COMS W4115

Programming Languages and Translators

Event Driven State Language (EDSL)

"It's not your daddy's car"

Final Report

Christopher D. Sargent
cds2131@columbia.edu
December 17, 2007

Contents

Examples and Figures

Introduction

In an era in which the prices of highly capable microcontrollers are plummeting as technology continues to improve at a rapid rate, the extra development costs associated with hardware-only or low-level, highly-optimized languages for devices requiring state machines can be mitigated by utilizing a compiled language specifically designed for the creation of event-driven state machines.

As hardware becomes more programmable and abstract, as is the case with some of the newer military radios, which are software-based and intended to replace the fixed-waveform legacy radios of the past, languages that allow the user to design hardware dependent functionality without relying on specific hardware are becoming more practical.

Languages like HDL and VHDL which can be used for this purpose have existed for years, but their focus in the way that variables are defined and utilized has always been from the hardware aspect. The optimizers for these languages have not progressed to the level of software programming languages, such as C or C++.

As many simpler electronic devices rely on events and states, an easy-to-program language tailored for state machines is beneficial not only from the stand-point of bringing a product to market earlier, but also from the stand-point of being able to fix a flawed design with a product without requiring a hardware swap or even worse, a complete recall.

An example of an electronic device that would be a good candidate for this technology is a coffee maker with a timed shut-off. The coffee maker has several states: off, on-making coffee, and on-warming coffee. A problem that can occur with coffee makers with timed shut-offs is that a design problem or lack of redundancy can cause the timing device within the coffee maker to malfunction, possibly creating a situation in which a fire hazard can occur. Being able to replace the software in this device to fix a design flaw is a rapid and cost effective way of mitigating risk. And depending on the level of technological advancement, the coffee maker may have the capacity of receiving the update via the internet.

Assumptions

A language of this nature cannot entirely abstract its hardware dependency, and the hardware unlike that of a personal computer, must be tailored to some degree for its particular application in the interest of cost savings amortized over the potentially large number of production units. What this necessitates is adaptive or multi-use hardware capable of producing input events and triggering output events.

But intelligent hardware is not all that is necessary to bridge the gap between an imperative programming language and a mass-produced device. The linker itself must have an understanding of the hardware that it is marrying the state machine with, to the extent that the programmer only needs to know a minimum of the specifics. In an ideal world, the programmer need not know whether an on button is a physical button that is pushed by the finger of a user or a button on a screen that the user clicks. For purposes of this demonstration, it shall be assumed that a hardware linker exists or could be created and that this component need only be simulated.

Requirements

This language shall closely resemble a common software programming language. It shall be intuitive and hardware-oriented, but not at the sacrifice of flexibility. Its compiler shall compile EDSL source into a common and reasonably portable intermediate language for further compilation and linking.

After the intermediate source is generated, the remainder of the final compilation and linking shall be a simulation of what would actually occur if the language and its compiler and linker were to be implemented in their entirety. The simulation shall output data suitable for determining whether compiled programs are functioning properly.

Concepts

The general concepts of EDSL are the domain, the events, the state machine, and its states. The domain is the space within which all events and the state machine exist. It is, for all intents and purposes, everything.

Events within EDSL are variables and they span the scope of the entire state machine and its states. Events are associated with *linkage*:

> *input* – An input event receives data from an external source. Such an input might be a button that the user could press.
>
> *output* – An output event transmits data to an external destination. Such an output might be a light emitting diode (LED) indicating the power on or off status.
>
> *inout* – An input/output event receives data from and transmits data to an external source/destination. It essentially combines the function of an input and an output.
>
> *internal* – An internal event or variable does not communicate externally. It can be used to cache values.

Events are also associated with *type*:

bool – An event of the Boolean type may take only one of two values, either *true* or *false*.

int – An event of the integer type may take a whole number value that is either positive, negative, or zero. Its length is bound by its bit-length representation.

timer – An event of the timer type is essentially a positive or zero integer in terms of its representation, but once set, it will decrement toward zero in increments of seconds, internally.

The state machine is at the core of programs written in EDSL. All of the executable statements within the main program make up the state machine. Transitions to states within the state machine can be automatic or they can be forced. Within the state machine are an arbitrary number of states, which can be explicitly identified with labels.

Language Tutorial

The mechanics of EDSL are very much like the C-language, only much simpler. The simplest of source files consists of:

```
/*
 * No Statements
 */
{
      // This is where the state machine must go
}
```

Example EDSL Source: No Statements

Essentially, without the comment lines, the program would consist only of the open and close braces.

Inputs and Outputs

A slightly more complex program might read a Boolean input and transmit its value to a Boolean output.

```
/*
 * Simple Input and Output
 */

// The input and output signal declarations
input bool blnInValue;
output bool blnOutValue;
{
// Single state
start:
      blnOutValue = blnInValue;
      goto start; // Loop
}
```

Example EDSL Source: Simple Input and Output

With this simple example program, what is not shown is the representation of the input and output as simulated on a personal computer. Within the simulated input and output, Boolean values are represented by 0 and 1, 0 representing *false* and 1 representing *true*. Additionally, the input and output files are named using their event or variable names with the prefix *var_* and the suffix *.txt*. This file is most-likely located in the execution path of the program, although this might be operating system dependent.

The first value in the input file, *var_blnInValue.txt* is a flag indicating whether or not there is new data to be read. When the input is read, this flag is reset to 0. If the user

provides new data, he or she should set this flag to 1. The second value in the file is the data itself, be it a Boolean or integer literal. The only value in an output file, *var_blnOutValue.txt* is the data value.

The following is an example of the simulation of this program and the values within the input and output files:

Description	**var_blnInValue.txt**		**var_blnOutValue.txt**
Initial Values	0	0	0
User Sets Input Value to *true*	1	1	0
Program Reads the Input from the File	0	1	0
Program Writes the Output to the File	0	1	1

Warning: If the program requires an input, then the user must specify the input file. This would not be a problem if the program were communicating with actual hardware, but as this is a simulation and because short-cuts were taken the input file must be present. This is not a problem with output, because the output file is generated by the program.

EDSL also contains a linkage specifier that allows the input and output program to be simplified further:

```
/*
 * Combined Input and Output
 */

// The input/output signal declaration
inout bool blnInOutValue;
{
// Single state
start:
      goto start; // Loop
}
```

Example EDSL Source: Combined Input and Output

This program does essentially nothing, but when the user sets the value in *var_blnInOutValue.txt*, it is immediately written back to this file for the purpose of resetting the new data flag. The file format for *inout* linkage is the same as that for *input* linkage.

States, Gotos, and Labels

Transitioning between states can happen implicitly, if no statement of code prevents a transition or it can happen explicitly through the use of a `goto` statement:

```
/*
 * State Transition
 */
```

```
// The output state declaration
output int intState;
{
state1:                 // label
      intState = 1;
state2:                 // label
      intState = 2;
state3:                 // label
      intState = 3;
      goto state1;
}
```

Example EDSL Source: State Transition

Within this simple example, the current state is written to the output file var_intState.txt. The transitions from state 1 to 2 and state 2 to 3 are implicit. The transition from state 3 to 1 occurs when the `goto` statement is executed, and it is therefore explicit.

Conditional Statements

The only conditional statement allowed within EDSL is the `if-else` statement. With this statement, the keyword `if` is followed by an expression encapsulated by parentheses. If the expression evaluates to *true*, then the statement that follows is executed. If it is *false*, then if an `else` keyword is present, the statement which follows the `else` is executed.

```
/*
 * Conditional Statement
 */

// The declarations
input bool blnIn1;
input int intIn2;
internal int intVal;
output int intOut;
{
      intOut = 0;
      intVal = 0;

state:
      // First conditional
      if (true == blnIn1)
            intOut = intVal;
      else
            intVal = intIn2;

      goto state; // Loop
}
```

Example EDSL Source: Conditional Statement

This example is quite useless, but it demonstrates the conditional statement and the use of an internal variable. If the Boolean input is *true*, then the value cached in the internal variable is sent to the output. If it is *false*, then the value from the integer input is read and cached in the internal variable. Internal variables function like any other variable or event, but they do not interface externally. The following represents the values of the variables if this program were simulated:

blnIn1	**intIn2**	**intVal**	**intOut**
false	0	0	0
false	2	2	0
false	3	3	0
true	4	3	3
false	4	4	3
true	4	4	4

Expressions

There are unary and binary expressions within EDSL. Unary expressions take a single operand and binary expressions take two operands. Most of these operators function similarly to those that are in the C-language.

Unary Operators

There are two unary operators, one that takes a Boolean operand and returns an expression of the Boolean type and another that takes an integer operand and returns an expression of the integer type.

```
/*
 * Unary Expression
 */

// The input declarations
inout bool blnInOut1;
inout int intInOut2;
{
state:
      // Invert both signals
      blnInOut1 = !blnInOut1;       // e = not e
      intInOut2 = -intInOut2;       // e = -e

      goto state; // Loop
}
```

Example EDSL Source: Unary Expression

Both operators perform inversion, logical and additive inversion.

Binary Operators

The EDSL binary operators are mostly the C-language binary operators with a few additions. These operators include the assignment operator, logical operators, comparison and inequality operators, shift operators, and addition and multiplication operators.

```
/*
 * Binary Expression
 */

// The declarations
input int intIn1;
input int intIn2;
internal bool blnInterm1;
internal int intInterm2;
output int intOut;
{
state:
      // Binary calculations with assignments
      intInterm2 = intIn1 >> 2;           // right shift by 2
      intInterm2 = intInterm2 * intIn2;   // multiplication
      intInterm2 = intInterm2 + intIn1;   // addition

      // Inequalities and Exclusive-OR
      blnInterm1 = intInterm2 > intIn1;
      blnInterm1 = blnInterm1 ^^ (intIn1 > intIn2);

      // Inequality
      if (true == blnInterm1)
      {
            // Assignment
            intOut = intInterm;
      }
      else
      {
            // Assignment
            intOut = intIn1;
      }

      goto state; // Loop
}
```

Example EDSL Source: Binary Expression

More Realistic Example

Up to this point, the examples have been simple in nature and relatively useless. The following example should demonstrate the potential of this language as applied to a realistic control problem: a coffee-maker control system.

```
/*
 * Coffee Maker
```

```
 */

// Power button event
inout bool blnPowerButton;

// Water heating element event
output bool blnHeatingElement;

// Hot plate event
output bool blnHotPlate;

// Automatic shut-off timer
internal timer timShutoff;

// Water empty timer
internal timer timWaterEmpty;

{
// Initial state
powerOff:
      // Initialize the heaters to off
      blnHeatingElement = false;
      blnHotPlate = false;

      // Stay in this state if the power is off
      if (blnPowerButton == false)
            goto powerOff;

      // Turn on the heating element
      blnHeatingElement = true;

      // Set the water empty timer
      timWaterEmpty = 5;

// Make the coffee
makeCoffee:
      // Make sure the power isn't off
      if (blnPowerButton == false)
            goto powerOff;

      // Check to see if there is water
      if (timWaterEmpty > 0)
      {
            // Stay in this state
            goto makeCoffee;
      }

      // Make sure the power isn't off
      if (blnPowerButton == false)
            goto powerOff;

      // Turn off the heating element
      blnHeatingElement = false;

      // Turn on the hot plate
      blnHotPlate = true;
```

```
      // Initialize the shutoff timer
      timShutoff = 8;

// Wait until the pot is shut off or timer runs out
standby:
      // Make sure the power isn't off
      if (blnPowerButton == false)
            goto powerOff;

      // Check to see if the timer runs out
      if (timShutoff > 0)
      {
            // Loop
            goto standby;
      }

      // Turn off the hot plate
      blnHotPlate = false;

      // Turn the power off
      blnPowerButton = false;

      // Allow the program to terminate for scripting
}
```

Example EDSL Source: Coffee Maker

Language Manual

Many of the sections and format of the language manual are similar to the C-language reference manual, because of its concise nature and the fact that much of the EDSL language syntax and functionality is similar to the C-language.

Tokens

Tokens can consist of whitespace, comments, identifiers, keywords, literals, or operators. Even though whitespace and comments are not part of the language, they are included within this section for syntactical completeness, but are not included within the concise grammar specification.

Whitespace

Whitespace:
‘ ‘
‘\t’
‘\n’
‘\r\n’

Whitespace consists of blanks or spaces, horizontal tabs, and newlines, or a concatenation of one or several of these. For purposes of compatibility with UNIX and Windows platforms, a newline may be either a linefeed character or a carriage return followed by a linefeed. Whitespace serves no other purpose than to separate other tokens.

