State Machine Generator Language Proposal

COMS W4115
Oliver Zhou
ohz2101
February 11, 2014

Section 1. Motivation of Language

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

In many Digital Logic courses that early engineers and computer scientists take, fundamental understanding of the computer is introduced through the concept of finite state machines. However, these 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.

While it is possible to re-create state machine logic in many other languages, it is cumbersome and difficult to read at a glance. In fact, trying to prototype a state machine in a hardware description language, it is very easy to get caught up in minute details instead of the logic of the problem you're trying to tackle. SMGL will solve those issues by allowing for very easy to read descriptions of states and the transitions between states, allowing for even the layman to pick up the language and begin creating intricate finite state machines with mere cursory experience in the subject.

The language will have the ability to create both the ability to simulate Moore and Mealy type state machines, with the various states having the ability to provide output.

Section 2. Description of Language

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.
Input can be provided to the function that simulate the behavior of the variables that control state transition, and the output of the system at each state can be defined to a mixture of the current state that the program is in, as well as the current state of all variables and stored values, depending on what the desired output is.

Section 3. Syntax

The majority of the syntax of the language will be very similar to C, and other than SMGL specific keywords generally all other keywords will be identical to those found in C.

Important SMGL Keywords
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

STATE, used to define an object that describes a particular state in the state machine

SIGNALS, used to create a section to pre-define all of the inputs and outputs that the state machine will be using

How States are Handled:
States are treated as objects, defined with the variable that controls the transition from state to state. The state to state transitions are also defined in the state variable. For example, this sample code shows that the transition CTRL TMS.

```
STATE: test_logic_reset(TMS)
{
        if(TMS=1)
                {
                        CURRENT_STATE=test_logic_reset
                        //output
                }
        else
                {
                        CURRENT_STATE=run_test_idle
                        //output
                }
}
```

After states are defined, the main function will control the remainder of the logic involved with coding in SMGL. In the main function, input will be seamlessly handled from standard input, and output can be controlled similarly.

Comments:
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
{
….
}

Section 4. Example Program:

JTAG TAP controller state machine implementation:
One of the most important state machines that many computer engineers will work with is the JTAG State Machine. It would be easy to implement on the SMGL, and would allow for the programmer to understand the state machine's logic. For this example, the exact output at each state, which goes into another controller, is commented out for ease of understanding the basic principles of the language, but would be changing the state of bits to allow for various registers inside of the device that implemented JTAG to be read and controlled.

This is an image of the TAP Controller

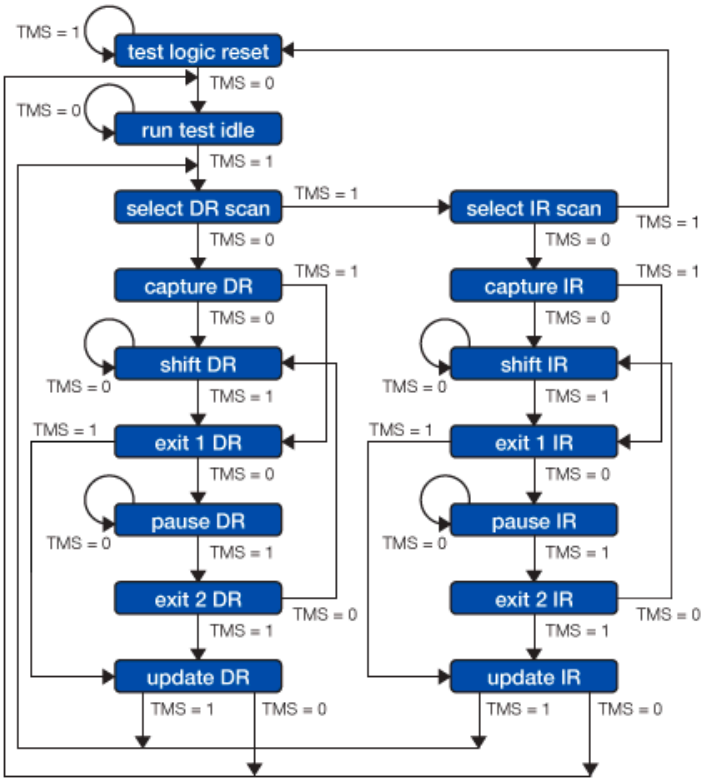

Figure 1. JTAG TAP Controller
Source: http://www.xjtag.com/support-jtag/jtag-technical-guide.php

//JTAG_TAP_Controller.smgl

```
//initial input/output signals
SIGNALS
{
	CTRL_IN TMS;
	CTRL_OUT TAP_OUTPUT_SIGNALS; //Jtag TAP Controller output signals
(simplified)
}


//initial state definitions
STATE: test_logic_reset(TMS)
{
	if(TMS=1)
		{
			CURRENT_STATE=test_logic_reset
			//output TAP_OUTPUT_SIGNALS is modified to reflect how the
state changes its output
		}
	else
		{
			CURRENT_STATE=run_test_idle
			//output TAP_OUTPUT_SIGNALS is modified to reflect how the
	state changes its output
		}
}


STATE: run_test_idle(TMS)
{
	if(TMS=1)
		{
			CURRENT_STATE=select_DR_scan
			//output TAP_OUTPUT_SIGNALS is modified to reflect how the
state changes its output
		}
	else
		{
			CURRENT_STATE=run_test_idle
			//output TAP_OUTPUT_SIGNALS is modified to reflect how the
	state changes its output
		}
}
STATE: select_DR_scan(TMS)
{
	if(TMS=1)
		{
			CURRENT_STATE=select_IR_scan
			//output TAP_OUTPUT_SIGNALS is modified to reflect how the
		state changes its output
```

```
		}
	else
		{
			CURRENT_STATE=capture_DR
			//output TAP_OUTPUT_SIGNALS is modified to reflect how the
	state changes its output
		}
}
STATE: select_IR_scan(TMS)
{
	if(TMS=1)
		{
			CURRENT_STATE=test_logic_reset
			//output TAP_OUTPUT_SIGNALS is modified to reflect how the
state changes its output
		}
	else
		{
			CURRENT_STATE=capture_IR
			//output TAP_OUTPUT_SIGNALS is modified to reflect how the
	state changes its output
		}
}
STATE: shift_IR(TMS)
... //cut to save space on paper, continues the state machine links
STATE:exit_1_IR(TMS)
…
STATE:pause_IR(TMS)
…
STATE:exit_2_IR(TMS)
…
STATE:update_IR(TMS)
…
STATE:capture_DR(TMS)
…
STATE:shift_DR(TMS)
…
STATE:exit_1_DR(TMS)
…
STATE:pause_DR(TMS)
…
STATE:exit_2_DR(TMS)
…
STATE:update_DR(TMS)


main() //main that allows for manipulation and use of the previously defined states
{
	//Slightly fluid, data can be accessed and displayed in any format
```

```
        //set initial state of state machine
        CURRENT_STATE=test_logic_reset(TMS)
        //request input
        printf("Request input of TMS\n, Type "end" when completed")
        if((input=in())!="end")
        {
        //store input
        TMS=input;
        printf("Current State is : %s, Current Status of TMS is : %c", Current Output of
State Machine is %o", CURRENT_STATE, tms[], TAP_OUTPUT_SIGNALS);
        }
}
```