CPSC 490 Final Report

BooLeX: Language-based Circuit Simulator for Digital Logic

Student: Daniel Dickstein, SY ’16
Adviser: Ruzica Piskac

Overview

BooLeX originated in CPSC 439: Software Engineering, Spring 2014 as a project to address the difficulties new computer science and electrical engineering students face in first learning the laws of digital circuitry and Boolean logic. BooLeX is an educational tool that allows students to assemble virtual circuits and observe their performance on various input signals. By offering a visual aid to students learning about digital logic and circuitry, BooLeX enables students to test their own circuit designs and solidify the fundamental concepts of Boolean algebra.

Why BooLeX?

The tools currently available to students studying digital logic are unfortunately lacking. Some tools such as Logic.ly\(^1\) suffer unexpected behavior in certain sequential circuit designs, while others such as CircuitLab\(^2\) operate on a lower level than the abstracted principles of digital logic, and though offer more detailed simulations, can be overwhelming for new students struggling with the concepts of Boolean algebra. Most simulators such as Logic.ly and CircuitLab further suffer from the slow rate at which the simulations must be built, as the circuits must be built using a drag-and-drop graphical interface rather than using a text-based interface that allows students to simply write a description for the circuit they would like to simulate.

BooLeX uses a novel evaluative model by translating circuits created in its drag-and-drop interface into a simple domain-specific language (DSL) that is evaluated on the back end. This linguistic evaluation allows circuits to be simulated visually in a similar way to the other simulators, but offers the additional benefit of allowing users to directly express circuits in the BooLeX DSL so they can express more complex circuits more rapidly and can build more intricate designs.

Recognizing the advantage of directly expressing circuit designs over using a drag-and-drop simulator, some electrical engineering courses rely on hardware description languages such as VHDL to teach students to construct more complex
circuits. However, many students encountering code for the first time find the fully featured VHDL overwhelming and its pedagogical utility is limited. The BooLeX DSL, in contrast, is designed to be simple and intuitive, and is stripped down to the most basic essentials necessary for constructing and analyzing circuits in an educational context.

BooLeX DSL

The top-level entity in the BooLeX DSL is a module, which is comprised of a series of circuit definitions for the top-level “main” circuit and each integrated circuit upon which it relies. The basic structure of a circuit definition is:

```boolex
\text{circuit} \ \text{circuitName}(\text{input1, input2, \ldots})
\ a, \ b, \ldots = \text{exp1, exp2, \ldots}
\ x, \ y, \ldots = \text{exp1, exp2, \ldots}
\ \ldots
\ \text{out} \ output1, output2, \ldots
\end
```

Comments

BooLeX supports single-line comments beginning with a # symbol like many standard scripting languages.

Circuit Signature

Circuit definitions begin with a header, or signature, that declares the circuit’s name and formal parameters. It was designed to parallel the common function signature format of scripting languages like Ruby and Python. The \text{circuit} keyword is followed by the name of the circuit, and then by an optional list of input parameters. For a circuit that takes no input parameters, parentheses are omitted. After the signature comes the circuit body, followed by the \text{end} keyword to denote the conclusion of the circuit definition.

Circuit Body

The circuit body is comprised of a sequence of assignment statements, and concludes with an output statement. Circuits with no assignment statements are permitted, but every circuit needs to define an output statement with at least one output expression.
Wires, Sockets, and Variables

Although every wire in a circuit has a state (HIGH / LOW) at any given time, multiple wires can share the same state. In particular, multiple wires connected to the same gate’s output will all share the same state, because each signal that emerges from that gate is sent to each output wire. Likewise, multiple wires from the same input source all share the same state.