Comments

Comment:
Traditional-Comment
End-of-Line-Comment

Traditional-Comment:
/* *Comment-End*

Comment-End:
* *Comment-End-Star*
[^*] *Comment-End*

Comment-End-Star:
/

* *Comment-End-Star*
[^/*] *Comment-End*

End-of-Line-Comment:
// *Characters-in-End-of-Line-Comment*

Characters-in-End-of-Line-Comment:
[^ '\n' '\r\n']
Characters-in-End-of-Line-Comment [^ '\n' '\r\n']

The traditional comment begins with the characters `/*` and ends with the characters `*/`. It can span multiple lines, but it does not nest with other comments.

The end-of-line comment begins with the characters `//`, and ends with the newline character or characters.

Identifiers

Identifier:
Characters-in-Identifier

Characters-in-Identifier:
*([A-Z] | [a-z] | _) ([A-Z] | [a-z] | [0-9] | _)**

An identifier begins with at least one letter or an underscore and follows with a sequence of zero or more letters, digits, or underscores. Identifiers are case sensitive and may have any length, subject to platform specific limitations.

Keywords

Keywords are identifiers that have special meaning within the language. Keywords should not be used except for their intended purpose. Keywords are identified within the following sections of this document by their courier font.

Literals

Literal:
Boolean-Literal
Integer-Literal

Boolean-Literal:
`true`
`false`

Integer-Literal:
    Decimal-Literal
    Hexadecimal-Literal
    Binary-Literal

Decimal-Literal:
    [0-9]+

Hexadecimal-Literal:
    0 (x|X) ([A-F] | [a-f] | [0-9])*

Binary-Literal:
    0 (b|B) [0-1]*

A Boolean literal has one of two values, either `true` or `false`.

An integer literal can be a decimal, hexadecimal, or binary number. The prefix is necessary for the parser to be able to distinguish the three different kinds of integer.

Operators

The function of the operators depends on their operands and they will be further explained in a later section.

Type and Type Conversion

There are three basic types. The first two types are simple value storage types. The last is unique in that it stores a value, but it also operates on that value.

Like the Boolean literal, a Boolean variable takes either the `true` or `false` value.

The integer variable takes a 32-bit signed integer. The minimum value is `-2147483648` and the maximum is `2147483647`.

A timer has the same size and representation as a positive integer, but it is constantly decremented until its value reaches zero. Its units are seconds relative to wall-clock time.

Automatic type conversion or promotion or truncation occurs as a result of certain operators and their required operands. Automatic or implicit conversion is the only allowed type conversion mechanism.

Boolean and Integer

A `bool` may be converted to an `int` without loss. If its value is `false`, then its integer representation will be `0`. If its value is true, then its integer representation will be a non-zero value.

An `int` may be converted to a `bool` with data loss. If its value is `0`, then its Boolean representation will be `false`. If its value is not `0` (either a positive or negative value), then its Boolean representation will be `true`.

Note that both Booleans and integers are internally represented as integers.

Integer and Timer

A `timer` may not take a value that is not an acceptable `int` value. In addition, a timer may not store a negative value.

The value that a `timer` currently has may be converted to an `int` without loss.

Expressions

The precedence of the expressions is in order that the expressions are in, in this section.

Primary Expressions

primary-expression:
Identifier
Literal
(*expression*)

The primary expressions consist of identifiers, literals, and expressions in parentheses. The expressions in parentheses are evaluated as if they did not have parentheses.

Unary Expressions

unary-expression:
primary-expression
- unary-expression
! *unary-expression*

The unary operators: - and ! evaluate from right-to-left.

- *unary-expression*

For the '-' or 'negative' operator, the result of the evaluation is the negative of the operand. The result of this expression is always an `int`. If the operand is a `bool`, then it is promoted to an `int`.

! *unary-expression*

For the '!' or logical 'negation' operator, the result of the `bool` expression is `true` if the operand is `false` and `false` if the operand is `true`. If the operand is an `int`, then it is truncated to a `bool`.

Multiplication, Division, or Remainder Expressions

multiplication-expression:
unary-expression
multiplication-expression * *unary-expression*
multiplication-expression / *unary-expression*
multiplication-expression % *unary-expression*

The multiplication, division, and remainder operators: *, /, and % evaluate from left-to-right. The result of these expressions is always an `int`. Both operands are of the `int` type, and if either or both are of the `bool` type then they are promoted to the `int` type.

multiplication-expression * *unary-expression*

For the '*' or 'multiplication' operator, the result of the first operand is multiplied by the result of the second operand.

multiplication-expression / *unary-expression*

For the '/' or 'division' operator, the result of the first operand is divided by the result of the second operand. If the second operand is zero, the result is undefined.

multiplication-expression % *unary-expression*

For the '%' or 'remainder' operator, the result is the integer remainder of the division of the first operand by the second operand. "If both operands are non-negative, then the remainder is non-negative and smaller than the divisor; if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor."[1]

[1] Kernighan and Ritchie 205.

Addition or Subtraction Expressions

addition-expression:
multiplication-expression
addition-expression + *multiplication-expression*
addition-expression - *multiplication-expression*

The addition and subtraction operators: + and – group from left-to-right. The result of these expressions is always an `int`. Both operands are of the `int` type, and if either or both are of the `bool` type then they are promoted to the `int` type.

addition-expression + *multiplication-expression*

For the '+' or 'addition' operator, the result of the first operand is added to the result of the second operand.

addition-expression - *multiplication-expression*

For the '-' or 'subtraction' operator, the result of the second operand is subtracted from the result of the first operand.

Shift Expressions

shift-expression:
addition-expression
shift-expression << *addition-expression*
shift-expression >> *addition-expression*

The two shift operators: << and >> group from left-to-right. The result of these expressions is always an `int`. Both operands are of the `int` type, and if either or both are of the `bool` type then they are promoted to the `int` type.

shift-expression << *addition-expression*

For the '<<' or 'shift left' operator, the result of the first operand is left-shifted by the number of bits given in the result of the second operand.

shift-expression >> *addition-expression*

For the '>>' or 'shift right' operator, the result of the first operand is right-shifted by the number of bits given in the result of the second operand.

Inequality Expressions

inequality-expression:
shift-expression
inequality-expression < *shift-expression*
inequality-expression > *shift-expression*
inequality-expression <= *shift-expression*
inequality-expression >= *shift-expression*

The four inequality operators: <, >, <=, >= group from left-to-right. The result of these expressions is always a `bool`. Both operands are of the `int` type, and if either or both are of the `bool` type then they are promoted to the `int` type.

inequality-expression < *shift-expression*

For the '<' or 'less than' operator, if the first operand is less than the second operand, then the expression yields `true`, if not, it yields `false`.

inequality-expression > *shift-expression*

For the '>' or 'greater than' operator, if the first operand is greater than the second operand, then the expression yields `true`, if not, it yields `false`.

inequality-expression <= *shift-expression*

For the '<=' or 'less than or equal to' operator, if the first operand is less than or equal to the second operand, then the expression yields `true`, if not, it yields `false`.

inequality-expression >= *shift-expression*

For the '>=' or 'greater than or equal to' operator, if the first operand is greater than or equal to the second operand, then the expression yields `true`, if not, it yields `false`.

Equality Expressions

equality-expression:
inequality-expression
equality-expression == *inequality-expression*
equality-expression != *inequality-expression*

The two equality operators: == and != group from left-to-right. The result of these expressions is always a `bool`. Both operands are of the `bool` type, and if either or both are of the `int` type then they are truncated to the `bool` type.

equality-expression == *inequality-expression*

For the '==' or 'equal to' operator, if the first operand is equal to the second operand, then the expression yields `true`, if not, it yields `false`.

equality-expression != *inequality-expression*

For the '!=' or 'not equal to' operator, if the first operand is not equal to the second operand, then the expression yields `true`, if not, it yields `false`.

Bitwise AND Expressions

bitwise-AND-expression:
equality-expression
bitwise-AND-expression & *equality-expression*

The bitwise 'and' operator evaluates from left-to-right. The result of these expressions is an int if either operand is an int, otherwise the result is a `bool`.

Bitwise XOR Expressions

bitwise-XOR-expression:
bitwise-AND-expression
bitwise-XOR-expression ^ *bitwise-AND-expression*

The bitwise 'exclusive or' operator evaluates from left-to-right. The result of these expressions is an int if either operand is an int, otherwise the result is a `bool`.

Bitwise OR Expressions

bitwise-OR-expression:
bitwise-XOR-expression
bitwise-OR-expression | *bitwise-XOR-expression*

The bitwise 'inclusive or' operator evaluates from left-to-right. The result of these expressions is an int if either operand is an int, otherwise the result is a `bool`.

Logical AND Expressions

logical-AND-expression:

bitwise-OR-expression
logical-AND-expression && bitwise-OR-expression

The logical ‘and’ operator evaluates from left-to-right. The result of these expressions is always a `bool`. Both operands are of the `bool` type, and if either or both are of the `int` type then they are truncated to the `bool` type.

Logical XOR Expressions

logical-XOR-expression:
logical-AND-expression
logical-XOR-expression ^^ *logical-AND-expression*

The logical ‘exclusive or’ operator evaluates from left-to-right. The result of these expressions is always a `bool`. Both operands are of the `bool` type, and if either or both are of the `int` type then they are truncated to the `bool` type.

Logical OR Expressions

conditional-expression:
logical-XOR-expression
conditional-expression || *logical-XOR-expression*

The logical ‘inclusive or’ operator evaluates from left-to-right. The result of these expressions is always a `bool`. Both operands are of the `bool` type, and if either or both are of the `int` type then they are truncated to the `bool` type.

Assignment Expressions

expression:
conditional-expression
identifier = expression

The assignment operator evaluates from right-to-left and requires a modifiable variable as its left operand. The value of expression replaces the value of the variable referred to by the variable and the expression is promoted or truncated as needed to match the type of the variable.

Variable Declarations

declaration:
Linkage Type Identifier ;

Linkage:

```
internal
input
output
inout
```

Type:

```
bool
int
timer
```

The scope of a variable begins at the end of its declaration and persists to the end of the program in which it appears. Variables have the same lifetime as the running program. A variable declaration is associated with an identifier and has two configurable attributes: linkage and type.

There are two types of linkage: `internal` and external and three types of external linkage: `input`, `output`, and `inout`. The `inout` linkage is the only bi-directional external linkage type.

Statements

statement:

label
expression-statement
compound-statement
selection-statement
jump-statement

Unlike expressions, statements do not have values.

Labels

label:

Identifier :

A label with an identifier declares that identifier. This identifier then becomes the target of a `goto`.

Expression Statements

expression-statement:

expression$_{opt}$ *;*

Each expression statement is followed by a semi-colon.

Compound Statements

compound-statement:
{ statement-list$_{opt}$ *}*

statement-list:
statement
statement-list statement

The compound statement is a list of zero or more statements of any kind.

Selection Statements

selection-statement:
`if` (*expression*) *statement ;*
`if` (*expression*) *statement ;* `else` *statement ;*

In both forms of the `if` statement, the expression, which must have arithmetic type, is evaluated …. If it is `true`, the first substatement is executed. In the second form, the second substatement is executed if the expression is `false`. "The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if` at the same block nesting level."[2]

Jump Statements

jump-statement:
`goto` *Identifier* ;

The identifier must be a label. Control transfers to the labeled statement.

Program Declaration

program-declaration:
declaration-list$_{opt}$ *compound-statement*

[2] Kernighan and Ritchie 223.

declaration-list:
    declaration
    declaration-list declaration

The entire program consists of an optional list of variable declarations followed by a compound statement, which is a list of statements surrounded by braces.

