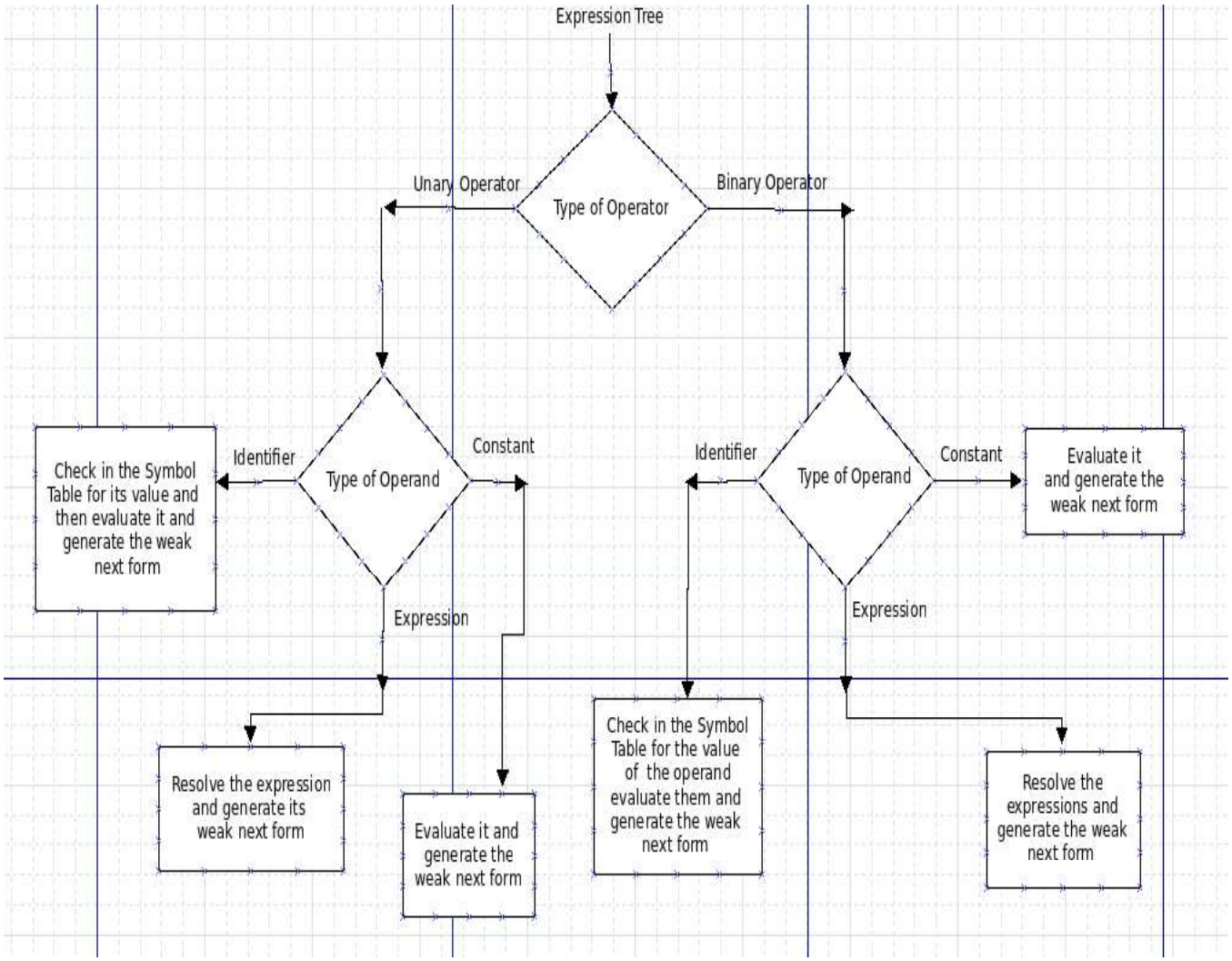


Fig 2. *Flow of Generic Algorithm*

Input to it is an expression tree, first step is to decide whether the operator is a binary operator or a unary operator, this task is done by the 1st decision statement. In the next step what type of operand the operator will operate is checked, it might be an identifier, or an expression or a constant (which might be an integer or a boolean constant). Next let it be considered that the operator is an unary operator, if its operand is an identifier then the symbol table of the previous state is used to access its value and evaluate it at the same time generate its associated weaknext form, if the identifier does not have a value then appropriate error message will be generated. If the operand is an constant the directly evaluation is possible and its weaknext

forms are generated. For the operand being an expression it is further resolved into simpler form containing operators whose operands are either constants or identifiers and thereafter its weaknext forms are obtained. For binary operator if the operands are constants or identifiers then evaluation will be done easily by the previous stated method but if the operands are expressions then the same thing will be done twice once for each operator i.e once for the 1st operand all the above stated operations are carried out and the same task with the 2nd operand are performed.

6.4.2 Operator Specific Algorithm:

The operators that are used in Tempura can be broadly classified into two categories, one is the non-temporal operator and other are the temporal operators. In this section, how these operators are implemented in details are discussed.

Non-Temporal Operators:

In Tempura there are many non-temporal operators which includes logical operators like and, or, etc, arithmetical operators like +,-,*,/ and assignment operators like = ,etc.
As stated earlier all the operators will be following the generic algorithm, with slight modification. Here is the description of the algorithm for the and operator.

The and operator can have as its operand either an expression or an identifier or a constant or combination of these. The algorithm takes as its arguments the two operands of the and operator. Then it checks whether the 1st operand is an expression or not, if it is an expression then it is resolved further. Similarly the 2nd operand is also tested. If the operands are identifiers then by accessing the symbol table their respective values can be obtained and if they are constants then their values are already known. After evaluation is done the weaknext part of the and operator and its operands will be evaluated.

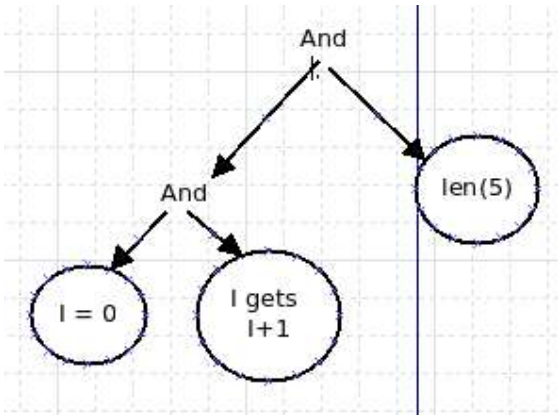


Fig.3 *Expression Tree for the 'And' operator as per the eg. program*

Eg. Program: I=0 and I gets I+1 and len(5).
Fig 1. gives the pictorial view of the expression tree of the above code snippet. For the and which is at the right most end the part to its LHS is an expression which has to resolved further for evaluation and the RHS is an operator which requires no further resolving, it just declares that there are five intervals or six states in this tempura program. The expression of the LHS of the left most and can be resolved further into an assignment statement i.e I's value is 0 in this state and a temporal operator gets.