Figure 1: Full-Adder Circuit

For example, in the above full-adder circuit diagram (figure 1), the three inputs each have two wires connected to them, and the values on these wires change simultaneously with the input. Likewise, the output of the first XOR gate is connected to two wires, each of which will change simultaneously with the output of the gate. Thus, representing individual wires as variables is impractical because there are multiple wires that can be affected simultaneously. Instead, we want each variable to represent a cluster of wires that share one state.*

The way the original BooLeX team decided to handle this problem by conceiving of the basic element of a circuit as a socket rather than a wire. We defined a socket to be a “point of state” in a circuit, and placed input and output sockets before and

* If BooLeX were designed to more realistically depict circuit state, it would need to track each wire independently, because state changes are actually rather complex. A more advanced timing simulator would account for contamination and propagation delays and for meta-stability. Since these concepts are advanced for new programmers and engineers, and since BooLeX is designed first and foremost to be a simple educational tool, we remove these complications from our model, so for our purposes the wires share the same state.
after each gate respectively. A wire that connects the output of one gate to the input of another is modeled as the output socket of one gate targeting to the input socket of another. The significant difference here though is that a socket can have multiple targets, and signal changes are relayed to all targets at once. We also assign a socket to each circuit input. Thus, the sockets and their targets for the full-adder circuit can be modeled as follows:

![Diagram of Full-Adder circuit with socket representation](image)

In the above diagram (figure 2), sockets are represented by the circles, and connections between source sockets and target sockets by arrows. Note that although each socket can have multiple targets, each socket can only have one source. Thus, we let variables represent sockets such that assignment to variables (which occurs once and only once per variable) represents the socket source, and each use of the variable in an expression represents a target of the socket.

**Assignment Statement**

Assignment statements take the following simple form:

\[
\text{var1, var2, ...} = \text{exp1, exp2, ...}
\]

BooLeX expressions are discussed further in the next section. Note that the number of variables need not correspond to the number of expressions, but rather must correspond to the number of expression outputs. While most expressions have exactly one output (for which this distinction is meaningless), some expressions can have multiple outputs, and must therefore have multiple variables to which these outputs are assigned.
Expressions

Valid BooLeX expressions can be defined recursively as follows:

Base expressions
- A variable, which can be any Java-standard alphanumeric literal
- A value, which can be either true or false.
- The expression clock(period) (representing clock-based inputs)

Recursive expressions
- Not expression: not exp | exp'
- Buffered expression: `exp
- And expression: exp1 and exp2 | exp1 * exp2
- Or expression: exp1 or exp2 | exp1 + exp2
- Xor expression: exp1 xor exp2 | exp1 ^ exp2
- Nand expression: exp1 nand exp2
- Nor expression: exp1 nor exp2
- Xnor expression: exp1 xnor exp2
- Circuit call: circuitName(exp1, exp2, ...)

The order of precedence for these recursive expressions is as follows:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true, false, any variable</td>
</tr>
<tr>
<td>2</td>
<td>Clocks, circuit calls</td>
</tr>
<tr>
<td>3</td>
<td>Buffered expressions</td>
</tr>
<tr>
<td>4</td>
<td>Not expressions</td>
</tr>
<tr>
<td>5</td>
<td>And / Nand expressions</td>
</tr>
<tr>
<td>6</td>
<td>Or / Nor expressions</td>
</tr>
<tr>
<td>7</td>
<td>Xor / Xnor expressions</td>
</tr>
</tbody>
</table>

Note that of all of these expressions, the only type of expression that can yield more than a single output is the circuit call, because the invoked circuit can have multiple outputs. All expressions yield at least one output.

Output Statement

The output statement takes the following simple form:

```plaintext
out output1, output2, ...
```

The outputs of a circuit are the outputs of the expressions in the output statement. As in the case of variable assignment, it is the total number of outputs across all of
the output expressions, rather than the number of output expressions itself that determines the number of outputs for the circuit.