Grammar

The grammar is a concatenated listing of the entire grammar from the previous sections. The grammar listing progresses from lower precedence to higher precedence.

program-declaration:
    declaration-list$_{opt}$ *compound-statement*

declaration-list:
    declaration
    declaration-list declaration

declaration:
    Linkage Type Identifier ;

Linkage:
    `internal`
    `input`
    `output`
    `inout`

Type:
    `bool`
    `int`
    `timer`

statement:
    label
    expression-statement
    compound-statement
    selection-statement
    jump-statement

label:
    Identifier :

expression-statement:
    expression$_{opt}$ *;*

compound-statement:
    { statement-list$_{opt}$ *}*

statement-list:
    statement
    statement-list statement

selection-statement:
    `if` (*expression*) *statement ;*
    `if` (*expression*) *statement ;* `else` *statement ;*

jump-statement:
    `goto` *Identifier* ;

expression:
    conditional-expression
    identifier = expression

conditional-expression:
    logical-XOR-expression
    conditional-expression || logical-XOR-expression

logical-XOR-expression:
    logical-AND-expression
    logical-XOR-expression ^^ logical-AND-expression

logical-AND-expression:
    bitwise-OR-expression
    logical-AND-expression && bitwise-OR-expression

bitwise-OR-expression:
    bitwise-XOR-expression
    bitwise-OR-expression | bitwise-XOR-expression

bitwise-XOR-expression:
    bitwise-AND-expression
    bitwise-XOR-expression ^ bitwise-AND-expression

bitwise-AND-expression:
    equality-expression
    bitwise-AND-expression & equality-expression

equality-expression:
    inequality-expression
    equality-expression == inequality-expression

equality-expression != *inequality-expression*

inequality-expression:
shift-expression
inequality-expression < *shift-expression*
inequality-expression > *shift-expression*
inequality-expression <= *shift-expression*
inequality-expression >= *shift-expression*

shift-expression:
addition-expression
shift-expression << *addition-expression*
shift-expression >> *addition-expression*

addition-expression:
multiplication-expression
addition-expression + *multiplication-expression*
addition-expression - *multiplication-expression*

multiplication-expression:
unary-expression
multiplication-expression * *unary-expression*
multiplication-expression / *unary-expression*
multiplication-expression % *unary-expression*

unary-expression:
primary-expression
- *unary-expression*
! *unary-expression*

primary-expression:
Identifier
Literal
(*expression*)

Literal:
Boolean-Literal
Integer-Literal

Boolean-Literal:
`true`
`false`

Integer-Literal:
Decimal-Literal
Hexadecimal-Literal

Binary-Literal

Decimal-Literal:
[0-9]+

Hexadecimal-Literal:
0 (x|X) ([A-F] | [a-f] | [0-9])*

Binary-Literal:
0 (b|B) [0-1]*

Identifier:
Characters-in-Identifier

Characters-in-Identifier:
*([A-Z] | [a-z] | _) ([A-Z] | [a-z] | [0-9] | _)**

Project Plan

From conception to implementation, this project did not have a formal plan, and it lacked the need for one because it was not the work of more than one person. But it did have general guidelines that were followed.

Planning, Specification, Development, and Testing Process

A number of milestones were established up-front, some of which correspond to the course assignments. And as the project proceeded, additional milestones were added:

1. Identify the Language
2. Language Reference Manual
3. Identify the Output Format and Develop a Sample Output
4. Lexer/Parser
5. Tree Walker
6. Testing of Tree Walker using a Pretty Printer
7. Code Generator
8. Symbol Table
9. Error Handler
10. Testing of Code Generator
11. Report

The general concept for the language was devised as a result of exposures to C and VHDL, and the desire to create a language that would interface with hardware, but would not be overly difficult to program in. This concept was developed further and a preliminary example created for the whitepaper deliverable.

The language reference manual (LRM) was created as a result of research into other language reference manuals, including Java, C, and C++. The concepts for the LRM were formalized for the course deliverable.

Because of its similarity with the C-language, C was chosen as the output (or intermediate compiled format) for EDSL. A sample output was generated that would take advantage of the critical functionality of the language, providing an example of the translation that needed to be automated by the compiler.

Based on the LRM, the lexer and parser were coded in the ANTLR 2.7.7 language. The initial version of the ANTLR source was almost an exact translation of the grammar as specified.

Following the creation of the lexer/parser, a tree walker was coded in ANTLR 2.7.7. Initially, it was very similar to the design of the parser; however, as the tree nodes were better specified, the tree walker code was simplified. It was at this time that the grammar

and tree grammar were recoded in ANTLR 3.0.1, because of the tools available to simulate and debug grammars and the fixes to the error handling within the tree walker.

To ensure that the tree walker was working properly and to ease in to the development of the code generator, a pretty printer was created to display the simulation input.

The code generator was created primarily by replacing the pretty printer statements with C source code statements, almost one-for-one. At this point, the intermediate code format was changed from C to C++, because it was determined that a C++ class template and class would allow for greater flexibility for testing and would also simplify the declarations that the compiler needed to generate.

The symbol table was created as a hash table with its value being an object of a class, containing the linkage, type, and assigned state of the variable. The class type was expanded to include an abstract base class with derived classes to represent label and variable symbol table entries.

The error handler was much more difficult to implement and involved a deeper understanding of the errors and warnings that were needed. It was designed by first reviewing the compiler errors and warnings generated by C when similar syntactic and semantic errors were introduced. When the errors and warnings were identified, then the errors and warnings were implemented one-by-one and alpha testing performed.

The testing of the code generator involved the identification of the test categories: compiler usage, lexer, parser, semantic, and runtime. Based on knowledge of the language, tests were devised that would attempt to stress the language and expose design flaws.

The last step was the creation of the final report, detailing the project.

Programming Style Guide

For this project, the following style guidelines were established to provide good source encapsulation, concise and understandable source, and good documentation:

1. As much of the compiler source code as possible is included within the ANTLR source code.
2. ANTLR variables are generally a single lowercase letter corresponding to the first letter in a rule or token.
3. An extra space is inserted after an opening parenthesis and before a closing parenthesis in ANTLR expressions.
4. Rule and token identifiers in ANTLR expressions are included on their own lines and followed by their associated statements.
5. The usage of specific return values within ANTLR statements is used sparingly.

6. The Java library is utilized to the greatest extent possible to avoid integration of third party libraries that might have licensing restrictions.
7. Javadoc tags are included within the Java source code.
8. All Java and C++ variables are prefixed with:
    a. A single character identifier for the scope, followed by an underscore character ("m_" indicates a member variable within a class). Local variables do not have a scope prefix.
    b. Three or four characters identify the variable's type ("bln" indicates a Boolean, "int" indicates an integer, "map" indicates some type of map, "obj" indicates a generic object or an object that does not have a specifically assigned prefix).
9. Block comments are utilized for class, function, and variable declarations and line comments for statement description. Line comments are used liberally within the source.
10. Compound statements are utilized with statements such as if-statements or loops even when these statements are followed up by only a single statement, to explicitly segregate the source statements.
11. Simple literals are placed on the left side of variables in equality expressions to avoid potential problems that could occur if an assignment operator is mistakenly typed for an equality operator.
12. Each indentation level is a single tab.
13. Open braces indicating a compound statement begin on a new line at the current tab level.
14. Line lengths are not a set number of characters, but in general, each line of source should print without wrap-around.
15. Identifier names consisting of multiple words have them concatenated together with the first character of each of the words capitalized, except in the case of method names which begin with a lowercase letter and start with a verb.

Project Timeline

The dates recorded for these milestones are approximate. All milestones and components were completed by Christopher D. Sargent.

Milestone	**Date Completed**
Identify the Language	2007-09-20
Language Reference Manual	2007-10-14
Identify the Output Format and Develop a Sample Output	2007-10-17
Lexer/Parser	2007-10-27
Tree Walker	2007-11-01
Testing of Tree Walker using a Pretty Printer	2007-11-04
Code Generator	2007-11-24
Symbol Table	2007-11-25
Error Handler	2007-12-03

Testing of Code Generator	2007-12-17
Report	2007-12-18

Software Development Environment

EDSL was developed on a Dell Precision M65 with the Windows XP Professional operating system. ANTLR 2.7.7 was initially used to generate the compiler, but was replaced with ANTLR 3.0.1. Because of path problems within the ANTLR 3.0.1 runtime JAR and the fact that this JAR was not standalone, ANTLR was run through the ANTLRWorks 1.1.5 JAR, available from:

http://www.antlr.org/works/index.html

The ANTLR-generated Java source files were built with the Java JDK 1.6.0 Update 3, using the following command-line syntax:

java –cp antlrworks-1.1.5.jar org.antlr.Tool EDSL.g EDSLCompiler.g

For purposes of testing, the intermediate C++ source code generated by the EDSL compiler was built using Visual C++ in Visual Studio 2005 Professional using the command-line compiler *cl.exe*.

Because all of the testing was scripted, all applications were executed from the command-line. However, the ANTLRWorks IDE was used for some grammar syntax debugging.

Project Log

The project log includes the specific tasks needed for the completion of each milestone.

Task	**Date**
Assess Project Scope and Deliverables	2007-09-12
Brainstorm and Develop a Concept	2007-09-15
Create the Proposal	2007-09-20
Download ANTLR 2.7.7	2007-09-22
Become Familiar with Utilization of ANTLR 2.7.7	2007-09-30
Read through C, C++, and Java Grammars	2007-10-02
Design the EDSL Grammar	2007-10-11
Create the Language Reference Manual	2007-10-14
Create a Sample Output File for the Compiler in C	2007-10-17
Translate the Grammar into ANTLR 2.7.7 source	2007-10-27
Create a Tree Walker in ANTLR 2.7.7 source	2007-11-01
Simplify the Tree Walker	2007-11-02

Add Pretty Printer Statements to the Tree Walker	2007-11-04
Translate the ANTLR 2.7.7 Grammar to ANTLR 3.0.1	2007-11-07
Translate the ANTLR 2.7.7 Tree Walker to ANTLR 3.0.1	2007-11-08
Create the Event Template Class	2007-11-10
Create the Derived Timer Event Class	2007-11-17
Create the Code Generation Statements	2007-11-24
Implement the Symbol Class	2007-11-25
Implement the Symbol Table in the Parser and Tree Walker	2007-11-25
Implement Expression Type Passing	2007-11-27
Generate Code to Convert Binary and Hex Literals to Decimal	2007-11-28
Add Semantic Error/Warning Checking to the Parser	2007-12-02
Add Semantic Error/Warning Checking to the Tree Walker	2007-12-03
Clean up the main program and integrate it in a .g file	2007-12-05
Create a list of Tests	2007-12-10
Compiler Unit Testing	2007-12-12
Lexer Testing	2007-12-11
Parser Testing	2007-12-15
Semantic Testing	2007-12-17
Create the Final Report	2007-12-18

Architectural Design

The architecture of the EDSL compiler is broken into two diagrams: the major components of the translator and the external components that enable EDSL source to be turned into executable code.

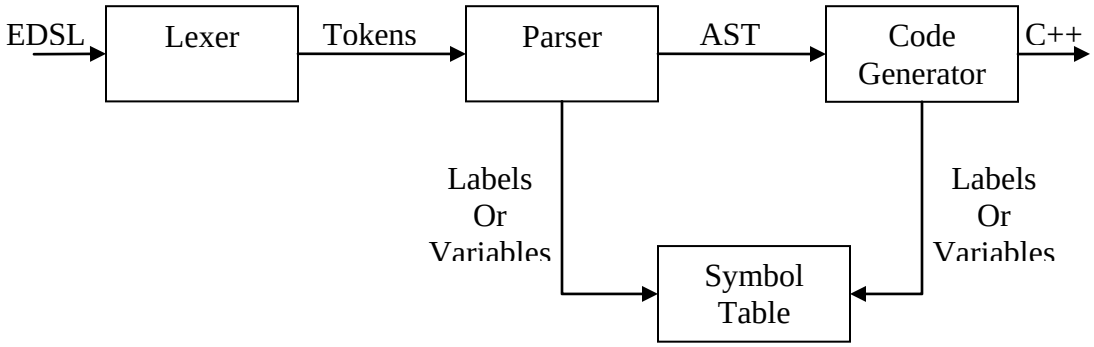

Figure: Translator Major Components Block Diagram

Although the Lexer and Parser are built from the same grammar file in ANTLR 3.0.1, their Java components are separate.

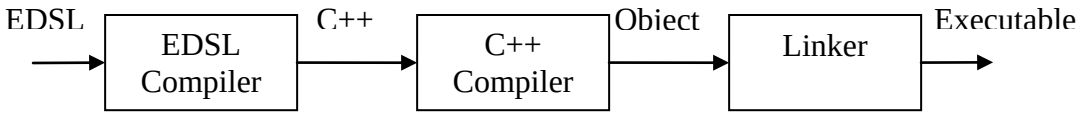

Figure: External Components Block Diagram

Lexer → Parser Interface

The interface between the Lexer and Parser is the stream of tokens read from the EDSL source file. These tokens are broken into lexemes.

Parser → Symbol Table Interface

The Parser uses the Symbol Table to store label and variable identifiers. The variables are stored with their linkage, type, assignment, and usage status. The labels are only stored with their usage status.

Parser → Code Generator Interface

The interface between the Parser and the Code Generator is the Abstract Syntax Tree, which contains the hierarchical structure of the tokens read from the source program.

Code Generator → Symbol Table Interface

The Code Generator or Tree Walker uses the Symbol Table to retrieve the variable and label information for semantic analysis.

Test Plan

Many small test programs were written to test specific aspects of the language. The following subset of runtime test programs were chosen because they demonstrate state machines that an actual user of the language might generate in an attempt to solve a real-world problem—and they happened to be devices that I was around quite a bit when I was thinking about this project.

