Finite State Machine Creation And Editing Utility

Kyle Wall

Joshua Wascom

10/26/2008

Table of Contents

Overview of Important Terms:

When saved to the hard drive, each FSM is stored in an **FSM file**. This file is reloaded if when a user wishes to interact with a previously constructed FSM.

The **Names** data structure holds references to and metadata about FSM's that are currently being edited, and holds references to their component states. This allows the display of a Windows Explorer-like hierarchy so that a user can select an FSM, a state, or a transition by name. The Names structure is implemented as a Singleton.

An **FSM** data structure represents a finite state machine that has been constructed by the user. It contains information about its unique identifier in the Names structure as well as references to all of its component states, its start state, and its end state.

Each **state** data structure represents one state in one of the user's FSMs. Each state holds a reference to its parent FSM, references to every transition leading out of it, and information about any state logic.

State logic changes variables of the FSM in order to affect the final results of traversing the machine.

Each **transition** contains information about which state it leads to, how many times it has been traversed, and any guards in place on it.

Guard conditions ensure that certain conditions (specified by the user) are in place before allowing traversal of their parent transitions.

Major Goals:

G1: Provide users with the ability to create and edit finite state machines in a two-dimensional visual environment.

G2: Allow users to perform basic file operations on complete finite state machines such as saving, loading, and printing.

G3: Test overall validity of finite state machines to ensure that basic necessary conditions are being met. Examples of this include ensuring that there is one and only one initial state, that every state is reachable, and that there is an ending state.

G4: Allow user to specify sample data and to observe results of this input both in terms of the state-by-state traversal and the final state reached.

List of User Activities:

A1: Creating, loading, and saving FSM files.

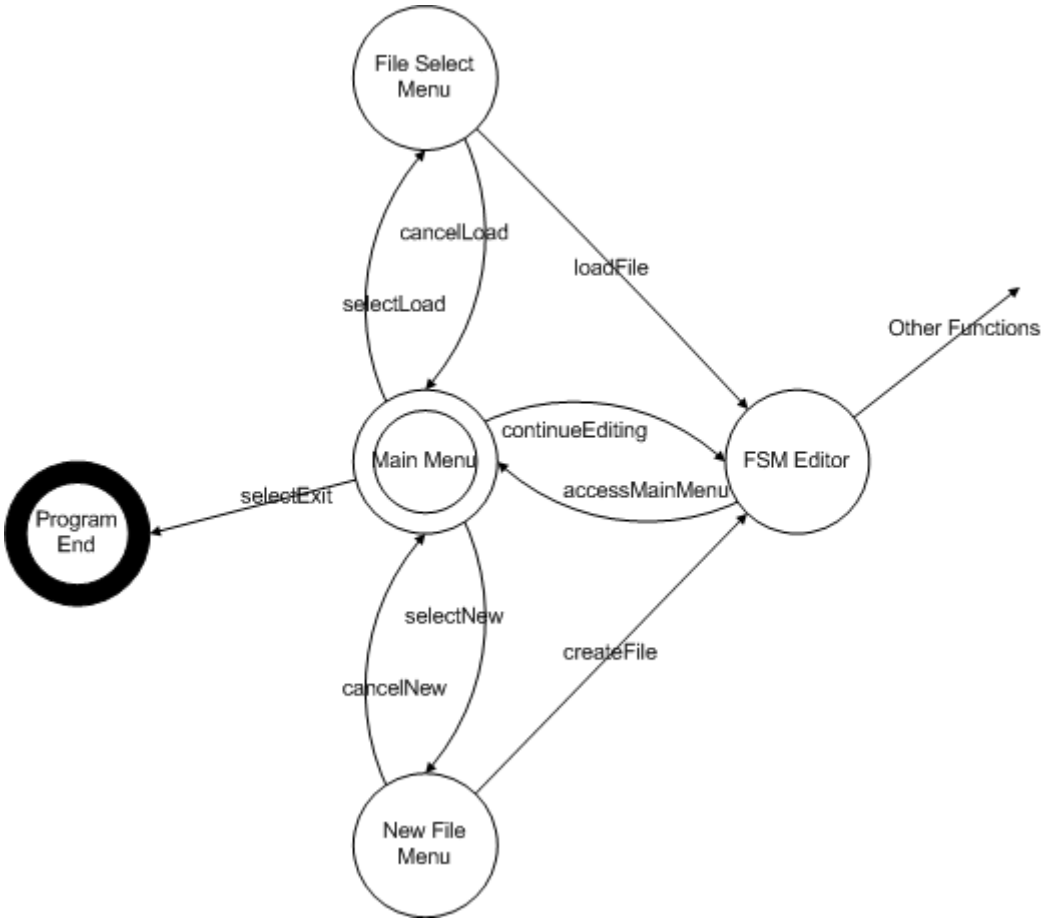
A2: Adding States, State Logic, Transitions, and Guard Conditions to an FSM.