Below is an example of BooLeX code that can generate the full-adder circuit from figure 1.

```boolex

circuit FullAdder(A, B, C_in)
  D = A xor B
  S = D xor C_in
  C_out = A and B or C_in and D
  out S, C_out
end
```

Notice that the variables each represent a socket, which we define so that we can use the value in multiple places. The complex computation of `C_out` does not involve any cluster of wires that share states, so we can string together the gates in a single expression rather than having several intermediate variables. We could have also eliminated the `S` and `C_out` variables entirely and placed the expressions in the output statement, but it is good practice to create variables for readability even if they are not strictly necessary.

**Representing Sequential* Circuits**

One of the important differences between BooLeX and a standard scripting language is that since variables in BooLeX describe circuit sockets (which are abstractions of wires), they can be defined interdependently to create a loop. For example, consider the following definition of an SR-latch:

```boolex

circuit SRLatch(S, R)
  Q = R nor Q2
  Q2 = S nor Q
  out Q, Q2
end
```

* A *sequential circuit* is a “stateful circuit.” It maintains some sort of state with a loop so that the output is both a function of the new input and the old input. In contrast, a *combinational circuit* is a circuit that maintains no state whose input directly affects its output (e.g. a full-adder).
In this example, the result of Q depends on Q2, and the result of Q2 depends on Q. While this mutual dependence is unacceptable in a normal program that executes from top to bottom and requires variables to be defined before they are used, loops are permitted in circuits (and are often necessary), so this mutual dependence is supported in BooLeX so long as all variables that are used have an associated assignment at some point in the circuit definition. Also, since these variables represent clusters of wires that are fixed in place, they cannot be reassigned values (that would signify the dynamic movement of a wire in the circuit). The consequence of both of these rules is that every variable used in an expression in the circuit definition either must have exactly one associated assignment or must be a circuit parameter.

Back End Architecture

The new version of BooLeX is written in Scala (instead of Java) in order to take advantage of its expressive power and set of functional programming features such as pattern matching. Because the BooLeX back end is structured as a compiler, a more functional language was more appropriately suited for the task of translation and interpretation.

Note that none of the original logic from the first versions of BooLeX is present in the latest product. In addition to transitioning from Java to Scala, the parser, type checker, and evaluation model were all entirely rewritten to improve design, functionality, and flexibility.

The following discussion of the new BooLeX architecture is divided into the multiple phases of BooLeX compilation.

Lexical and Syntactic Analysis

While the original version of BooLeX relied on ANTLR$^3$ to handle this stage of compilation, BooLeX now relies on Scala’s own parser-combinators (available in the standard library). BooLeX uses a Packrat parser$^4$ to allow for unlimited look-ahead and linear parse time. Packrat parsers simply choose the first matching rule, so they do not suffer from ambiguity, and the BooLeX grammar is designed to accommodate this expectation. Additionally, although Packrat parsers could not originally support left-recursive grammars, more recent research$^5$ has rendered them more flexible, so the BooLeX grammar uses direct left-recursion.

Each type of BooLeX expression that is parsed is represented in a corresponding case class, and the parse tree is translated into an abstract syntax tree represented by these case classes. The primary advantage of representing each expression and
statement as a case class is that case classes support pattern matching, which is used extensively by the type-checker, circuit builder, and VHDL pre-processor.

One unfortunate consequence of relying on Scala’s parser-combinators is that only one lexical or syntactic error is reported at a time. Fortunately, the number of such mistakes one can make in the DSL is limited due to its simplicity, and the more common mistakes that are detected during the following semantic analysis phase can be reported in clusters.

**Semantic Analysis (Type checking)**

BooLeX represents calls to circuits with no inputs as simply a statement of the circuit’s name, so such calls are indistinguishable from regular variables when being parsed. Thus, before the type checking begins, the abstract syntax tree is briefly traversed, and all of these mislabeled variable expressions are replaced by circuit calls expressions.