Coffee Maker

The first test is a simple coffee maker design. Its EDSL source is located in the language tutorial. It has a single external input: power button and two external outputs: heating element on/off and hot plate on/off. Note that the power button is really an inout, so that it can be reset as a button would be after pushed. Even though a coffee maker or other state machine would probably not terminate like this, this is done to allow for scripted testing. This source is compiled with the EDSL compiler into the following intermediate source code:

```
#include <stdio.h>
#include <time.h>
#include "event.h"
#include "eventTimer.h"
int main()
{
CEvent<int> var_blnPowerButton("var_blnPowerButton", true, true);
CEvent<int> var_blnHeatingElement("var_blnHeatingElement", false,
true);
CEvent<int> var_blnHotPlate("var_blnHotPlate", false, true);
CEventTimer var_timShutoff("var_timShutoff", false, false);
CEventTimer var_timWaterEmpty("var_timWaterEmpty", false, false);
lab_powerOff:
var_blnHeatingElement = 0;
var_blnHotPlate = 0;
if (var_blnPowerButton == 0)
goto lab_powerOff;
var_blnHeatingElement = 1;
var_timWaterEmpty = 5;
lab_makeCoffee:
if (var_blnPowerButton == 0)
goto lab_powerOff;
if (var_timWaterEmpty > 0)
{
goto lab_makeCoffee;
}
if (var_blnPowerButton == 0)
goto lab_powerOff;
var_blnHeatingElement = 0;
var_blnHotPlate = 1;
var_timShutoff = 8;
lab_standby:
if (var_blnPowerButton == 0)
```

```
goto lab_powerOff;
if (var_timShutoff > 0)
{
goto lab_standby;
}
var_blnHotPlate = 0;
var_blnPowerButton = 0;
; }
```

Example C++ Source: Coffee Maker

After compiling and linking the C++ intermediate source, the program is executed. The following represents its output:

```
Read: <var_blnPowerButton> Old Value: 0 New Value: 1 (Time = 0)
Assign: <var_blnHeatingElement> Old Value: 0 New Value: 0 (Time = 0)
Assign: <var_blnHotPlate> Old Value: 0 New Value: 0 (Time = 0)
Assign: <var_blnHeatingElement> Old Value: 0 New Value: 1 (Time = 0)
Assign: <var_timWaterEmpty> Old Value: 0 New Value: 5 (Time = 0)
Assign: <var_blnHeatingElement> Old Value: 1 New Value: 0 (Time = 5)
Assign: <var_blnHotPlate> Old Value: 0 New Value: 1 (Time = 5)
Assign: <var_timShutoff> Old Value: 0 New Value: 8 (Time = 5)
Assign: <var_blnHotPlate> Old Value: 1 New Value: 0 (Time = 13)
Assign: <var_blnPowerButton> Old Value: 1 New Value: 0 (Time = 13)
```

Example Output: Coffee Maker

The reads and assignments are documented for each variable along with their old and new values. The time stamp allows for testing of the timer functionality—it is in seconds.

Microwave Oven

This program represents a simple microwave oven design. It has three external inputs: time set, start, and cancel and one external output: cooker on/off. Like the power button in the coffee maker example, the start and cancel inputs are really inout externals.

```
/*
 * Simple Microwave Oven
 */

// Inout Declarations -- so they can be reset
inout int   intTimeSet;
inout bool  blnStart;
inout bool  blnCancel;

// Output Declaration
output bool blnCooker;

// Internal Declaration
internal timer timCookTimer;
```

```
// State Machine
{
reset:
      // Reset the cooker
      blnCooker = false;

wait:
      // Check to see if the start button was pressed
      if (true == blnStart)
      {
            // Reset the start button
            blnStart = false;

            // Set the timer
            timCookTimer = intTimeSet;

            // Turn the cooker on
            blnCooker = true;

            // Start to cook
            goto cook;
      }

      // Wait
      goto wait;

cook:
      // Check to see if the cancel button was pressed
      if (true == blnCancel)
      {
            // Reset the timer
            timCookTimer = 0;

            // Reset the cancel button
            blnCancel = false;
      }

      // Check to see if time has expired
      else if (timCookTimer > 0)
      {
            // Keep cooking
            goto cook;
      }

      // Turn the cooker off
      blnCooker = false;

      // Terminate so the script will continue running
}
```

Example EDSL Source: Microwave Oven

As is the case with the coffee maker, the microwave oven example is allowed to terminate for scripted testing. This source is compiled with the EDSL compiler into the following intermediate source code:

```
#include <stdio.h>
#include <time.h>
#include "event.h"
#include "eventTimer.h"
int main()
{
CEvent<int> var_intTimeSet("var_intTimeSet", true, true);
CEvent<int> var_blnStart("var_blnStart", true, true);
CEvent<int> var_blnCancel("var_blnCancel", true, true);
CEvent<int> var_blnCooker("var_blnCooker", false, true);
CEventTimer var_timCookTimer("var_timCookTimer", false, false);
var_blnCooker = 0;
lab_wait:
if (1 == var_blnStart)
{
var_blnStart = 0;
var_timCookTimer = var_intTimeSet;
var_blnCooker = 1;
goto lab_cook;
}
goto lab_wait;
lab_cook:
if (1 == var_blnCancel)
{
var_timCookTimer = 0;
var_blnCancel = 0;
}
else
if (var_timCookTimer > 0)
{
goto lab_cook;
}
var_blnCooker = 0;
; }
```

Example C++ Source: Microwave Oven

After compiling and linking the C++ intermediate source, the program is executed. The following represents its output:

```
Read: <var_intTimeSet> Old Value: 0 New Value: 15 (Time = 0)
Read: <var_blnStart> Old Value: 0 New Value: 1 (Time = 0)
Read: <var_blnCancel> Old Value: 0 New Value: 0 (Time = 0)
Assign: <var_blnCooker> Old Value: 0 New Value: 0 (Time = 0)
Assign: <var_blnStart> Old Value: 1 New Value: 0 (Time = 0)
Assign: <var_timCookTimer> Old Value: 0 New Value: 15 (Time = 0)
Assign: <var_blnCooker> Old Value: 0 New Value: 1 (Time = 0)
Assign: <var_blnCooker> Old Value: 1 New Value: 0 (Time = 15)
```

Example Output: Microwave Oven

Random Value Generator

Secure identification cards generate a new random value after a certain period of time expires. These random values are utilized as part of a password for access to a system. Additionally, they display a certain number of bars to indicate to the user how much time he or she has to use the random value before it expires and is replaced by another value.

This example is a simple random value generator—real secure ID cards must use absolute time as part of their seed value such that the same technology can be used to determine whether the access is valid or not. The random value is between 0 and 32767.

```
/*
 * Random Value Generator
 *
 * Note:
 * The random algorithm was taken from rand.c in the
 * Microsoft Standard C library. The copyright statement is:
 *
 * Copyright (c) Microsoft Corporation. All rights reserved.
 *
 * The actual algorithm used is not important to the example.
 */

// The random value output
output int intRandomValue;

// The number of bars to display
output int intNumBars;

// The timer to determine when to change the value
internal timer timChange;

// The timer used to terminate for scripting
internal timer timTerminate;

// The last random value (or the seed at the start)
internal int intHold;

{
      // Reset the terminate timer
      timTerminate = 30;

      // Reset the time to change value
      timChange = 0;

      // Set the seed
      intHold = 0;

// Calculate the random value
calc:
      // Check to see if it is time to terminate
      if (timTerminate > 0)
      {
```

```
            // Check to see if it is time to change values
            if (timChange == 0)
            {
                  // Reset the time until next change
                  timChange = 10;

                  // Generate the hold value
                  intHold = intHold * 214013 + 2531011;

                  // Generate a new random value
                  intRandomValue = (intHold >> 16) & 0x7fff;
            }

            // Display the number of bars (out of a possible 5)
            intNumBars = 5 * timChange / 10;

            // Keep looping
            goto calc;
      }

      // Terminate after 30 seconds for scripting
}
```

Example EDSL Source: Random Value Generator

As is the case with the coffee maker, the microwave oven example is allowed to terminate for scripted testing. This source is compiled with the EDSL compiler into the following intermediate source code:

```
#include <stdio.h>
#include <time.h>
#include "event.h"
#include "eventTimer.h"
int main()
{
CEvent<int> var_intRandomValue("var_intRandomValue", false, true);
CEvent<int> var_intNumBars("var_intNumBars", false, true);
CEventTimer var_timChange("var_timChange", false, false);
CEventTimer var_timTerminate("var_timTerminate", false, false);
CEvent<int> var_intHold("var_intHold", false, false);
var_timTerminate = 50;
var_timChange = 0;
var_intHold = 0;
lab_calc:
if (var_timTerminate > 0)
{
if (var_timChange == 0)
{
var_timChange = 10;
var_intHold = var_intHold * 214013 + 2531011;
var_intRandomValue = (var_intHold >> 16) & 32767;
}
var_intNumBars = 5 * var_timChange / 10;
goto lab_calc;
}
```

```
; }
```

Example C++ Source: Random Value Generator

After compiling and linking the C++ intermediate source, the program is executed. The following represents its output:

```
Assign: <var_timTerminate> Old Value: 0 New Value: 30 (Time = 0)
Assign: <var_timChange> Old Value: 0 New Value: 0 (Time = 0)
Assign: <var_intHold> Old Value: 0 New Value: 0 (Time = 0)
Assign: <var_timChange> Old Value: 0 New Value: 10 (Time = 0)
Assign: <var_intHold> Old Value: 0 New Value: 2531011 (Time = 0)
Assign: <var_intRandomValue> Old Value: 0 New Value: 38 (Time = 0)
Assign: <var_intNumBars> Old Value: 0 New Value: 5 (Time = 0)
Assign: <var_intNumBars> Old Value: 5 New Value: 4 (Time = 1)
Assign: <var_intNumBars> Old Value: 4 New Value: 3 (Time = 3)
Assign: <var_intNumBars> Old Value: 3 New Value: 2 (Time = 5)
Assign: <var_intNumBars> Old Value: 2 New Value: 1 (Time = 7)
Assign: <var_intNumBars> Old Value: 1 New Value: 0 (Time = 9)
Assign: <var_timChange> Old Value: 0 New Value: 10 (Time = 10)
Assign: <var_intHold> Old Value: 2531011 New Value: 505908858 (Time = 10)
Assign: <var_intRandomValue> Old Value: 38 New Value: 7719 (Time = 10)
Assign: <var_intNumBars> Old Value: 0 New Value: 5 (Time = 10)
Assign: <var_intNumBars> Old Value: 5 New Value: 4 (Time = 11)
Assign: <var_intNumBars> Old Value: 4 New Value: 3 (Time = 13)
Assign: <var_intNumBars> Old Value: 3 New Value: 2 (Time = 15)
Assign: <var_intNumBars> Old Value: 2 New Value: 1 (Time = 17)
Assign: <var_intNumBars> Old Value: 1 New Value: 0 (Time = 19)
Assign: <var_timChange> Old Value: 0 New Value: 10 (Time = 20)
Assign: <var_intHold> Old Value: 505908858 New Value: -755606699 (Time = 20)
Assign: <var_intRandomValue> Old Value: 7719 New Value: 21238 (Time = 20)
Assign: <var_intNumBars> Old Value: 0 New Value: 5 (Time = 20)
Assign: <var_intNumBars> Old Value: 5 New Value: 4 (Time = 21)
Assign: <var_intNumBars> Old Value: 4 New Value: 3 (Time = 23)
Assign: <var_intNumBars> Old Value: 3 New Value: 2 (Time = 25)
Assign: <var_intNumBars> Old Value: 2 New Value: 1 (Time = 27)
Assign: <var_intNumBars> Old Value: 1 New Value: 0 (Time = 29)
```

Example Output: Random Value Generator

As can be seen in the output, the number of bars decreases by one from five every 2 seconds and the random value is reset every 10 seconds.

Test Suites

As was stated earlier, the tests are broken into categories: compiler usage, lexer, parser, semantic, and runtime. The runtime tests are those at the beginning of this section—real world applications of EDSL.

Compiler Usage

The Compiler Usage tests are used to verify that the compiler is accepting the command line parameters and opening and reading or writing files properly.

1. No Command Line Arguments.
2. Invalid Number of Command Line Arguments: 1
3. Invalid Number of Command Line Arguments: 3
4. Non-existent Source Input File
5. Invalid Source Input File Path
6. Invalid Intermediate Output File Path

Lexer

The Lexer tests are used to verify that the compiler is reading tokens and creating the proper lexemes.

1. Simple Comment
2. Nested Comments Failure
3. Nested Comments Success
4. Line Comments and Mixed Comments
5. No Main Program Failure
6. Literal: Valid Boolean (true and false)
7. Literal: Valid Integer (Zero, Decimal, Binary, Hexadecimal)
8. Literal: Invalid Integer (Zero, Decimal, Binary, Hexadecimal)
9. Declaration: Invalid Linkage
10. Declaration: Invalid Type
11. Declaration: Swapped Linkage and Type
12. Keyword Invalid
13. Missing Semicolon

Parser

The Parser tests are used to verify that the compiler is executed the rules as they should be executed and in the proper order.