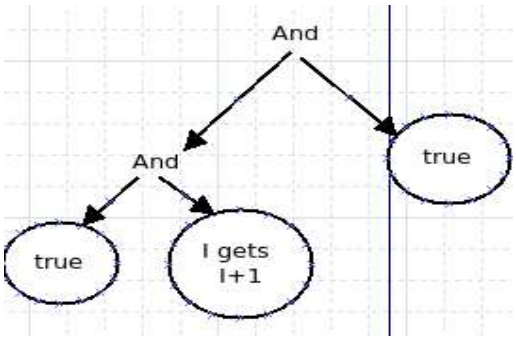


Fig 4. *the weak next form of the program*

Fig 2 shows us the pictorial description of the weak next form of this code snippet that will be executed in the next state. The assignment i.e I=0 is replaced by true in order to satisfy the and logic and the assignment will be done in the 1st state only. The weak next part of gets is same as that in the 1st state. While the len operator will be decremented by 1 as the no. of states after this state will be one less than that in this state. The other non-temporal operators will follow similarly with slight changes, based on their logic which are quite intuitive.

Temporal Operator:

In this section we will discuss some of the most frequently used temporal operators.

1. **Always:** As from the previous discussion it is known that the expression within the always operator is executed in each state.
   So, for an expression like
   Program: I=0 and I gets I+1 and always(J=2.I) and halt(J==4).

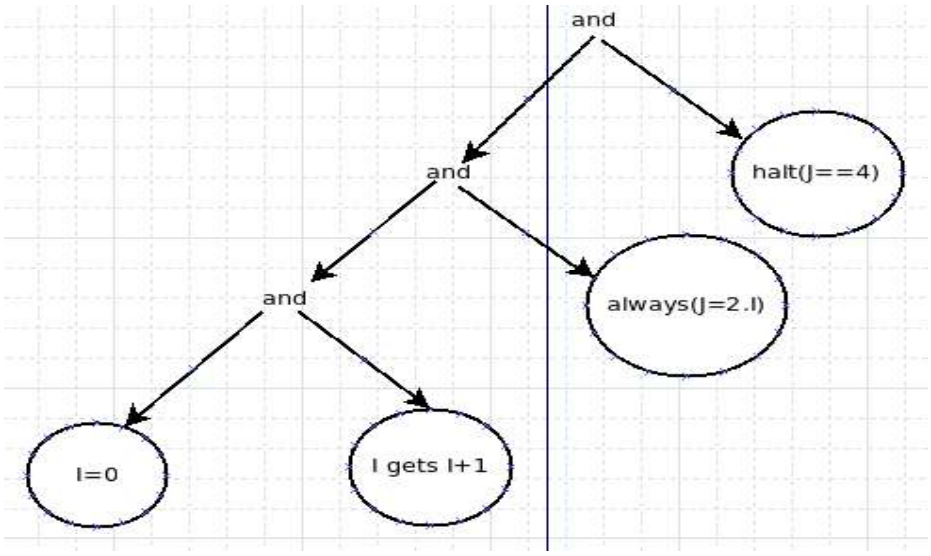


Fig.3 *the expression tree of the code*

Fig. 3 shows us the expression tree of the code snippet .

Here also the and in the right most part of the expression is checked first. The part to the left of and is an expression and the part to the right is an operator which is halt that is checked at each state to determine the value of Terminate_Flag variable. The expression is further resolved i.e I=0 and I gets I+1 and always(J=2.I) . This time the right most and operator will be considered again and resolve the expression into LHS which is an expression and RHS is the always operator. The part within the always operator will be evaluate and symbol table will be updated. Again similar process will be carried out but this time the operands are both operators.

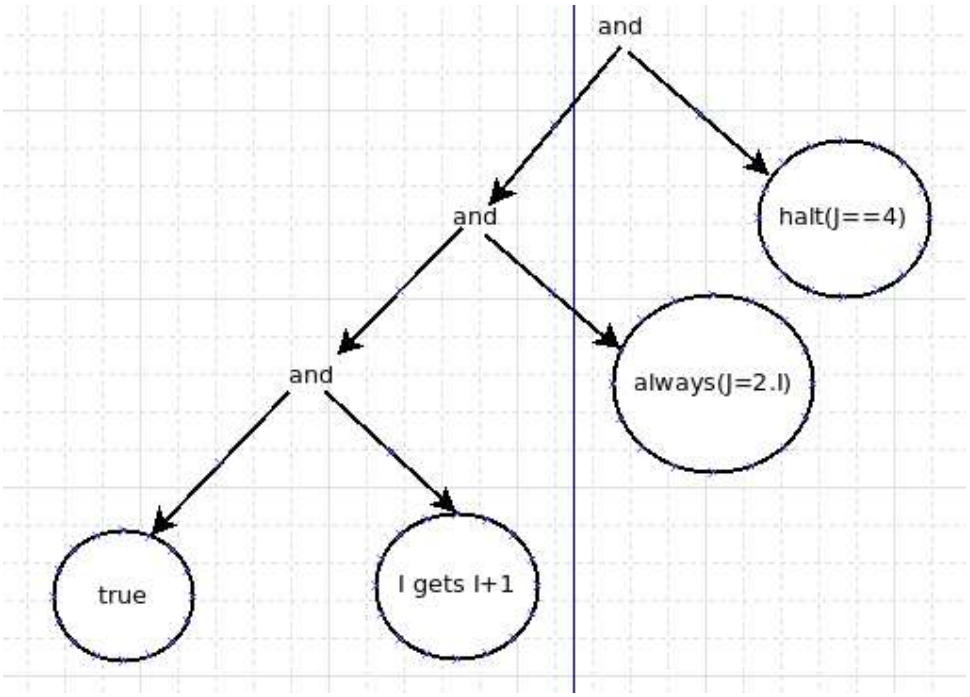


Fig 4. w*eak next form of the example Program*

Above figure shows the weak next form of the Program that will be executed in the upcoming states. The assignment of I=0 will be replaced by true, while all the others operators will remain same.

2. **Gets:**
The gets operator can be written written in a simplified form in terms of next and always operator.
I gets J is equivalent to always{if more then (next I) = J}.
So the gets operator's each occurrence is replaced by the above form. 'more' is a keyword which means that this is not the last state there are states after this. So, it is resolved into simpler form as discussed in the previous cases and evaluation and generation of weak next form is done.

6.5 Implementation Details

6.5.1 Software Requirements Specification

A simulator is basically software with certain features available in it. In this section, all general and unique aspects of the simulator will be described briefly.

The GUI of the whole simulator has been divided into three sections.