While checking types is fairly straightforward since there are only two basic types to which all expressions resolve (circuits and Boolean values), the type checker also includes a number of other more complex checks in its semantic analysis. The total list of verifications made by the type checker appears below:

**Standard Checks**
- 1. Ensure no duplicate identifiers
- 2. Ensure presence of main circuit
- 3. Ensure variable and value counts match for each assignment statement
- 4. Ensure correct number of parameters for circuit calls
- 5. Ensure correct number of parameters for Boolean expressions
- 6. Ensure types match expressions where appropriate

**Advanced Checks**
- 1. Ensure no cyclic dependencies exist among circuits
- 2. Support for underscore to represent ignored value
- 3. Ensure that all forward-references are satisfied by subsequent declarations
- 4. Warn about any variables not affected by circuit input
- 5. Warn about any variables no affecting circuit output

One of the challenges in ensuring agreement between variable and value assignment is that the number of expressions yielding values does not necessarily correspond to the number of outputs from those expressions (since circuit calls can produce more outputs). Thus to type check any given circuit, we first need to know how many outputs are actually emitted from the circuits on which it depends. To address this problem, as well as to detect circular circuit dependencies, the type checker constructs a directed graph in which each defined circuit is represented as a
node with edges to its dependencies (the other circuits it calls) and confirms that the graph has no cycles. After this confirmation, a topological sort is applied to the directed acyclic graph (DAG) and the circuits are processed in order so that all the dependencies of any given circuit have already been checked before the circuit’s own type checking begins (which means the number of outputs from each of the circuit calls is known).

Note that previous versions of BooLeX had the requirement that circuits be defined after their dependencies (similar to C), which removed the need for this more complex computation. The new version of BooLeX developed for this project added the flexibility of allowing circuit definitions to be arranged in any order (similar to Java).

**Intermediate Tree Generation**

After the submitted code has been type checked and the abstract syntax tree (AST) generated, the back-end forks in two possible directions: interpretation and translation. Interpretation is the standard back end that assembles a circuit from the AST and simulates circuit events by firing signals through the assembled circuit of interconnected objects. The translation back end, in contrast, generates VHSIC Hardware Description Language (VHDL) files with a description that models the circuit represented by the AST so that it can be analyzed with industrial-grade simulators like Quartus II and executed on a field-programmable gate array (FPGA). This report will discuss each in turn.

**Constructing the Evaluation Model**

The AST returned from the type checker is interpreted by a depth-first traversal where each node returns a partial circuit representing that sub-tree. At each circuit call, the definition of the dependency circuit is revisited so that a separate sub-circuit is generated with new objects on each independent circuit call. The partial circuit that is returned at each node is a composition of the basic socket and gate elements, which are the fundamental building blocks of circuits in the evaluation model.

BooLeX sockets, discussed in the earlier DSL section of this paper, each track clusters of wires with shared state. Each socket can have multiple targets (circuit elements to which the signal is forwarded) that can either be gates or other sockets. The circuit is constructed such that every gate has one or two input sockets (depending on whether it is a unary or binary gate) and one output socket, where the input sockets have their associated gate as their target and output sockets target other input sockets to pass along the signal.
BooLeX gates are simple objects that perform a Boolean computation on the values of each input socket and forward the appropriate signal to the output socket.

**Simulating the Evaluation Model**

Once the circuit has been constructed, we initialize it by propagating the signals from inputs driven by constants. Next we launch each clocked input on an independent thread. Finally, we prepare to receive updated input signals.

The core simulation has two primary threads: the “loader” and the “launcher.” As input signals arrive, they are inserted into a ConcurrentLinkedQueue, so they will be loaded in the order in which they are received. The loader thread’s job is to read incoming values off the queue and insert them into the “signal queue,” which is a priority queue of signals that are being simulated, where the highest priority signals are those next to be fired. The priority queue is maintained as a ConcurrentSkipListSet.