1. Mathematical Precedence
2. Cascading If-statements
3. Label Outside of Main Program
4. Label Immediately Before End of Main Program
5. Multiple Lone Semicolons
6. Expressions and Operators
7. Multiple Goto Statements

Semantic

The Semantic tests are used to verify that the compiler is generating the proper warnings or errors during static semantic analysis and converting values between types properly. Most of these tests do not generate an output because of the induced errors—they are important primarily because they test the generation of error and warning messages.

1. Unassigned Variable Used in Expression
2. Automatic Variable Promotion
3. Automatic Variable Truncation
4. Invalid Left-hand-side
5. Duplicate Label
6. Duplicate Variable
7. Variable Used as Label
8. Goto Undeclared Label
9. Assignment to Undeclared Variable
10. Expression with Label
11. Assignment to Input
12. Evaluate an Output
13. Integer too Large

Testing Automation

The compilation of the EDSL compiler and all of the testing performed was automated using a Windows shell batch file. There are two limitations of this means of automated testing:

1. Programs that have infinite loops cannot be tested—or at least the infinite loop cannot be tested within these programs.
2. External input cannot be deterministically modified during the program's execution.

These limitations were determined to be acceptable limitations of the testing procedure.

There are essentially three possible steps for each test in the script:

a. Compile the EDSL source into C++ source.
b. Compile and link the C++ into an executable.
c. Run the executable.

During each of these steps, the output is captured and written to a file to preserve the results of the test.

Lessons Learned

The most important thing that I learned during throughout the course of this project is that error handling and testing for a compiler is an extremely tedious and time-consuming process that can never be started early enough. Even when the compiler was essentially done to the point that the syntax checker and code generator were complete, I was still many hours away from a finished compiler.

If I had it to do over again, I would have budgeted at least 40% of my time for error handling and testing, and I would advise any future team or individual to do the same.

Appendix

The source includes the two ANTLR grammar (.g) files and the two C++ include files used for the class template and class implementations for the EDSL variable declarations in the intermediate language.

EDSL.g

```
////////////////////////////////////////////////////////////////////////////////
//
// EDSL Grammar (Combined Lexer and Parser)
// Christopher D. Sargent
//
// This combined lexer/parser grammar is built using ANTLR 3.0.1 contained in
// ANTLRWorks 1.1.5. ANTLRWorks is used because the ANTLR 3.0.1 JAR does not
// contain the StringTemplate library. It was just easier to use ANTLRWorks,
// which does. The command line syntax to execute ANTLR on this file is:
//
// java -classpath antlrworks-1.1.5.jar org.antlr.Tool EDSL.g
//
// ANTLRWorks is available from the following web page:
//
// http://www.antlr.org/works/index.html
//
////////////////////////////////////////////////////////////////////////////////

grammar EDSL;

options
{
	output = AST;
	ASTLabelType = CommonTree;
}

tokens
{
	BOOL;
	BRACE;
	DECL;
	DECLS;
	INT;
	LABEL;
	MAIN;
	PAREN;
	PROG;
	VAR;
}

@header
{
	package edsl;

	import java.lang.Boolean;
	import java.lang.Integer;
	import java.lang.Long;
	import java.util.Hashtable;
}

@lexer::header
{
	package edsl;
}

@members
```

```
{
	/**
	 * The EDSL symbol table.
	 */
	private static Hashtable<String, EDSLSymbol> m_mapSymbols =
		new Hashtable<String, EDSLSymbol>();

	/**
	 * Returns the EDSL symbol table.
	 *
	 * @return The EDSL symbol table.
	 */
	public static Hashtable<String, EDSLSymbol> getSymbols()
	{
		return m_mapSymbols;
	}

	/**
	 * The basic EDSL symbol class.
	 */
	public abstract class EDSLSymbol
	{
		/**
		 * The flag indicating whether the symbol is declared.
		 */
		private boolean m_blnIsDeclared = false;

		/**
		 * The flag indicating whether the symbol is used.
		 */
		private boolean m_blnIsUsed = false;

		/**
		 * Real constructor
		 *
		 * @param blnIsDeclared Flag indicating whether symbol is declared.
		 */
		public EDSLSymbol(boolean blnIsDeclared)
		{
			if (true == blnIsDeclared)
			{
				// It is being declared
				m_blnIsDeclared = true;
			}
			else
			{
				// If not declared, then it must be used
				m_blnIsUsed = true;
			}
		}

		/**
		 * Returns true if the symbol is a variable.
		 *
		 * @return True if the symbol is a variable.
		 */
		public abstract boolean getIsVariable();

		/**
		 * Accessor for the flag indicating whether the symbol is declared.
		 *
		 * @return The flag indicating whether the symbol is declared.
		 */
		public boolean getIsDeclared()
		{
			return m_blnIsDeclared;
		}

		/**
		 * Mutator for the flag indicating whether the symbol is declared.
		 *
```

```
         * @param blnIsDeclared Flag indicating whether symbol is declared.
         */
        public void setIsDeclared(boolean blnIsDeclared)
        {
                m_blnIsDeclared = blnIsDeclared;
        }

        /**
         * Accessor for the flag indicating whether the symbol is used.
         *
         * @return The flag indicating whether the symbol is used.
         */
        public boolean getIsUsed()
        {
                return m_blnIsUsed;
        }

        /**
         * Mutator for the flag indicating whether the symbol is used.
         *
         * @param blnIsUsed The flag indicating whether the symbol is used.
         */
        public void setIsUsed(boolean blnIsUsed)
        {
                m_blnIsUsed = blnIsUsed;
        }
}

/**
 * The EDSL symbol sub-class for a label.
 */
public class EDSLSymbolLabel extends EDSLSymbol
{
        /**
         * Real constructor
         *
         * @param blnIsDeclared Flag indicating whether symbol is declared.
         */
        public EDSLSymbolLabel(boolean blnIsDeclared)
        {
                super(blnIsDeclared);
        }

        /**
         * Returns true if the symbol is a variable.
         *
         * @return True if the symbol is a variable.
         */
        public boolean getIsVariable()
        {
                return false;
        }
}

/**
 * The EDSL symbol sub-class for a variable.
 */
public class EDSLSymbolVariable extends EDSLSymbol
{
        /**
         * The flag indicating whether a value is assigned.
         */
        private boolean m_blnIsAssigned = false;

        /**
         * The linkage specifier string.
         */
        private String m_strLinkage = "";

        /**
         * The type specifier string.
```

```
 */
private String m_strType = "";

/**
 * Default constructor.
 *
 * It is assumed that this is a usage, not a declaration.
 */
public EDSLSymbolVariable()
{
        super(false);
}

/**
 * Real constructor.
 *
 * @param strLinkage The linkage specifier string.
 * @param strType    The type specifier string.
 */
public EDSLSymbolVariable(String strLinkage, String strType)
{
        super(true);

        m_strLinkage = strLinkage;
        m_strType = strType;
}

/**
 * Returns true if the symbol is a variable.
 *
 * @return True if the symbol is a variable.
 */
public boolean getIsVariable()
{
        return true;
}

/**
 * Accessor for the flag indicating whether a value is assigned.
 *
 * @return The flag indicating whether a value is assigned.
 */
public boolean getIsAssigned()
{
        return m_blnIsAssigned;
}

/**
 * Mutator for the flag indicating whether a value is assigned.
 *
 * @param blnIsAssigned Flag indicating whether a value is assigned.
 */
public void setIsAssigned(boolean blnIsAssigned)
{
        m_blnIsAssigned = blnIsAssigned;
}

/**
 * Accessor for the linkage specifier string.
 *
 * @return The linkage specifier string.
 */
public String getLinkage()
{
        return m_strLinkage;
}

/**
 * Mutator for the linkage specifier string.
 *
 * @param strLinkage The linkage specifier string.
```

```
 */
public void setLinkage(String strLinkage)
{
        m_strLinkage = strLinkage;
}

/**
 * Accessor for the type specifier string.
 *
 * @return The type specifier string.
 */
public String getType()
{
        return m_strType;
}

/**
 * Mutator for the type specifier string.
 *
 * @param strType The type specifier string.
 */
public void setType(String strType)
{
        m_strType = strType;
}

/**
 * Returns true if the evaluation type is an integer.
 *
 * This includes the timer type which evaluates to an integer.
 *
 * @return True if the evaluation type is an integer.
 */
public boolean getIsInt()
{
        return false == getType().equals("bool");
}

/**
 * Returns true if the variable can be assigned.
 *
 * @return True if the variable can be assigned.
 */
public boolean getCanBeAssigned()
{
        return false == getLinkage().equals("input");
}

/**
 * Returns true if the variable can be evaluated.
 *
 * @return True if the variable can be evaluated.
 */
public boolean getCanBeEvaluated()
{
        return false == getLinkage().equals("output");
}

/**
 * Returns true if the variable can be written.
 *
 * @return True if the variable can be written.
 */
public Boolean getCanBeWritten()
{
        return new Boolean((true == getLinkage().equals("output")) ||
                (true == getLinkage().equals("inout")));
}

/**
 * Returns true if the variable can be read.
```

```
                 *
                 * @return True if the variable can be read.
                 */
                public Boolean getCanBeRead()
                {
                        return new Boolean((true == getLinkage().equals("input")) ||
                                (true == getLinkage().equals("inout")));
                }
        }

        /**
         * A flag indicating that an error occurred during parsing.
         */
        private boolean m_blnError = false;

        /**
         * Returns true if an error occurred during parsing.
         *
         * @return True if an error occurred during parsing.
         */
        public boolean errorOccurred()
        {
                return m_blnError;
        }

        /**
         * Reports an exception that occurred during tree walking.
         *
         * @param e The exception that occurred.
         */
        public void reportError(RecognitionException e)
        {
                // Report the error
                super.reportError(e);

                // Set the flag indicating that an error occurred
                m_blnError = true;
        }

        /**
         * Displays an exception type as a warning.
         *
         * @param e The exception type to display the contents of.
         */
        public static void displayWarning(FailedPredicateException e)
        {
                System.err.println("Warning: line " + e.line + ":" +
                        e.charPositionInLine + " rule " + e.ruleName +
                        " predicate: {" + e.predicateText + "}?");
        }
}

/**
 * Program
 */
program
        :       d = declarationList ( '{' s = statementList '}' ) ->
                ^( PROG ^( DECLS $d ) ^( MAIN $s ) ) EOF
        ;

/**
 * Declaration
 */
declarationList
        :       ( declaration )*
        ;

declaration
        :       l = Linkage t = Type i = ID ';'
                {
                        // Look up the symbol
```

```
                        EDSLSymbol objSymbol = getSymbols().get($i.getText());

                        // See if the symbol has been declared or used
                        if (null == objSymbol)
                        {
                                // Add the symbol
                                getSymbols().put($i.getText(), new EDSLSymbolVariable(
                                        $l.getText(), $t.getText()));
                        }
                        else
                        {
                                // It was already added
                                throw new FailedPredicateException(input,
                                        "Symbol Declaration Check",
                                        "symbol cannot be declared again");
                        }
                }
                -> ^( DECL $i )
        ;

/**
 * Statement
 */
statementList
        :       ( statement )*
        ;

statement
        :       ';' -> // eliminate empty statements
        |       i = ID ':'
                {
                        // Look up the symbol
                        EDSLSymbol objSymbol = getSymbols().get($i.getText());

                        // See if the symbol has been declared or used
                        if (null == objSymbol)
                        {
                                // Add the symbol
                                getSymbols().put($i.getText(),
                                        new EDSLSymbolLabel(true));
                        }
                        else if (true == objSymbol.getIsDeclared())
                        {
                                // It was already added
                                throw new FailedPredicateException(input,
                                        "Symbol Declaration Check",
                                        "symbol cannot be declared again");
                        }
                        else
                        {
                                // Declare it
                                objSymbol.setIsDeclared(true);
                        }
                }
                -> ^( LABEL $i )
        |       assignmentExpression ';'!
        |       Goto^ i = ID ';'!
                {
                        // Look up the symbol
                        EDSLSymbol objSymbol = getSymbols().get($i.getText());

                        // See if it was already added to the symbol table
                        if (null == objSymbol)
                        {
                                // Add the symbol as being used only
                                getSymbols().put($i.getText(),
                                        new EDSLSymbolLabel(false));
                        }
                        else
                        {
                                // Make sure that it is not a variable symbol
```

```
                                if (true == objSymbol.getIsVariable())
                                {
                                        // It is a variable
                                        throw new FailedPredicateException(input,
                                                "Proper Symbol Usage",
                                                "symbol cannot be used as a label");
                                }

                                // Set that it is being used
                                objSymbol.setIsUsed(true);
                        }
                }
        |       ifStatement
        |       block
        ;

ifStatement
        options
        {
                backtrack = true;
        }
        :       If '(' e = expression ')' s1 = statement Else s2 = statement ->
                 ^( If ^( PAREN $e ) $s1 Else $s2 )
        |       If '(' e = expression ')' s = statement ->
                 ^( If ^( PAREN $e ) $s )
        ;

block
        :       '{' l = statementList '}' -> ^( BRACE $l )
        ;

/**
 * Expression
 */
assignmentExpression
        options
        {
                backtrack = true;
        }
        :       i = ID '='^ assignmentExpression
                {
                        // Look up the symbol
                        EDSLSymbol objSymbol = getSymbols().get($i.getText());

                        // See if it was already added to the symbol table
                        boolean blnIsDeclared = false;
                        if (null == objSymbol)
                        {
                                // Add the symbol as being used only
                                objSymbol = new EDSLSymbolVariable();
                                getSymbols().put($i.getText(), objSymbol);
                        }
                        else
                        {
                                // Make sure that the symbol is a variable
                                if (false == objSymbol.getIsVariable())
                                {
                                        // It is a label
                                        throw new FailedPredicateException(input,
                                                "Proper Symbol Usage",
                                                "only variables can be assigned");
                                }

                                // Set the declared flag
                                blnIsDeclared = objSymbol.getIsDeclared();
                        }

                        // Check to see if the declared flag is set
                        if (false == blnIsDeclared)
                        {
                                // It should have been added
```

```
                                throw new FailedPredicateException(input,
                                        "Symbol Declaration Check",
                                        "variable must be declared to be used");
                        }

                        // Get the variable
                        EDSLSymbolVariable objVariable =
                                (EDSLSymbolVariable) objSymbol;

                        // Make sure that variable can be assigned
                        if (false == objVariable.getCanBeAssigned())
                        {
                                // The variable cannot be assigned
                                throw new FailedPredicateException(input,
                                        "Variable Linkage Check",
                                        "variable cannot be assigned");
                        }

                        // Set that the variable is assigned
                        objVariable.setIsAssigned(true);
                }
        |       expression
        ;

expression
        :       logicalXorExpression ( '||'^ logicalXorExpression )*
        ;

logicalXorExpression
        :       logicalAndExpression ( '^^'^ logicalAndExpression )*
        ;

logicalAndExpression
        :       orExpression ( '&&'^ orExpression )*
        ;

orExpression
        :       xorExpression ( '|'^ xorExpression )*
        ;

xorExpression
        :       andExpression ( '^'^ andExpression )*
        ;

andExpression
        :       equalityExpression ( '&'^ equalityExpression )*
        ;

equalityExpression
        :       comparisonExpression ( ( '==' | '!=' )^ comparisonExpression )*
        ;

comparisonExpression
        :       shiftExpression ( ( '<' | '>' | '<=' | '>=' )^ shiftExpression )*
        ;

shiftExpression
        :       additionExpression ( ( '<<' | '>>' )^ additionExpression )*
        ;

additionExpression
        :       multExpression ( ( '+' | '-' )^ multExpression )*
        ;

multExpression
        :       notExpression ( ( '*' | '/' | '%' )^ notExpression )*
        ;

notExpression
        :       '!'^ negationExpression
        |       negationExpression
```

```
	;

negationExpression
	:	'-'^ primaryExpression
	|	primaryExpression
	;

primaryExpression
	:	'(' e = expression ')' -> ^( PAREN $e )
	|	atom
	;

/**
 * Basic atom
 */
atom
	options
	{
		backtrack = true;
	}
	:	i = ID
		{
			// Look up the symbol
			EDSLSymbol objSymbol = getSymbols().get($i.getText());

			// See if it was already added to the symbol table
			boolean blnIsDeclared = false;
			if (null == objSymbol)
			{
				// Add the symbol as being used only
				objSymbol = new EDSLSymbolVariable();
				getSymbols().put($i.getText(), objSymbol);
			}
			else
			{
				// Make sure that the symbol is a variable
				if (false == objSymbol.getIsVariable())
				{
					// It is a label
					throw new FailedPredicateException(input,
						"Proper Symbol Usage",
						"only variables can be assigned");
				}

				// Set the declared flag
				blnIsDeclared = objSymbol.getIsDeclared();
			}

			// Check to see if the declared flag is set
			if (false == blnIsDeclared)
			{
				// It should have been added
				throw new FailedPredicateException(input,
					"Symbol Declaration Check",
					"variable must be declared to be used");
			}

			// Get the variable
			EDSLSymbolVariable objVariable =
				(EDSLSymbolVariable) objSymbol;

			// Make sure that variable can be evaluated
			if (false == objVariable.getCanBeEvaluated())
			{
				// The variable cannot be assigned
				throw new FailedPredicateException(input,
					"Variable Linkage Check",
					"variable cannot be evaluated");
			}

			// Check to see if the variable is internal
```

```
                if (true == objVariable.getLinkage().equals("internal"))
                {
                        // Check to see if the variable is assigned
                        if (false == objVariable.getIsAssigned())
                        {
                                // Warning: unsafe use of a Boolean
                                displayWarning(new FailedPredicateException(
                                        input, "Variable Usage Check",
                                        "uninitialized variable used"));
                        }
                }

                // Set that the variable is used
                objSymbol.setIsUsed(true);
        }
        -> ^( VAR $i )
|       i = IntLiteral
        {
                // Get the integer string
                String strInt = $i.getText();

                // Default is a decimal value
                int intRadix = 10;

                // Check special literal m_strTypes
                if (true == strInt.startsWith("0"))
                {
                        // If the size is 1, then the value is zero
                        if (strInt.length() > 1)
                        {
                                // Binary literal
                                if ((true == strInt.startsWith("b", 1)) ||
                                        (true == strInt.startsWith("B", 1)))
                                {
                                        intRadix = 2;
                                        strInt = strInt.substring(2);
                                }

                                // Hex literal
                                else
                                {
                                        intRadix = 16;
                                        strInt = strInt.substring(2);
                                }
                        }
                }

                // Maximum value of an equivalent unsigned integer
                final long lngMax = (long) Integer.MAX_VALUE * 2 + 1;

                // Parse into a long (because it is unsigned)
                long lngValue;

                try
                {
                        // Parse the integer string into a long integer
                        lngValue = Long.parseLong(strInt, intRadix);

                        // Verify that it is within the limits for an integer
                        if (lngValue > lngMax)
                        {
                                // Number exceeds bounds
                                throw new NumberFormatException();
                        }
                }
                catch (NumberFormatException e)
                {
                        // Number exceeds bounds
                        throw new FailedPredicateException(input,
                                "Size of Integer Literal",
                                "integer literal is too large");
```

```
                        }

                        // Set the integer value
                        Integer intValue = new Integer((int) lngValue);
                        $i.setText(intValue.toString());
                }
                -> ^( INT $i )
        |       b = BoolLiteral -> ^( BOOL $b )
        ;

/**
 * Keywords
 */
If              :       'if'                                            ;
Else            :       'else'                                          ;
Goto            :       'goto'                                          ;
Linkage         :       'input' | 'output' | 'inout' | 'internal'       ;
Type            :       'bool' | 'int' | 'timer'                        ;
BoolLiteral     :       'true' | 'false'                                ;

/**
 * Lexical rules
 */
IntLiteral
        :       DecLiteral
        |       BinLiteral
        |       HexLiteral
        ;

fragment
DecLiteral
        :       '0' | ( '1' .. '9' ) ( '0' .. '9' )*
        ;

fragment
BinLiteral
        :       '0' ( 'b' | 'B' ) BinDigit+
        ;

fragment
BinDigit
        :       '0' | '1'
        ;

/**
 * Lexical rules from the ANTLR 3.0.1 documentation
 */
ID
        :       ( 'a' .. 'z' | 'A' .. 'Z' | '_' )
                ( 'a' .. 'z' | 'A' .. 'Z' | '0' .. '9' | '_' )*
        ;

fragment
HexLiteral
        :       '0' ( 'x' | 'X' ) HexDigit+
        ;

fragment
HexDigit
        :       ( '0' .. '9' | 'a' .. 'f' | 'A' .. 'F' )
        ;

WS
        :       ( ' ' | '\r' | '\t' | '\u000C' | '\n' )
                { $channel = HIDDEN; }
        ;

COMMENT
        :       '/*' ( options { greedy = false; } : . )* '*/'
                { $channel = HIDDEN; }
        ;
```

```
LINE_COMMENT
        :       '//' ~( '\n' | '\r' )* '\r'? '\n' { $channel = HIDDEN; }
        ;
```