1. Input Section:
A user can provide the input tempura program in the following two ways:

1.1. File Browsing Option
A file browsing option has been provided so that each time user click the “Browse” button, he is prompted with a file chooser. User will select a valid Tempura (“.t” file) through a file chooser and after selecting the file the content of the file will be displayed in the text-area section in this section.

1.2. Text Area
A text area portion has also been provided. User can write down a tempura program in that text-area. Also on choosing valid tempura file from file chooser described above, the tempura program written in the file will be displayed in this section.

1.3. Run Button
A “Run” button has been provided. After providing the code, user needs to click on the “Run” Button and on clicking it, code will be sent to the Simulator as an input and Simulator will start working.

2. Output Section
An output section has been provided so that actual visual output can be displayed. It has mainly two sections.

2.1. Grid
A grid has been given where a graphical output will be displayed. It will show a graph which will display the magnitude of the variables used in a tempura program with respect to state. State is represnted by x-axis and magnitude is displayed by y-axis.

2.2. Text-area
A text-area has been provided below the grid where textual output or any error in the input tempura program will be displayed.

3. Parameter Input Section
In this section, a text-field and a “continue” button has been provided. If there is any parameter required for running a tempura program, it can be provided here. On clicking the “continue” button, the parameter value will be fed to the interpreter.

6.5.2 Data Flow Diagrams

Following are the data flow diagrams of the project mainly showing the important modules.

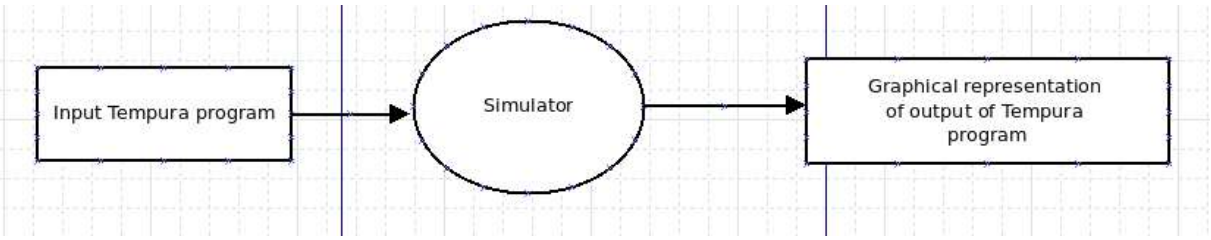


Fig 5. *Level 0 DFD*

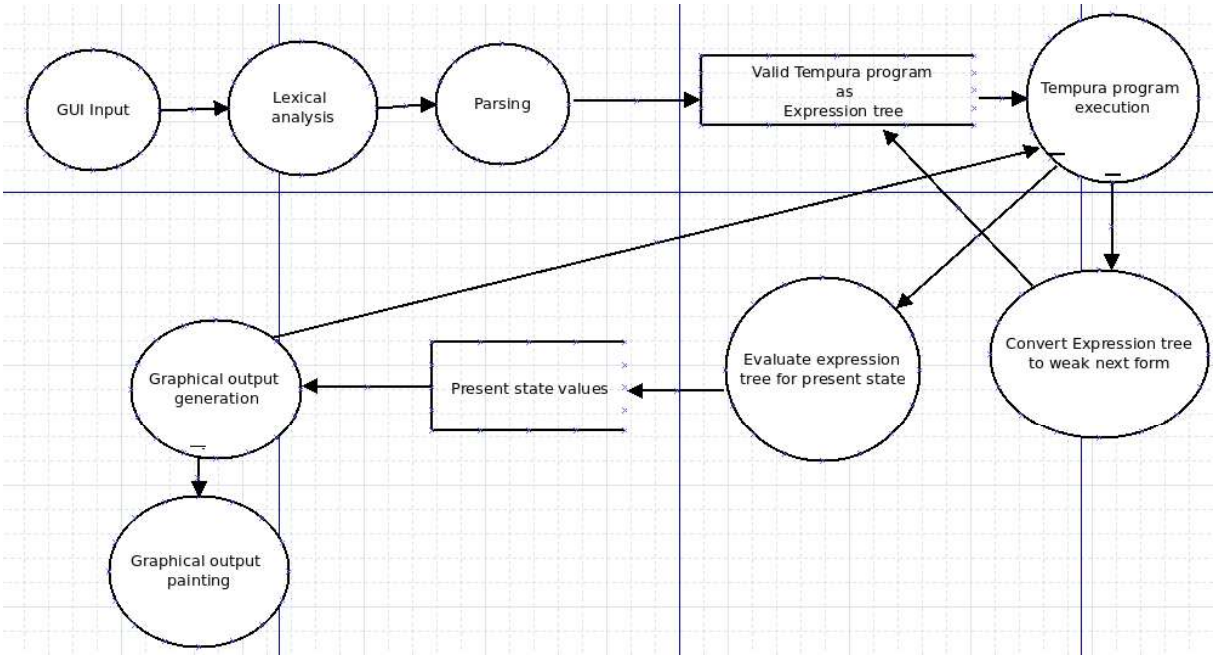
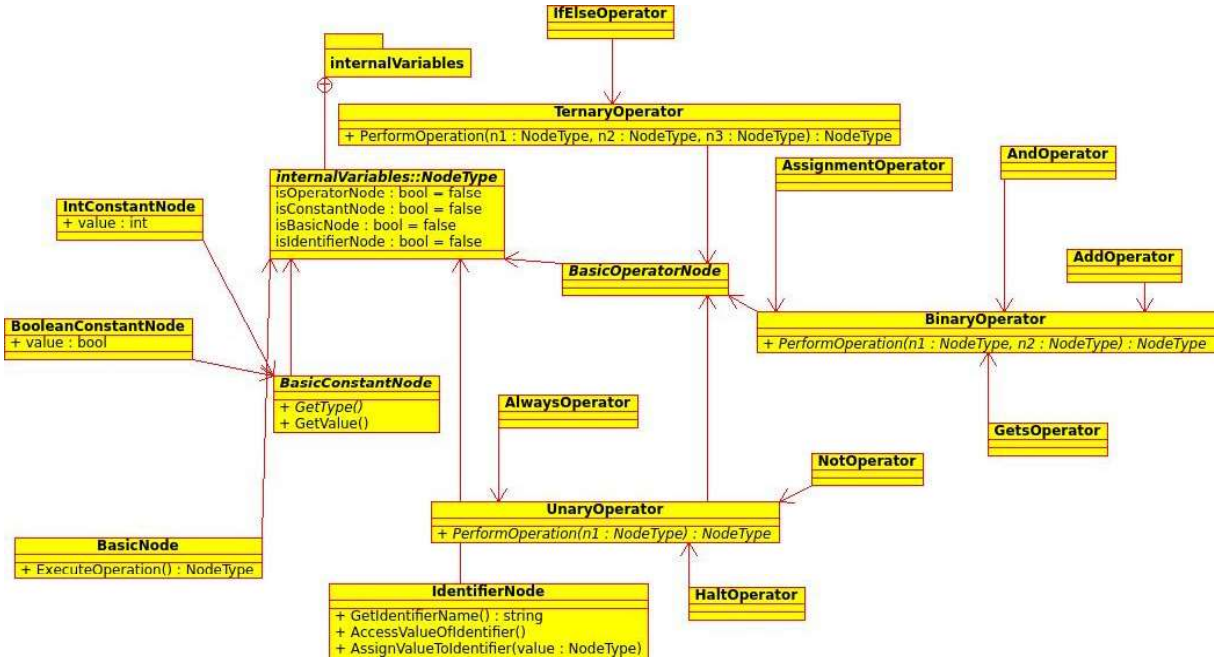


