State Machine Generator Language Reference Manual

COMS W4115

Oliver Zhou
ohz2101

March 13, 2014

Contents

1 Introduction

The State Machine Generator Language, or SMGL, is a programming language focused on facilitating development utilizing finite state machines.

Finite state machines, or FSM, are powerful mathematical models that can simulate and solve nearly any type of problem in a clear and understandable way. There are many areas of study in this area, and are considered part of the field of Automata Theory.

Using only high level statements, SMGL allows for anyone to quickly and easily create a simulation of a state machine without worrying about implementation details. The State Machine Generator Language is designed to facilitate rapid simulation and prototyping of the logic of state machines, and can be used to test the logic of your system.

2 Program

A program in SMGL is a file consisting of several major sections. The first section will define the signals involved with the input and output to the state machines. The second section will define the various states in the state machine to be tested. The third section can define the actions and inputs to the state machine, or optionally be left blank if further low level of modification of the resulting C is desired without the C providing any simulated output.

3 Execution Results

The result of the execution of a SMGL program would be a C file that represents the state machine defined in SMGL. The optional actions portion of a SMGL program also allow for rapid testing and viewing of the state machine being designed in SMGL if the C file is compiled and executed.

4 Lexical Conventions

Commenting Code

Comments are delineated as everything between the string “//” and the new line character, effectively meaning that the double backslash prevents the remainder of the line from being read.

```
 // commented out
STATE: run_test_idle(TMS) // this text is commented out
{
….
}
```

Keywords

Identifiers reserved by the language

CTRL_IN and *CTRL_OUT*, short for control, is used to define an input variable that controls the transitions from state to state, and to control the output at each state. Can be declared only with upper case characters

SIGNALS, used to create a section to pre-define all of the inputs and outputs that the state machine will be using boolean style logic that translates well to the systems that we are replicating. More complex things such as values can be represented by a combination of boolean values.

STATES, used to create a section to define all states of the state machine being created

ACTIONS, used to define a section that allows for simulating

DEFAULT_STATE, used to define the starting state

VERBOSE, used to generate more verbose outputs in the C generated

In addition to those above, there is a selection of keywords used in loops and boolean logic operations that are reserved.

Boolean operation operators

These are used to perform boolean logic
Table 1. Boolean Operators

Operator Name	**Symbol**	**Example**
Logical negation	!	!x is used when we want to match x = 0
Logical AND	&&	a&&b will expect both a and b to match up
Logical OR	ll	a or b will get a positive result

Table 2. Other symbols reserved for use by SMGL

Operator Name	**Symbol**	**Example**

State define start	:	statename: is used at the beginning to define a state
Parathesis	()	Used for order of operations
standard closing statement	;	The semicolon lets the parser know the definition has ended for a loop etc

Loop keywords
Used to control the program structure, the main type of structure will be the if loop behaves like many other programming languages. The if loop is used during definition of states as well. Given an input, the if loop will determine the next state, and if nothing happens, the current state will remain on that state.

Table 3. Loop keywords

Operator Name	**Symbol**	**Example**
if	if	if (!x) next_state; The if statement will keep
else	else	else after the semicolon of the if statement will give another opportunity for branching

Constants

In SMGL, constants will be in the form of booleans, which will be represented by 0 and 1, with 0 being false, and 1 being true.

Simulated Actions

In the actions portion of a SMGL program, the actions of the various states are defined in array format. All inputs need to have a matching number of input states, in the form of 0, 1, or X (don't care).
Example:
input_signal_1 = [0,0,0,1];
input_signal_2 = [1,X,1,0];

4. Syntax

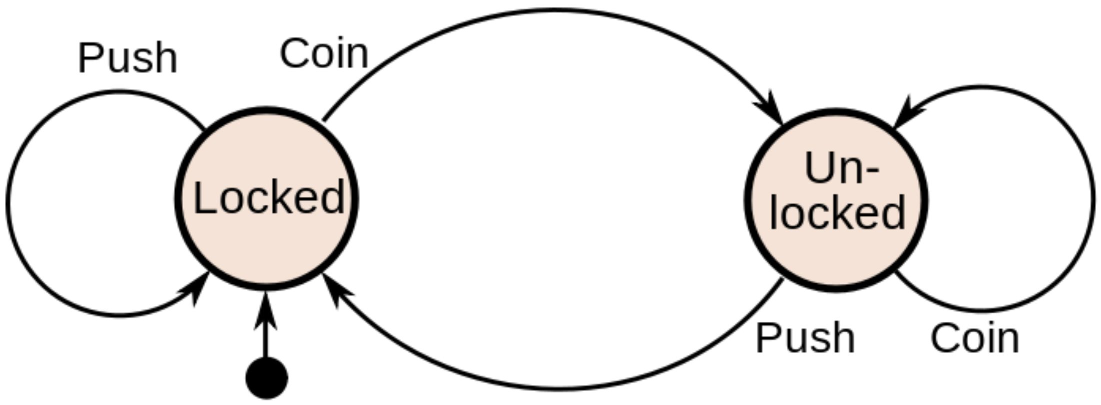

Figure 1 Example of Turnstile state machine (source: Wikipedia)
This figure is used to assist in demonstrating the syntax of a program

Machine Definition and Instantiation

Signals are defined first in its own group in a bracketed segment after the keyword SIGNALS is called. All signals are expected to be boolean style and will be used in both controlling state transitions and used in determining the actions performed at each state.

```
SIGNALS
{
        CTRL_IN Y
        CTRL_OUT X
}
```

Example fitting the example machine

```
SIGNALS
{
 CTRL_IN COIN; //Coin detection
 CTRL_IN PUSH; //Push gate Detection
 CTRL_OUT LOCK; //True being locked, false being unlocked
}
```

States are then defined next. Each state will be declared with lower case characters for differentiation from the signals. Each state is required to have one if statement to determine action for state to state transitions. An else statement is used if the state does not loop back into itself. All state definitions will fall under the brackets of the overall STATES keyword.

```
STATES
{
locked:
        if (coin) unlocked;
                //push is ignored because it does not affect the locked gate
unlocked:
        if(push) locked;
                //after a push, the gate is locked
                //coins are not needed to unlock the gate
}
```

The final section is separated by the ACTIONS keyword. This is where the chain of inputs to the state would be put in, which would also be simulated in the resulting C code after execution of the program. Starting default state needs to be defined here.

```
ACTIONS
{
        //DEFAULT_STATE needs to be defined
        DEFAULT_STATE =  locked;
        //actions happen here
        coin=[0,0,0,0,1,0]
        push=[X,X,X,X,X,1]
        VERBOSE;
}
```

6. Program Execution

The program file will be in an ASCII based file with the file ending of .smgl.
The file after it compiles will then be a C file that simulates the structures defined in the .smgl file.

If the ACTIONS portion of the program is defined properly, then code will also be generated in C that will print out the state of the state machine after the inputted signals are given to the program.

i.e., based on the state machine in Figure 1, if the following code

```
coin=[0,0,0,0,1,0];
push=[X,X,X,X,X,1] ;
```

is provided in the .smgl file, the C program, when compiled and executed, will print out the resulting status.

```
Current State : Locked
```

If certain flags are set, all state transitions and the state of each signal can be printed

```
…. //earlier state printouts
Current State : Locked
Coin = 1
Push = X
Current State : Unlocked
Coin = 0
Push = 1
Current State : Locked
```