EDSLCompiler.g

```
////////////////////////////////////////////////////////////////////////////////
//
// EDSL Grammar (Tree Parser/Compiler)
// Christopher D. Sargent
//
// This tree parser is the code generator and is built using ANTLR 3.0.1
// contained in ANTLRWorks 1.1.5. It must be built after EDSL.g is built. The
// command line syntax to execute ANTLR on this file is:
//
// java -classpath antlrworks-1.1.5.jar org.antlr.Tool EDSLCompiler.g
//
// ANTLRWorks is available from the following web page:
//
// http://www.antlr.org/works/index.html
//
// The main entry point for the EDSL compiler is also produced when this
// ANTLR source file is built.
//
////////////////////////////////////////////////////////////////////////////////

tree grammar EDSLCompiler;

options
{
        tokenVocab = EDSL;
        ASTLabelType = CommonTree;
        output = template;
}

@header
{
        package edsl;

        import java.io.File;
        import java.io.FileOutputStream;
        import java.io.PrintStream;

        import edsl.EDSLParser.EDSLSymbol;
        import edsl.EDSLParser.EDSLSymbolVariable;
}

@members
{
        /**
         * The output print stream.
         */
        private PrintStream m_objPrintStream = null;

        /**
         * Accessor for the compilation output stream.
         *
         * @return The compilation output stream.
         */
        public PrintStream getOutput()
        {
                return m_objPrintStream;
        }

        /**
         * Mutator for the compilation output stream.
         *
```

```
 * @param objStream The output stream.
 */
public void setOutput(PrintStream objStream)
{
	m_objPrintStream = objStream;
}

/**
 * A flag indicating that an error occurred in compilation.
 */
private boolean m_blnError = false;

/**
 * Returns true if an error occurred during compilation.
 *
 * @return True if an error occurred during compilation.
 */
public boolean errorOccurred()
{
	return m_blnError;
}

/**
 * Reports an exception that occurred during compilation.
 *
 * @param e The exception that occurred.
 */
public void reportError(RecognitionException e)
{
	// Report the error
	super.reportError(e);

	// Set the flag indicating that an error occurred
	m_blnError = true;
}

/**
 * The application entry point.
 *
 * @param strArgs The command line arguments.
 */
public static void main(String[] strArgs)
{
	// Make sure the right number of parameters are specified
	if (2 != strArgs.length)
	{
		// Check to see if there were any parameters specified
		if (strArgs.length > 0)
		{
			// There were, but they weren't the right number
			System.out.println("ERROR: Incorrect command line " +
				"parameters.");
		}

		// Display the syntax
		System.out.println();
		System.out.println("Syntax:");
		System.out.println();
		System.out.println("edsl.EDSLCompiler <Source> <Output>");
		System.out.println();
		System.out.println("Source - EDSL source file.");
		System.out.println("Output - C++ output file.");

		// Exit the application
		System.exit(0);
	}

	// Redirect standard error output for capture to files
	System.setErr(System.out);

	try
```

```
		{
			// Set the input file stream using the first parameter
			CharStream objInput = new ANTLRFileStream(strArgs[0]);

			// Display an initial message indicating compilation
			System.out.println("Building " + strArgs[0] + "...");

			// Create the lexer
			EDSLLexer objLexer = new EDSLLexer(objInput);

			// Get the token stream using the lexer
			CommonTokenStream objTokens = new CommonTokenStream(objLexer);

			// Create the parser
			EDSLParser objParser = new EDSLParser(objTokens);

			// Parse the token stream
			EDSLParser.program_return objReturn = objParser.program();

			// Make sure that no error occurred during scanning/parsing
			if (false == objParser.errorOccurred())
			{
				// Get the AST returned by the parser
				CommonTree objAST = (CommonTree) objReturn.getTree();
				CommonTreeNodeStream objTreeNodes =
					new CommonTreeNodeStream(objAST);

				// Create the tree parser (compiler)
				EDSLCompiler objCompiler =
					new EDSLCompiler(objTreeNodes);

				// Create the output file from the second parameter
				File objFile = new File(strArgs[1]);

				// Open the output file stream and set the print stream
				FileOutputStream objOutput =
					new FileOutputStream(objFile);
				objCompiler.setOutput(new PrintStream(objOutput));

				// Compile the program
				objCompiler.program();

				// Close the print stream
				objCompiler.getOutput().close();

				// Check for error an error within the file
				if (true == objCompiler.errorOccurred())
				{
					// File has errors, delete it
					objFile.deleteOnExit();
				}

				// Display a compile complete message
				System.out.println("Compile Finished...");
			}
		}
		catch (Exception e)
		{
			// Display the error that occurred
			System.out.println("Error: " + e.getMessage());
		}
	}
}

/**
 * Program
 */
program
	@init
	{
		// Print the header of the output file
```

```
                getOutput().println("#include <stdio.h>");
                getOutput().println("#include <time.h>");
                getOutput().println("#include \"event.h\"");
                getOutput().println("#include \"eventTimer.h\"");
                getOutput().println("int main()");
                getOutput().println("{");
        }
        @after
        {
                // Print the footer of the output file
                getOutput().println("; }");

                // The semicolon is added in case there is a label at the end
        }
        :       ^( PROG ^( DECLS declaration* ) ^( MAIN statementList ) )
        ;

/**
 * Declaration
 */
declaration
        :       ^( DECL i = ID )
                {
                        // Look up the symbol from the symbol table
                        EDSLParser.EDSLSymbolVariable objVar = (EDSLSymbolVariable)
                                EDSLParser.getSymbols().get($i.getText());

                        // Add a token to the variable name for code generation
                        String strVar = "var_" + $i.getText();

                        // Generate code for the timer declaration
                        if (true == objVar.getType().equals("timer"))
                        {
                                getOutput().println("CEventTimer " +
                                        strVar + "(\"" + strVar + "\", " +
                                        objVar.getCanBeRead().toString() + ", " +
                                        objVar.getCanBeWritten().toString() + ");");
                        }

                        // Generate code for the integer or Boolean declaration
                        else
                        {
                                // The Boolean declaration is converted to an integer
                                getOutput().println("CEvent<int> " + strVar +
                                        "(\"" + strVar + "\", " +
                                        objVar.getCanBeRead().toString() + ", " +
                                        objVar.getCanBeWritten().toString() + ");");
                        }
                }
        ;

/**
 * Statement
 */
statementList
        :       ( statement )*
        ;

statement
        :       expression
                {
                        // Generate code for the statement terminator
                        getOutput().println(";");
                }
        |       ^( LABEL i = ID
                {
                        // Look up the symbol
                        EDSLSymbol objSymbol = EDSLParser.getSymbols().
                                get($i.getText());

                        // Make sure that it is used
```

```
                        if (true == objSymbol.getIsUsed())
                        {
                                // Generate code for the label
                                getOutput().println("lab_" + $i.getText() + ":");
                        }
                        else
                        {
                                // Warning: label not used
                                EDSLParser.displayWarning(new
                                        FailedPredicateException(input,
                                        "Label Usage Check",
                                        "label declared but not used"));
                        }
                }
                )
        |       ^( Goto i = ID
                {
                        // Look up the symbol
                        EDSLSymbol objSymbol = EDSLParser.getSymbols().
                                get($i.getText());

                        // Check to see if the symbol was declared
                        if (false == objSymbol.getIsDeclared())
                        {
                                // It wasn't
                                throw new FailedPredicateException(input,
                                        "Proper Symbol Usage",
                                        "label was not declared");
                        }

                        // Generate code for the goto statement
                        getOutput().println("goto " + "lab_" + $i.getText() + ";");
                }
                )
        |       ifStatement
        |       block
        ;

ifStatement
        :       ^( If
                {
                        // Generate code for the if statement
                        getOutput().print("if (");
                }
                ^( PAREN expression )
                {
                        // Generate code for the end of expression
                        getOutput().println(")");
                }
                statement
                ( Else
                {
                        // Generate code for the else statement
                        getOutput().println("else");
                }
                statement
                )?
                )
        ;

block
        :       ^( BRACE
                {
                        // Generate code for the beginning of the block
                        getOutput().println("{");
                }
                statementList
                {
                        // Generate code for the end of the block
                        getOutput().println("}");
                }
```

```
                )
        ;

/**
 * Expression
 */
expression returns[boolean blnIsInt]
        options
        {
                backtrack = true;
        }
        :       ^( '=' i = ID
                {
                        // Generate the code for the variable usage
                        getOutput().print("var_" + $i.getText() + " = ");
                }
                e = expression
                {
                        // Look up the symbol from the symbol table
                        EDSLParser.EDSLSymbolVariable objVar = (EDSLSymbolVariable)
                                EDSLParser.getSymbols().get($i.getText());

                        // Set the type flag
                        $blnIsInt = objVar.getIsInt();

                        // See if the variable and expression are not the same type
                        if ($blnIsInt != $e.blnIsInt)
                        {
                                // See if this is a promotion
                                if (true == $blnIsInt)
                                {
                                        // Generate a warning indicating promotion
                                        EDSLParser.displayWarning(
                                                new FailedPredicateException(input,
                                                "Operand Usage Check", "promotion of " +
                                                "expression from Boolean to integer"));
                                }

                                // Or a truncation
                                else
                                {
                                        // Generate a warning indicating truncation
                                        EDSLParser.displayWarning(
                                                new FailedPredicateException(input,
                                                "Operand Usage Check", "truncation of " +
                                                "expression from integer to Boolean"));
                                }
                        }
                }
                )
        |       ^( o = ( '|' | '^' | '&' )    // i|b = i|b o i|b
                e1 = expression
                {
                        // Generate code for the operator
                        getOutput().print(" " + $o.getText() + " ");
                }
                e2 = expression
                {
                        // Check to see if the expressions are of different types
                        if ($e1.blnIsInt != $e2.blnIsInt)
                        {
                                // The expression is an integer
                                $blnIsInt = true;

                                // Generate a warning indicating promotion
                                EDSLParser.displayWarning(new FailedPredicateException(
                                        input, "Operand Usage Check", "promotion of " +
                                        "expression from Boolean to integer"));
                        }
                        else
                        {
```

```
                        // The type is whatever the expression type is
                        $blnIsInt = $e1.blnIsInt;
                }
        }
        )
|       ^( o = ( '==' | '!=' )         // b = i|b o i|b
        e1 = expression
        {
                // Generator code for the operator
                getOutput().print(" " + $o.getText() + " ");
        }
        e2 = expression
        {
                // The resultant must be a Boolean
                $blnIsInt = false;

                // Check to see if the expressions are of different types
                if ($e1.blnIsInt != $e2.blnIsInt)
                {
                        // Generate a warning indicating promotion
                        EDSLParser.displayWarning(new FailedPredicateException(
                                input, "Operand Usage Check", "promotion of " +
                                "expression from Boolean to integer"));
                }
        }
        )
|       ^( o = ( '||' | '^^' | '&&' )         // b = b o b
        e1 = expression
        {
                // Make sure that the first expression is a Boolean
                if (true == $e1.blnIsInt)
                {
                        // Generate a warning indicating truncation
                        EDSLParser.displayWarning(new FailedPredicateException(
                                input, "Operand Usage Check", "truncation of " +
                                "expression from integer to Boolean"));
                }

                // Generate code for the operator
                getOutput().print(" " + $o.getText() + " ");
        }
        e2 = expression
        {
                // Make sure that the second expression is a Boolean
                if (true == $e2.blnIsInt)
                {
                        // Generate a warning indicating truncation
                        EDSLParser.displayWarning(new FailedPredicateException(
                                input, "Operand Usage Check", "truncation of " +
                                "expression from integer to Boolean"));
                }

                // The expression is always a Boolean
                $blnIsInt = false;
        }
        )
|       ^( o = ( '<' | '>' | '<=' | '>=' )           // b = i o i
        e1 = expression
        {
                // Check to see if the first expression is a Boolean
                if (false == $e1.blnIsInt)
                {
                        // Generate a warning indicating promotion
                        EDSLParser.displayWarning(new FailedPredicateException(
                                input, "Operand Usage Check", "promotion of " +
                                "expression from Boolean to integer"));
                }

                // Generate code for the operator
                getOutput().print(" " + $o.getText() + " ");
        }
```

```
        e2 = expression
        {
                // Check to see if the second expression is a Boolean
                if (false == $e2.blnIsInt)
                {
                        // Generate a warning indicating promotion
                        EDSLParser.displayWarning(new FailedPredicateException(
                                input, "Operand Usage Check", "promotion of " +
                                "expression from Boolean to integer"));
                }

                // The expression is always a Boolean
                $blnIsInt = false;
        }
        )
|       ^( o = ( '<<' | '>>' | '+' | '-' | '*' | '/' | '%' ) // i = i o i
        e1 = expression
        {
                // Check to see if the first expression is a Boolean
                if (false == $e1.blnIsInt)
                {
                        // Generate a warning indicating promotion
                        EDSLParser.displayWarning(new FailedPredicateException(
                                input, "Operand Usage Check", "promotion of " +
                                "expression from Boolean to integer"));
                }

                // Generate code for the operator
                getOutput().print(" " + $o.getText() + " ");
        }
        e2 = expression
        {
                // Check to see if the second expression is a Boolean
                if (false == $e2.blnIsInt)
                {
                        // Generate a warning indicating promotion
                        EDSLParser.displayWarning(new FailedPredicateException(
                                input, "Operand Usage Check", "promotion of " +
                                "expression from Boolean to integer"));
                }

                // The expression is always an integer
                $blnIsInt = true;
        }
        )
|       ^( '!'          // b = o b
        {
                // Generate code for the operator
                getOutput().print("!");
        }
        e = expression
        {
                // Check to see if the expression is an integer
                if (true == $e.blnIsInt)
                {
                        // Generate a warning indicating truncation
                        EDSLParser.displayWarning(new FailedPredicateException(
                                input, "Operand Usage Check", "truncation of " +
                                "expression from integer to Boolean"));
                }

                // The expression is always a Boolean
                $blnIsInt = false;
        }
        )
|       ^( '-'          // i = o i
        {
                // Generate code for the operator
                getOutput().print("-");
        }
        e = expression
```

```
                {
                        // Check to see if the expression is a Boolean
                        if (false == $e.blnIsInt)
                        {
                                // Generate a warning indicating promotion
                                EDSLParser.displayWarning(new FailedPredicateException(
                                        input, "Operand Usage Check", "promotion of " +
                                        "expression from Boolean to integer"));
                        }

                        // The expression is always an integer
                        $blnIsInt = true;
                }
                )
        |       ^( PAREN
                {
                        // Generate code for the beginning of the block
                        getOutput().print("(");
                }
                e = expression
                {
                        // The type flag is the flag from the expression
                        $blnIsInt = $e.blnIsInt;

                        // Generate code for the terminator of the block
                        getOutput().print(")");
                }
                )
        |       ^( VAR i = ID )
                {
                        // Look up the symbol from the symbol table
                        EDSLParser.EDSLSymbolVariable objVar = (EDSLSymbolVariable)
                                EDSLParser.getSymbols().get($i.getText());

                        // Set the expression type flag
                        $blnIsInt = objVar.getIsInt();

                        // Generate the code for the variable usage
                        getOutput().print("var_" + $i.getText());
                }
        |       ^( INT i = IntLiteral )
                {
                        // Integer literal
                        $blnIsInt = true;

                        // Generate code for the integer literal
                        getOutput().print($i.getText());
                }
        |       ^( BOOL b = BoolLiteral )
                {
                        // Boolean literal
                        $blnIsInt = false;

                        // Generate code for the Boolean literal
                        getOutput().print($b.getText().equals("true") ? "1" : "0");
                }
        ;
```