Fig 6. *Level 1 DFD*

6.5.3 Class Diagrams

The following is the class diagram for the nodes of the expression tree which is built up in the action routines of the parser.



Explanation:

InternalVariables package
This package contains the classes that make up the nodes of the expression tree, that is built up in the action routines of the parser.

1. **NodeType class**
The base class of all the classes is the abstract class NodeType. The declaration of this class is as follows:

```
public abstract class NodeType {

   public boolean isOperator;
   public boolean isConstant;
   public boolean isLanguageConstant;
   public boolean isBasicNode;
   public boolean isIdentifierNode;

   public NodeType() {
      isOperator=false;
      isConstant=false;
      isBasicNode=false;
      isIdentifierNode=false;
      isLanguageConstant=false;
   }

    public abstract NodeType ExecuteOperation();

}
```

As can be seen the data members denote whether the node is an operator, a constant, a Tempura language constant, a BasicNode(a compound node) or an identifier node. The class also contains some abstract methods that are implemented in the sub classes. The ExecuteOperation method is used to transform the node to its weak next form and get the present state values.

2. BasicConstantNode class
This class represents a basic constant base class from which the various constants are sub-classed. The declaration of the class is as follows:

```
public abstract class BasicConstant extends NodeType
{
   protected Type constantType;

   public BasicConstant() {
      super.isConstant=true;
   }

   public Type GetConstantType()
   {
      return this.constantType;
   }

   @Override
   public NodeType ExecuteOperation()
   {
      return this;
   }
```

```
    public abstract Object GetValue();
}
```
Type is an enumeration that contains the various types of constants like INT, BOOL, etc. The methods are self-explanatory, the abstract GetValue() method returns the value of the constant node.

2.1. IntConstantNode class

This class represents an integer constant. It inherits from the BasicConstantNode class. It has over-ridden method to get the integer value stored and the constructor is used to assign the value to its integer data member. The declaration of the class is as follows:

```
public class IntConstantNode extends BasicConstant {
    public int value;

    public IntConstantNode(int value) {
        super.constantType=Type.INT;
        this.value=value;
    }

    @Override
    public Integer GetValue()
    {
        return this.value;
    }
}
```

3. BasicOperatorNode class

This class represents an operator node. The declaration of the class is as shown:

```
public abstract class BasicOperator extends NodeType{
protected int operands;

 public BasicOperator() {
        super.isOperator=true;
    }
}
```

It contains a protected data member which denotes the number of operand the operator takes.

3.1. UnaryOperator class

This class represents an unary operator. The declaration of the class is as shown:

```
public abstract class UnaryOperator extends BasicOperator{

    public UnaryOperator() {
        super.operands=1;
    }

    public abstract NodeType PerformOperation(NodeType n1);

}
```

The constructor sets the number of operands to 1. There is an abstract PerformOperation method that takes in a single argument (the operand) and returns the weak next form after assigning the values in the present

state.

3.1.1. AlwaysOperator class

This class represent the unary temporal operator Always. The class is as follows:

```
public class AlwaysOperator extends UnaryOperator
{
 @Override
 public NodeType PerformOperation(NodeType n1)
 {
    if(n1.isBasicNode)
    {
       ((BasicNode)n1).ExecuteOperation();
    }
    if(n1.isConstant)
    {
      return new BooleanConstantNode(false);
    }
    return new BasicNode(new AlwaysOperator(), n1);
 }
```

The only operation in the class PerformOperation takes in a node and transforms it to its weak next form by applying the always temporal operator. The code in the function explains the transformation for the various cases.

3.2. BinaryOperator class

This class represents an binary operator. The declaration of the class is as shown:

```
public abstract class BinaryOperator extends BasicOperator
{
   public BinaryOperator() {
       super.operands=2;
   }

 protected abstract NodeType PerformOperation(NodeType n1, NodeType n2);
}
```

The constructor sets the number of operands to 2. There is an abstract PerformOperation method that takes in two arguments (the operands) and returns the weak next form after assigning the values in the present state.

3.2.1. GetsOperator class

This class represents the temporal binary operator gets. The class declaration is as follows:

```
public class GetsOperator extends BinaryOperator
{
 @Override
 public NodeType PerformOperation(NodeType n1, NodeType n2)
 {
    return new BasicNode(new AlwaysOperator(), new BasicNode(new IassignOperator(), n1,n2));
 }
}
```

The only operation in the class PerformOperation takes in two nodes and returns the weak next form after applying the always temporal operator. The code in the function explains the conversion to the weak next form.

3.2.2. AndOperator class

This class represents the binary operator and. The class declaration is as follows:

```
public class AndOperator extends BinaryOperator
{
 @Override
   public NodeType PerformOperation(NodeType n1, NodeType n2)
   {

      if(n1.isBasicNode || n1.isLanguageConstant)
      {
         n1 = n1.ExecuteOperation();
      }
      if(n2.isBasicNode || n2.isLanguageConstant)
      {
         n2 = n2.ExecuteOperation();
      }

      BooleanConstantNode value1=null,value2=null;
      if(n1.isConstant)
      {
         if(((BasicConstant)n1).GetConstantType() == Type.BOOL)
         {
            value1 = (BooleanConstantNode)n1;
         }
      }
      if(n1.isIdentifierNode)
      {
         BasicConstant basicConstant =
(BasicConstant)SymbolTable.AccessValueOfVariable(((IdentifierNode)n1).GetIdentifierName());
         if(basicConstant.GetConstantType() == Type.BOOL)
         {
            value1 = (BooleanConstantNode)basicConstant;
         }
      }

      if(n2.isConstant)
      {
         if(((BasicConstant)n2).GetConstantType() == Type.BOOL)
         {
            value2 = (BooleanConstantNode)n2;
         }
      }
      if(n2.isIdentifierNode)
      {
         BasicConstant basicConstant =
(BasicConstant)SymbolTable.AccessValueOfVariable(((IdentifierNode)n2).GetIdentifierName());

         if(basicConstant.GetConstantType() == Type.BOOL)
         {
            value2 = (BooleanConstantNode)basicConstant;
         }
      }

      if(value1==null && value2==null)
      {
```

```
            return new BasicNode(new AndOperator(), n1, n2);
        }
        else if(value1==null)
        {
            if(value2.GetValue() == true)
            {
                return n1;
            }
            else
            {
                return new BooleanConstantNode(false);
            }
        }
        else if(value2 == null)
        {
            if(value1.GetValue() == true)
            {
                return n2;
            }
            else
            {
                return new BooleanConstantNode(false);
            }
        }
        //both are boolean constants
        else
        {
            return new BooleanConstantNode(value1.GetValue() && value2.GetValue());
        }
    }
}
```

The only operation in the class PerformOperation takes in two nodes and returns the weak next form after applying the and operator. The code in the function explains the conversion to the weak next form and the assignment of values to the identifiers in the present state.