The launcher thread launches all the signals in the signal queue that have a priority of 0, and when no more are left, sends updates of any socket that changed its state to the front end and decrements the priority of all signals in the queue by 1. Every signal that is directly propagated from one socket to another has a delay of 0, and every signal that is propagated through a gate has a delay of 1 (simulating a gate delay).

The primary advantage of separating the loading step from the actual signal simulation is that incoming updates can be blocked while the circuit is still initializing, so that the visual simulation on the front end is easier to interpret.

Currently these updates appear in the standard output, but after the integration of the front end next term the updates will be visual.
Simulating: List(false, false, false)
{ (false, false, false) } 
{ (a: false), (b: false), (c_in: false) } 
{ (d: false), (s: false) } 
{ (c_out: false), (s: false) } 
{ (c_out: false) } 

Simulating: List(false, false, true)
{ (false, false), (true) } 
{ (a: false), (b: false) } 
{ (c_in: true), (d: false) } 
{ (c_out: false), (s: true) } 
{ (c_out: false) } 

Simulating: List(false, true, false)
{ (false, true), (false) } 
{ (c_in: false), (d: true) } 
{ (c_out: false), (s: false), (true) } 
{ (c_out: false) } 

Simulating: List(true, false, false)
{ (true, false), (false) } 
{ (c_in: false), (true) } 
{ (d: true), (s: true) } 
{ (c_out: false), (s: true) } 
{ (c_out: true) } 

Simulating: List(true, false, true)
{ (true, false), (false) } 
{ (c_in: false), (true) } 
{ (d: true), (s: false) } 
{ (c_out: false), (true) } 
{ (true) } 

Simulating: List(true, true, false)
{ (true, true), (false) } 
{ (c_in: false), (true) } 
{ (d: true), (false), (s: true) } 
{ (c_out: true), (true) } 
{ (true) } 

Simulating: List(true, true, true)
{ (true, true), (false) } 
{ (c_in: true), (false) } 
{ (true, true), (false) } 
{ (true, true), (true) } 
{ (true, true) } 

Figure 4: Output of Full-Adder circuit simulation
Translating BooLeX to VHDL

Translating BooLeX to VHDL involves several complications due to the expressive flexibility BooLeX was designed to accommodate and VHDL's rigid linguistic structure. In particular, the following differences complicated translation:

- BooLeX treats clock-based inputs as constant inputs, which can appear in any expression or be assigned to any variable, whereas VHDL requires that all inputs ultimately originate from the main entity as formal inputs.

- BooLeX allows expressions to be passed as arguments in circuit calls, and allows circuit calls to be used inside expressions. VHDL supports neither of these features; it requires that all inputs and outputs of each circuit call be explicitly defined variables.

- BooLeX is case-sensitive and distinguishes the variables x and X, while VHDL is not case-sensitive and also has a number of reserved keywords that cannot be used as variable and circuit names.

- VHDL requires separate variables for circuit input and output, whereas BooLeX allows the same input variable to be returned as output from the circuit.

Because of these challenges, the AST could not be directly translated to VHDL. First, an intermediate pre-processing stage rewrites the AST to accommodate these linguistic discrepancies: it parameterizes clock expressions, rewrites assignment and output statements to accord with VHDL syntax rules, and renames variables to adhere to VHDL naming restrictions while preserving their uniqueness and retaining as much of their original form as possible so the generated code is recognizable.

The second stage of translation is to transform this modified AST into VHDL. This piece of the translation is currently in progress, and should be finished prior to the start of the fall term.
Future Work (Fall Term)

Work on BooLeX will continue in the fall. Goals for next term include:

- Finishing the translation to VHDL
- Testing generated VHDL on FPGA
- Comprehensive unit testing of the back end
- Integrating the new BooLeX back end with the former front end
- Resolving remaining bugs on the front end
- Add support for user-written BooLeX code
- Add support for integrated circuits
- Add support for saving and loading circuits
- Expand the BooLeX tutorial
References

1 http://logic.ly/

2 https://www.circuitlab.com/