event.h

```
//////////////////////////////////////////////////////////////////////////////////////
//! \file  event.h
//! \brief Declaration and implementation of the CEvent template class.
//!
//! \section RevisionHistory Revision History
//! \verbatim
//! Author          Date        Description
//! ------          ----        -----------
```

```
//! Chris Sargent   2007-11-10  Initial version.
//! \endverbatim
//!
//! \addtogroup EDSL
////////////////////////////////////////////////////////////////////////////////

#ifndef EVENT_H_
#define EVENT_H_

#include <fstream>
#include <iostream>
#include <stdlib.h>
#include <string>
#include <time.h>

////////////////////////////////////////////////////////////////////////////////
//! \class   CEvent
//! \ingroup EDSL
//!
//! \brief
//! Provides a template for all variables types except the timer.
////////////////////////////////////////////////////////////////////////////////

template <class T>
class CEvent
{
public:
        // Real constructor.
        CEvent(const std::string &strName = "", bool blnRead = false,
                bool blnWrite = false);

        // Copy constructor.
        CEvent(const CEvent &objOrig);

        // Assignment operator.
        const CEvent &operator=(const CEvent &objOrig);

        // Assignment operator.
        virtual const T operator=(const T &typValue);

        // Destructor.
        virtual ~CEvent();

        // Returns the value.
        virtual operator const T ();

protected:
        // Reads the external parameter.
        void read(bool blnFirst = false);

        // Writes the external parameter.
        virtual void write();

        // Returns the time of the last read, set, or polled.
        virtual const time_t lastSet() const;

        //! A flag indicating that the parameter is read externally.
        bool m_blnRead;

        //! A flag indicating whether the parameter value is set.
        bool m_blnSet;

        //! A flag indicating that the parameter is written externally.
        bool m_blnWrite;

        //! The filename associated with reading or writing the parameter.
        std::string m_strFileName;

        //! The name associated with the external representation of the parameter.
        std::string m_strName;
```

```
        //! The value of the parameter.
        T m_typValue;

        //! The time of the construction of the variable.
        time_t m_timConstruct;

        //! The time that the parameter was last polled.
        time_t m_timLastPoll;

        //! The time that the parameter was last read or assigned.
        time_t m_timLastSet;
};

////////////////////////////////////////////////////////////////////////////////
//! \fn CEvent<T>::CEvent(const std::string &strName, bool blnRead, bool blnWrite)
//!
//! \brief
//! Real constructor.
//!
//! Initializes the member variables and reads the parameter from the external source,
//! if it is an input or inout parameter.
//!
//! \param[in]  strName    The name associated with the external representation.
//! \param[in]  blnRead    A flag indicating that the parameter is read externally.
//! \param[in]  blnWrite   A flag indicating that the parameter is written externally.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////

template <class T>
inline CEvent<T>::CEvent(const std::string &strName, bool blnRead, bool blnWrite) :
        m_blnRead(blnRead),
        m_blnSet(false),
        m_blnWrite(blnWrite),
        m_strFileName(strName + ".txt"),      // File extension to make editing easier
        m_strName(strName),
        m_typValue(),
        m_timConstruct(time(NULL)),
        m_timLastPoll(0),
        m_timLastSet(0)
{
        // Read the parameter from the external source if it is an input or inout
parameter
        if (true == m_blnRead)
        {
                read(true);
        }
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEvent<T>::CEvent(const CEvent &objOrig)
//!
//! \brief
//! Copy constructor.
//!
//! Copies the member variables and reads the parameter from the external source,
//! if it is an input or inout parameter.
//!
//! \param[in]  objOrig   The original object to copy from.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////

template <class T>
inline CEvent<T>::CEvent(const CEvent &objOrig) :
        m_blnRead(objOrig.m_blnRead),
        m_blnSet(objOrig.m_blnSet),
        m_blnWrite(objOrig.m_blnWrite),
        m_strFileName(objOrig.m_strFileName),
        m_strName(objOrig.m_strName),
        m_typValue(objOrig.m_typValue),
```

```
        m_timConstruct(objOrig.m_timConstruct),
        m_timLastPoll(objOrig.m_timLastPoll),
        m_timLastSet(objOrig.m_timLastSet)
{
        // Read the parameter from the external source if it is an input or inout
parameter
        if (true == m_blnRead)
        {
                read(true);
        }
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEvent<T>::operator=(const CEvent &objOrig)
//!
//! \brief
//! Assignment operator.
//!
//! Assigns the member variables and reads the parameter from the external source,
//! if it is an input or inout parameter.
//!
//! \param[in]  objOrig   The original object to assign from.
//!
//! \return A reference to the assigned object.
////////////////////////////////////////////////////////////////////////////////

template <class T>
inline const CEvent<T> &CEvent<T>::operator=(const CEvent &objOrig)
{
        // Verify that the object to assign from is not this object
        if (this == &objOrig)
        {
                return *this;
        }

        // Assign the value
        m_blnSet = true;
        m_typValue = objOrig.m_typValue;
        m_timLastSet = time(NULL);

        // Read the parameter from the external source if it is an input or inout
parameter
        if (true == m_blnRead)
        {
                read(true);
        }

        return *this;
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEvent<T>::operator=(const T &typValue)
//!
//! \brief
//! Assignment operator.
//!
//! Assigns the value member variable and writes it to the external location if the
//! parameter is an output or an inout.
//!
//! \param[in]  typValue   The value to assign to this parameter.
//!
//! \return A reference to the value assigned to this parameter.
////////////////////////////////////////////////////////////////////////////////

template <class T>
inline const T CEvent<T>::operator=(const T &typValue)
{
        // Get the original value
        T typOrig = operator const T();

        // Verify that the value has changed as a result of this assignment
```

```
        if ((typOrig != typValue) || (false == m_blnSet))
        {
                // Write the name of the parameter, its old value, and its new value
                std::cout << "Assign: <" << m_strName << ">";
                std::cout << " Old Value: " << typOrig;
                std::cout << " New Value: " << typValue;
                std::cout << " (Time = " << time(NULL) - m_timConstruct << ")";
                std::cout << std::endl;

                // Assign the value
                m_typValue = typValue;

                // The time last read is used in assignment as well
                m_timLastSet = time(NULL);

                // The value is now considered set
                m_blnSet = true;

                // Write it externally if the parameter is an output or an inout
                if (true == m_blnWrite)
                {
                        write();
                }

                // Return the new value
                return m_typValue;
        }

        // Return the original value
        return typOrig;
}

////////////////////////////////////////////////////////////////////////////////////////
//! \fn CEvent<T>::~CEvent()
//!
//! \brief
//! Destructor.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////////////

template <class T>
inline CEvent<T>::~CEvent()
{
}

////////////////////////////////////////////////////////////////////////////////////////
//! \fn CEvent<T>::operator const T()
//!
//! \brief
//! Returns the value.
//!
//! Attempts to read the value if it is either an input or inout parameter and it is time
//! to poll it and its new value flag is set.
//!
//! \return The value.
////////////////////////////////////////////////////////////////////////////////////////

template <class T>
inline CEvent<T>::operator const T ()
{
        // Attempt to read the parameter
        if (true == m_blnRead)
        {
                read();
        }

        return m_typValue;
}

////////////////////////////////////////////////////////////////////////////////////////
```

```
//! \fn CEvent<T>::read(bool blnFirst)
//!
//! \brief
//! Reads the external parameter.
//!
//! The parameter is read from an external source if it is time to poll it and either
//! the new input flag is set or this is the first read (initialization).
//!
//! \param[in]  blnFirst   A flag indicating whether this is the initialization.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////

template <class T>
inline void CEvent<T>::read(bool blnFirst)
{
	// Get the current time
	time_t timCurrent = time(NULL);

	// Check to see if it is time to poll the value
	if ((timCurrent - m_timLastPoll) >= 1)
	{
		// Set the time of last poll
		m_timLastPoll = timCurrent;

		// Open the file containing the value
		std::ifstream objInput(m_strFileName.c_str());

		// Read the flag indicating whether or not there is new data to read
		bool blnSet;
		objInput >> blnSet;

		// Check whether there is new input or whether this is the initialization
		if ((true == blnSet) || (true == blnFirst))
		{
			// Write the name of the parameter and its old value
			std::cout << "Read: <" << m_strName << ">";
			std::cout << " Old Value: " << m_typValue;

			// Read the value and close the input
			objInput >> m_typValue;
			objInput.close();

			// Set the time of the last read
			m_timLastSet = timCurrent;

			// Write the old value
			std::cout << " New Value: " << m_typValue;
			std::cout << " (Time = " << time(NULL) - m_timConstruct << ")";
			std::cout << std::endl;

			// Write value back for the purpose of reseting new input flag
			write();
		}
		else
		{
			objInput.close();
		}
	}
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEvent<T>::write()
//!
//! \brief
//! Writes the external parameter.
//!
//! The parameter is written to the file without a time check.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////
```

```
template <class T>
inline void CEvent<T>::write()
{
	// Open the file that contains the value as output
	std::ofstream objOutput(m_strFileName.c_str());

	// Output the new data flag first if this parameter is an inout parameter
	if (m_blnRead)
	{
		objOutput << false << ' ';
	}

	// Output the value and close the file
	objOutput << m_typValue;
	objOutput.close();
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEvent<T>::lastSet() const
//!
//! \brief
//! Returns the time of the last set.
//!
//! \return The time of the last set.
////////////////////////////////////////////////////////////////////////////////

template <class T>
inline const time_t CEvent<T>::lastSet() const
{
	return m_timLastSet;
}

#endif
```