3.3. TernaryOperator class

This class represents the ternary operator. The declaration of the class is as follows:

```
public abstract class TernaryOperator extends BasicOperator{

    public TernaryOperator() {
        super.operands=3;
    }

    protected abstract NodeType PerformOperation(NodeType n1, NodeType n2,NodeType n3);
}
```

The constructor sets the number of operands to 3. There is an abstract PerformOperation method that takes in three arguments (the operands) and returns the weak next form after assigning the values in the present state.

3.3.1. IfOperator class

This class represents the if else operator. The declaration of the class is as follows:

```
public class IfOperator extends TernaryOperator {
```

```
    @Override
    public NodeType PerformOperation(NodeType n1, NodeType n2,NodeType n3)
    {
        try
        {
            BooleanConstantNode booleanConstant = (BooleanConstantNode)n1.ExecuteOperation();
            if(booleanConstant.GetValue() == true)
            {
              n2.ExecuteOperation();
            }
            else
            {
               n3.ExecuteOperation();
            }
            return new BooleanConstantNode(true);
        }
        catch(ClassCastException cce)
        {
            Errors.OperandTypeIncorrect.PrintErrorMessage();
            return null;
        }
    }
}
```

The only operation in the class PerformOperation takes in three nodes and returns the weak next form after applying the if-else operator. The code in the function explains the working of the operator.

4. IdentifierNode class

This class represents an identifier. The class contains the identifier name and methods to get and set the value of the identifier in the global symbol tables. The declaration of the class is as follows:

```
public class IdentifierNode extends NodeType{

    private final String identifierName;

    public IdentifierNode(String identifierName)
    {
        super.isIdentifierNode=true;
        this.identifierName=identifierName;
    }

    public NodeType AccessValueOfIdentifier()
    {
        return SymbolTable.AccessValueOfVariable(this.identifierName);
    }

    public void InstallIdentifier()
    {
        SymbolTable.InstallVariable(this.identifierName);
    }

    public void AssignValueToIdentifier(NodeType value)
    {
        SymbolTable.AssignValueToVariable(this.identifierName, value);
    }
```

```
    @Override
    public NodeType ExecuteOperation()
    {
        //if this variable has value assigned to it then return the value
        return AccessValueOfIdentifier();
    }
   }
```

The data member contains the name of the identifier. There are public methods to access the values of the identifier. As can be seen from the method signature and method body, the identifier name and values are sent to the static SymbolTable class, which insert these in the appropriate symbol table.

5. BasicNode class

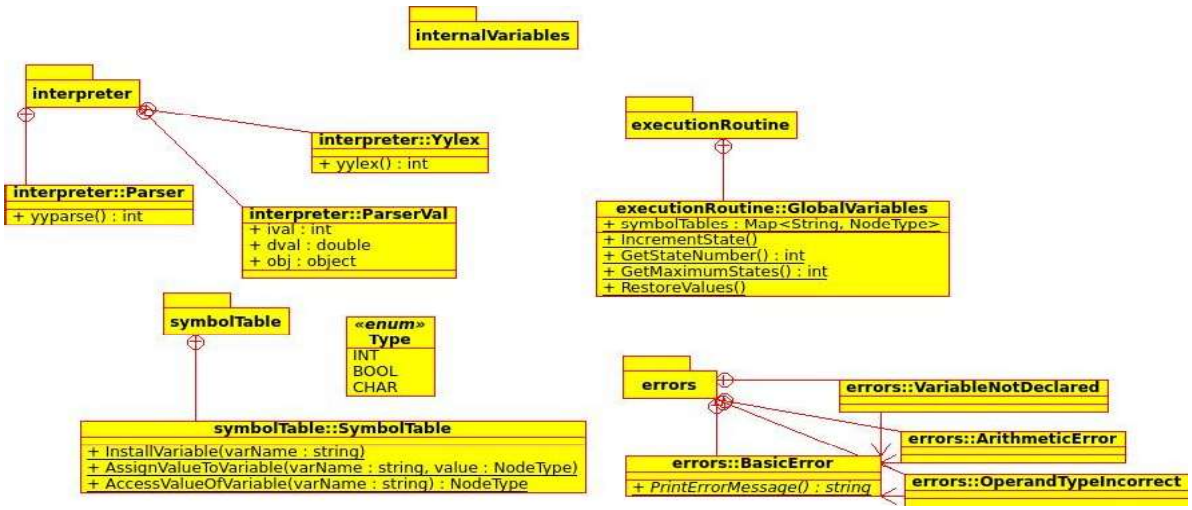
This class represents a compound node, a node that holds an operator node and at most 3 other operands which can be constant nodes, identifier nodes or other compound nodes. The declaration of the class is as follows:

```
public class BasicNode extends NodeType
{
    public int childCount;
    public NodeType [] childNodes;

    //make  super.isBasicNode=true in each of the constructors
    public BasicNode(NodeType n0,NodeType n1)
    {
    }

    public BasicNode(NodeType n0, NodeType n1, NodeType n2)
    {
    }

    public BasicNode(NodeType n0, NodeType n1, NodeType n2,NodeType n3)
    {
    }



    //this method will fire an appropriate operation based on the operator and return a NodeType
    @Override
    public NodeType ExecuteOperation()
    {
        //if the first child is an operator
        if(this.childNodes[0].isOperator==true)
        {
            BasicOperator basicOperator=(BasicOperator)this.childNodes[0];
            if(basicOperator.operands==1)
            {
                UnaryOperator unaryOperator=(UnaryOperator)childNodes[0];
                return unaryOperator.PerformOperation(childNodes[1]);
            }
            else if(basicOperator.operands==2)
            {
                BinaryOperator binaryOperator=(BinaryOperator)childNodes[0];
                return binaryOperator.PerformOperation(childNodes[1], childNodes[2]);
```

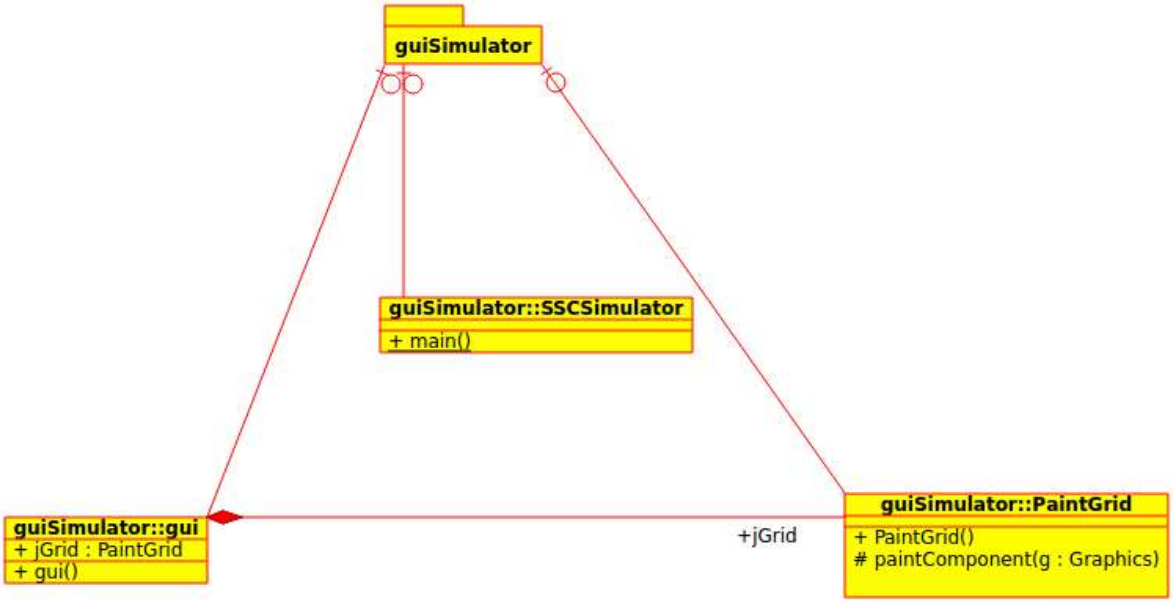
```
        }
        else if(basicOperator.operands==3)
        {
            TernaryOperator ternaryOperator=(TernaryOperator)childNodes[0];
            return ternaryOperator.PerformOperation(childNodes[1], childNodes[2], childNodes[3]);
        }
    }
    //if not an operator
    else
    {
        return this.childNodes[0].ExecuteOperation();
    }
    return null;
}
```

The data members contain a count of the number of children of the compound node and an array of that size that holds these children. The constructors accept varying number of arguments based on the number of children of the node. The method definition of the over-ridden method ExecuteOperation() is also shown that returns the transformed form of the node (its weak next form). This method works by checking whether the first child is an operator. If it is then based on the number of arguments accepted by the operator, the PerformOperation method of that operator is called with the arguments being the other elements in the array.

The following is the class diagram for the actual execution part of the interpreter.



Following is the class diagram of the Graphical User Interface portion.



6.5.4 Technologies Used

This chapter lists all the technologies that will be used for designing the simulator. Various technologies will be used for building various components. The features of the tools that are important for this purpose are also specified in detail.

1. Java
Java is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another.

Technical features of Java language that will be used:

- **Object-oriented:** Java provides all the features of object oriented programming including encapsulation, polymorphism, inheritance and data hiding.

- **Portable and Architecture-Neutral:** Java programs are capable of running on multiple environments and in addition it runs the same regardless of the environment.

Jflex has been used as the lexical generator which generates the lexer in Java and BYACC/J has been used as the parser generator which generates the parser in Java.

2. Java Swing
Swing is the primary Java GUI widget toolkit. It is an API for providing a Graphical User Interface(GUI) for Java programs. Swing provides a more sophisticated set of GUI components than the earlier Abstract Window Toolkit(AWT). Unlike AWT components, Swing components are not implemented by platform-specific code. Instead they are written entirely in Java and therefore are platform-independent. The term "lightweight" is used to describe such an element. It will be used to generate the front end for the

simulator. This includes standard user input interaction buttons as well as UI where the simulated output will be shown.

Technical features of Swing that will be used:

- **Loosely coupled and MVC:** The Swing library makes heavy use of the Model/View/Controller software design pattern which conceptually decouples the data being viewed from the user interface controls through which it is viewed.

- **Lightweight UI:** Swing's high level of flexibility is reflected in its inherent ability to override the native host Operating System's GUI controls for displaying itself. Thus, a Swing component does not have a corresponding native OS GUI component, and is free to render itself in any way that is possible with the underlying graphics GUIs.

3. Netbeans IDE 7.3

Netbeans is an Integrated Development Environment for developing primarily with Java. It is also an application platform for Java Desktop applications(Swing based) and others.

Technical features of Netbeans that will be used:

- GUI design tool: The GUI design-tool enables developers to prototype and design Swing GUIs by dragging and positioning GUI components.