eventTimer.h

```
////////////////////////////////////////////////////////////////////////////////
//! \file  eventTimer.h
//! \brief Declaration and implementation of the CEventTimer template class.
//!
//! \section RevisionHistory Revision History
//! \verbatim
//! Author          Date        Description
//! ------          ----        -----------
//! Chris Sargent   2007-11-17  Initial version.
//! \endverbatim
//!
//! \addtogroup EDSL
////////////////////////////////////////////////////////////////////////////////

#ifndef EVENTTIMER_H_
#define EVENTTIMER_H_

#include <time.h>

#include "event.h"

////////////////////////////////////////////////////////////////////////////////
//! \class   CEventTimer
//! \ingroup EDSL
//!
//! \brief
//! Provides a container for timer events.
////////////////////////////////////////////////////////////////////////////////

class CEventTimer : public CEvent<int>
{
```

```
public:
        // Real constructor.
        CEventTimer(const std::string &strName = "", bool blnRead = false,
                bool blnWrite = false);

        // Copy constructor.
        CEventTimer(const CEventTimer &objOrig);

        // Assignment operator.
        const CEventTimer &operator=(const CEventTimer &objOrig);

        // Assignment operator.
        virtual const int operator=(const int &intValue);

        // Destructor.
        virtual ~CEventTimer();

        // Returns the value.
        virtual operator const int();

protected:
        // Writes the external parameter.
        virtual void write();
};

////////////////////////////////////////////////////////////////////////////////
//! \fn CEventTimer::CEventTimer(const std::string &strName, bool blnRead, bool blnWrite)
//!
//! \brief
//! Real constructor.
//!
//! Calls the base constructor.
//!
//! \param[in]  strName    The name associated with the external representation.
//! \param[in]  blnRead    A flag indicating that the parameter is read externally.
//! \param[in]  blnWrite   A flag indicating that the parameter is written externally.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////

inline CEventTimer::CEventTimer(const std::string &strName, bool blnRead,
        bool blnWrite) :
        CEvent<int>(strName, blnRead, blnWrite)
{
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEventTimer::CEventTimer(const CEventTimer &objOrig)
//!
//! \brief
//! Copy constructor.
//!
//! Calls the base constructor.
//!
//! \param[in]  objOrig   The original object to copy from.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////

inline CEventTimer::CEventTimer(const CEventTimer &objOrig) :
        CEvent<int>(objOrig)
{
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEventTimer::operator=(const CEventTimer &objOrig)
//!
//! \brief
//! Assignment operator.
//!
//! Calls the base assignment operator.
```

```
//!
//! \param[in]  objOrig   The original object to assign from.
//!
//! \return A reference to the assigned object.
////////////////////////////////////////////////////////////////////////////////

inline const CEventTimer &CEventTimer::operator=(const CEventTimer &objOrig)
{
	// Verify that the object to assign from is not this object
	if (this == &objOrig)
	{
		return *this;
	}

	// Calls the base assignment operator
	CEvent<int>::operator=(objOrig);

	return *this;
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEventTimer::operator=(const int &intValue)
//!
//! \brief
//! Assignment operator.
//!
//! Calls the base assignment operator.
//!
//! \param[in]  intValue   The value to assign to this parameter.
//!
//! \return A reference to the value assigned to this parameter.
////////////////////////////////////////////////////////////////////////////////

inline const int CEventTimer::operator=(const int &intValue)
{
	return CEvent<int>::operator=(intValue);
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEventTimer::~CEventTimer()
//!
//! \brief
//! Destructor.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////

inline CEventTimer::~CEventTimer()
{
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEventTimer::operator const int()
//!
//! \brief
//! Returns the value.
//!
//! Converts from absolute to a relative time remaining.
//!
//! \return The value.
////////////////////////////////////////////////////////////////////////////////

inline CEventTimer::operator const int()
{
	int intTime = CEvent<int>::operator const int() - (int) (time(NULL) - lastSet());
	return intTime > 0 ? intTime : 0;
}

////////////////////////////////////////////////////////////////////////////////
//! \fn CEventTimer::write()
//!
```

```
//! \brief
//! Writes the external parameter.
//!
//! The parameter is written to the file without a time check.
//!
//! \return nothing
////////////////////////////////////////////////////////////////////////////////

inline void CEventTimer::write()
{
        // Compute the time remaining
        int intTime = CEvent<int>::operator const int() - (int) (time(NULL) - lastSet());

        // Set the time remaining
        CEvent<int>::operator=(intTime > 0 ? intTime : 0);

        // Write the parameter
        CEvent<int>::write();
}

#endif
```

Bibliography

[1] Aho, et al. *Compilers Principles, Techniques, & Tools. Second Edition*. Pearson Education: Boston, MA. 2007.

[2] Conway, et al. “Pencil: A Petri Net Specification Language for Java.” 2002.

[3] Gosling, et al. *The Java Language Specification. Third Edition*. Addison-Wesley: Boston, MA. 2005.

[4] Kernighan, Brian and Dennis Ritchie. *The C Programming Language. 2nd Edition*. Prentice-Hall: Stoughton, MA. 1988.