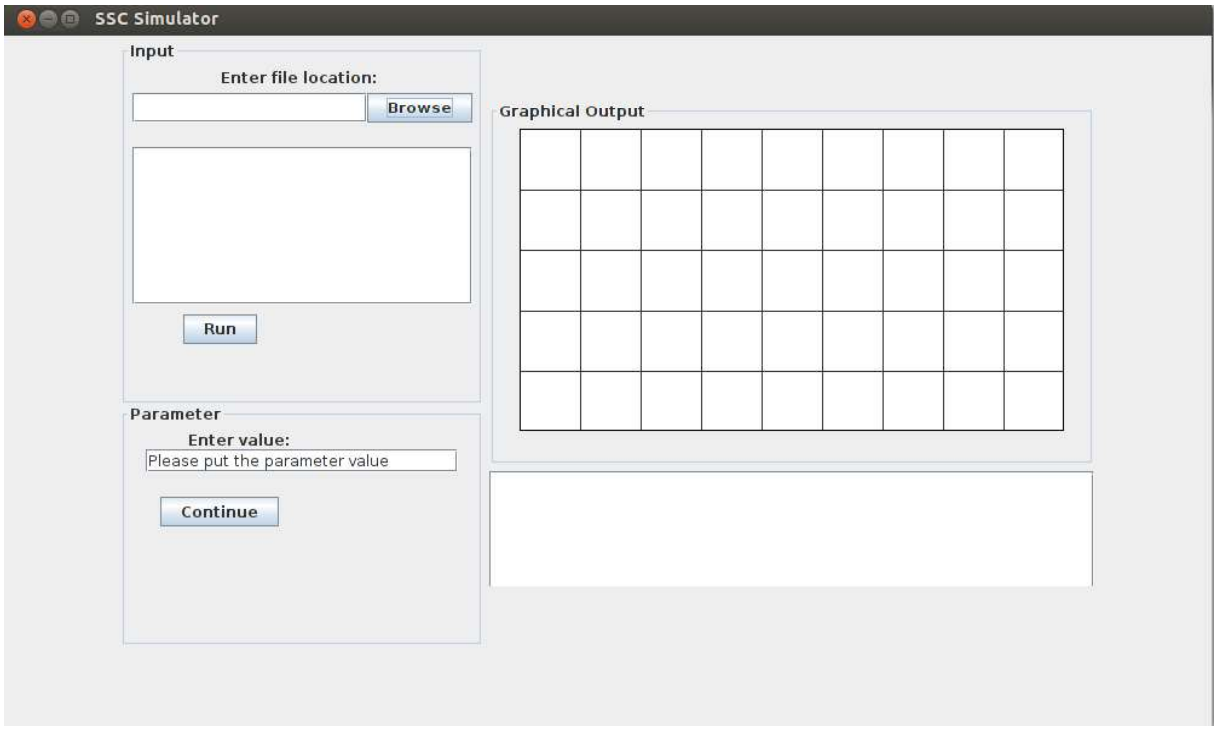
4. Documentation tools

For documentation purposes, **Dia** for block diagram drawing, **Pencil** for wire-frame drawing and **Umbrello** for class diagram drawing is being used.

Chapter 7: A Session with the Simulator

The functionalities that have been incorporated in the Simulator will now be explained with the screenshots of the circuit Simulator. User will be able to use this Simulator with ease by going through this interactive session.

Java must be installed in the terminal to use the circuit editor. Running the .jar executable file, a window will be opened like below.

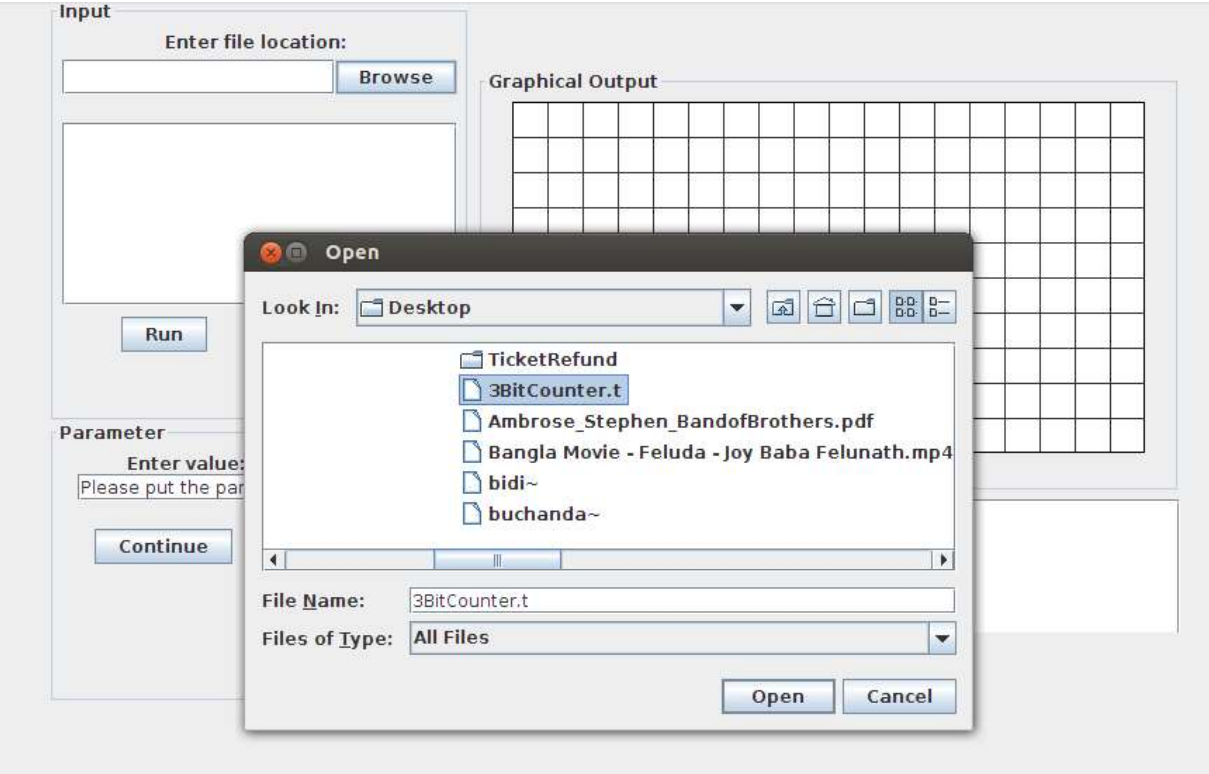


The Simulator will be opened with some default basic features. Each of the portions will be explained in this session. In the left-upper corner, Input section is situated, in the bottom-left corner Parameter section is situated and in the right hand side Output section is situated.

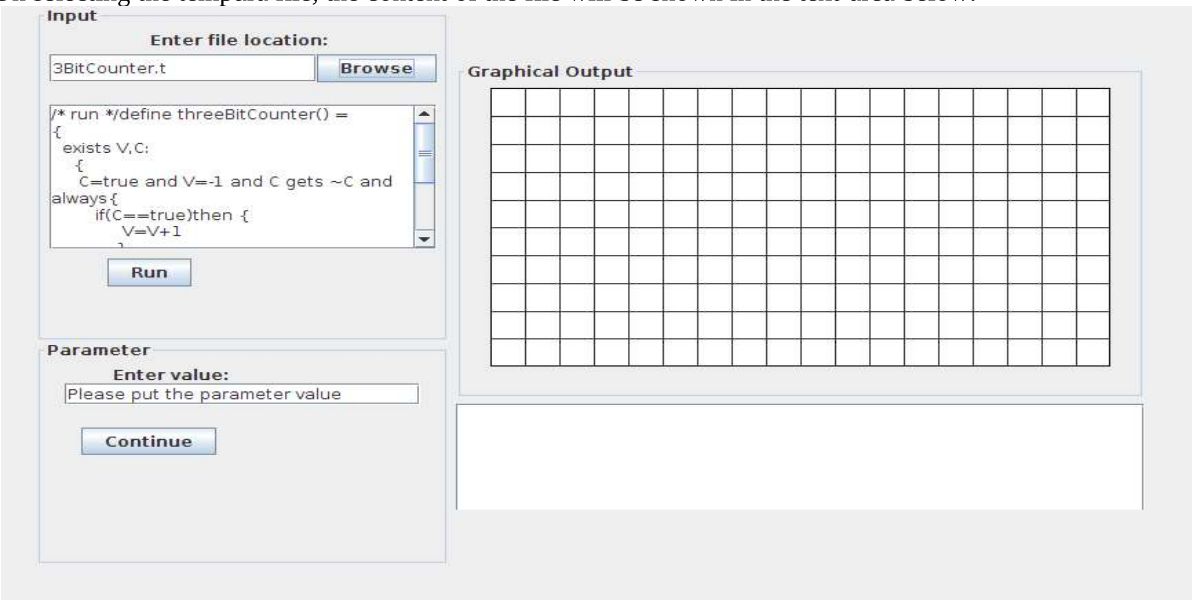
For demonstration, a 3-bit counter program has been taken as an example.

```
 /* run */define threeBitCounter() =
{
  exists V,C:
    {
     C=true and V=-1 and C gets ~C and always{
        if(C==true)then {
            V=V+1
           }
        else {
            V=V
        }} and len 15
    }
}.
```

In input section, on clicking “Browse” button, a file chooser will be displayed



On selecting the tempura file, the content of the file will be shown in the text-area below:



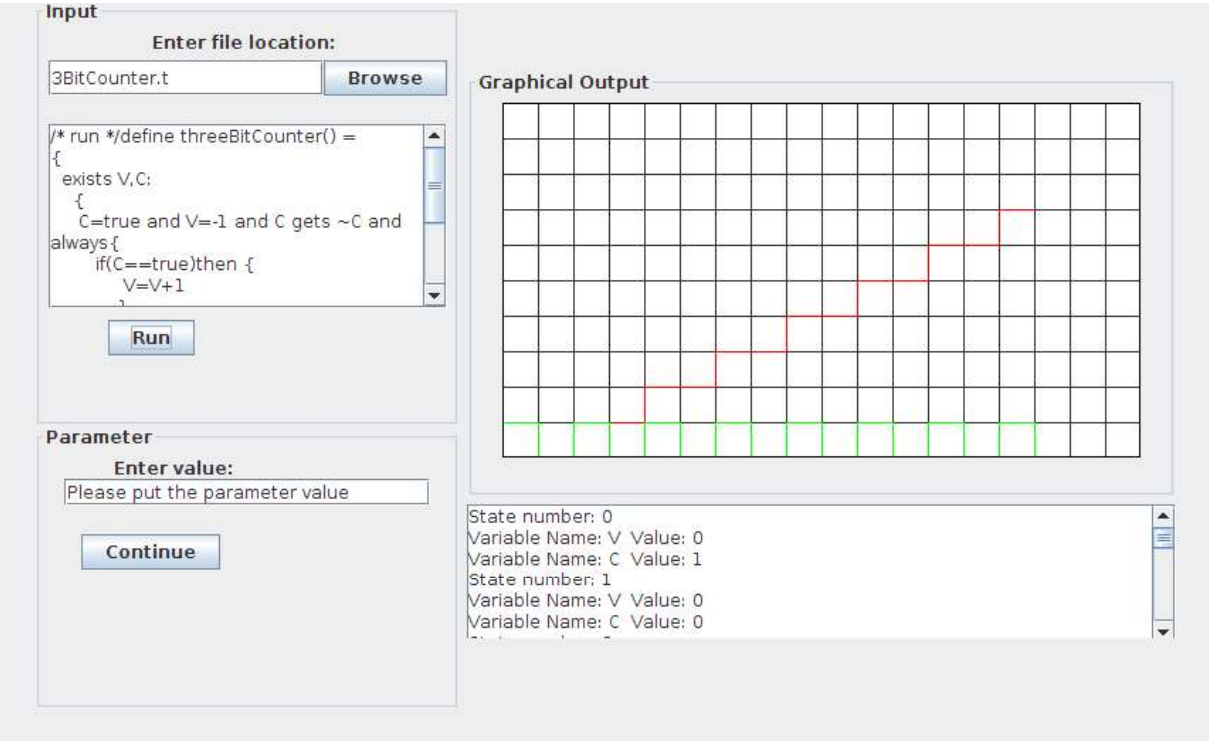
Through this file chooser, required tempura file will be choosen. The program given as input is displayed in the text-area mentioned above.

User can also provide Tempura program through this text-area.

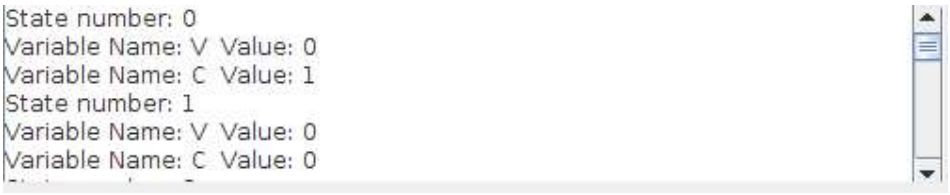
After providing the program, “Run” button in Input section is needed to be clicked.

On clicking the “Run” button, simulator will start working. And output will be displayed in the Output section.

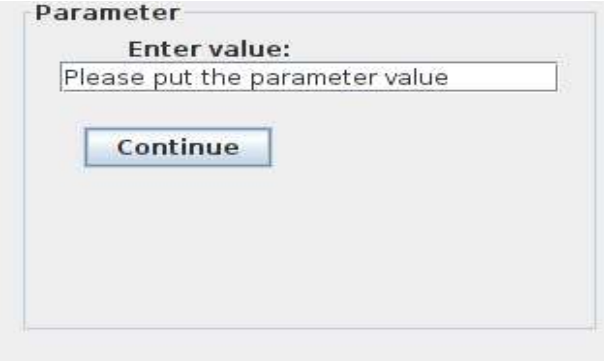
For the program taken for demonstartion, output will be displayed in the following manner.



and textual output will be as follows:



Additional parameter values can also be provided using the “Parameter” section.



This is a over-all description for running the Simulator. These functions are readily available with this circuit editor. More enhanced features will be enabled in future.

Chapter 8
Conclusion and Future Scope

The simulator is taking Tempura program as input. The interpreter built for this project is for Tempura programming language. But there are so many Tempura like language for describing hardware language. So the interpreter module can be replaced with interpreter of language other than Tempura and with small modifications in the GUI part, simulation can be done.

Chapter 9
Bibliography

1. **Executing Temporal logic programs** by **Ben Moszkowski**.
2. https://www.tech.dmu.ac.uk/STRL/ITL/itlhomepagese6.html

Links to the homepage of the tools used:
1. Java [http://www.java.com/en/]
2. JavaCC [https://javacc.java.net/]
3. NetBeans IDE [https://netbeans.org/]
4. Pencil [http://pencil.evolus.vn/]
5. Umbrello [http://umbrello.kde.org/]
6. Xfig [http://www.xfig.org/]

Appendix I:
Project Source Code DVD

Appendix II:
Link to Project .jar File: **http://bit.ly/1pkWqMa**