A3: Traverse FSMs with sample data to determine validity of input and resultant final state

Major Groups of Users:

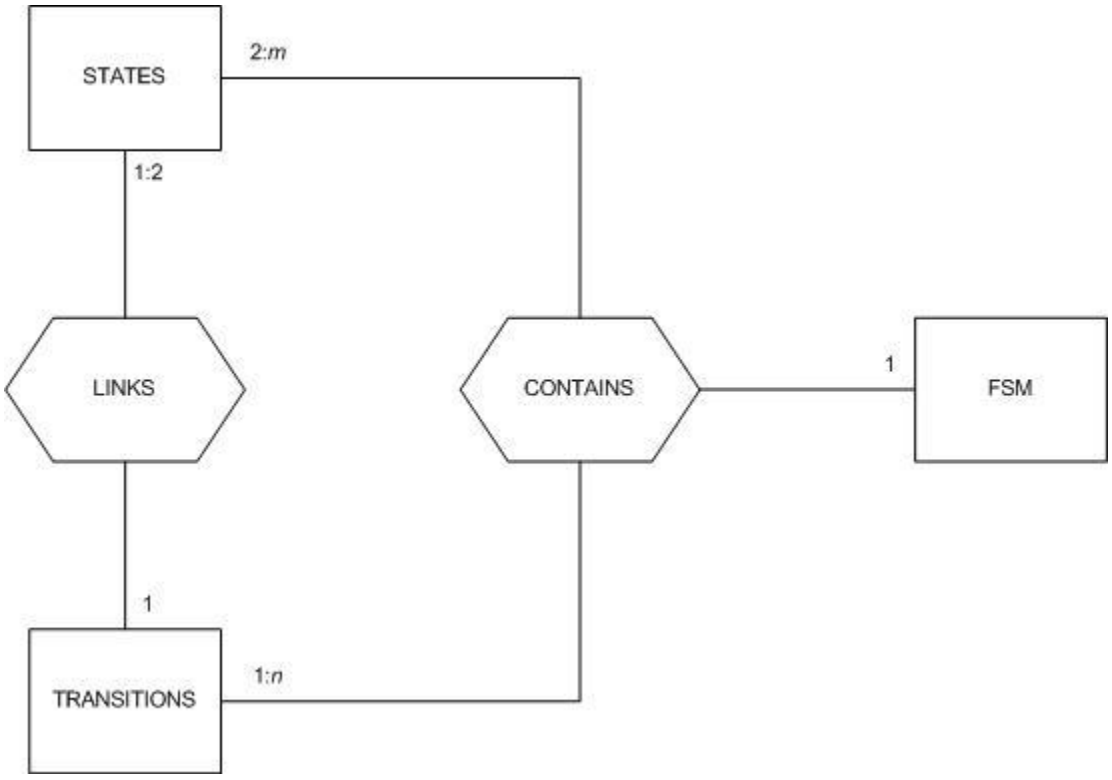
Our product will be intended for use by software developers, engineers, and anyone else who works with systems that can be usefully modeled by a finite state machine. The only type of direct user is the constructor of the FSM.

Finite State Machine for Main Menu



Expanded Finite State Machine for FSM Editor

ERDs



FSM(FSMName, StartState, EndState)

State(StateName, FSMName, Info, Data)

Transition(TransitionName, Weight, FromState, NextState, Counter)

Using the FSM Editor:

Building an FSM in the editor will consist a number of steps.

When editing an FSM, the user will select the object they desire to place, either State Object or Transition Object, from a toolbar.

States:

The State Object will allow the user to place a state onto the grid. Upon placing the state onto the grid, a properties menu will appear, allowing the user to specify a state name and any state logic. By default, a generic name is generated (by a method specified later), and the logic property is set to null.

Transitions:

The transition object will allow the user to place a transition from one state to itself or another state. Upon placing the transition object onto the grid, a properties menu will appear, allowing the user to specify a name for the transition and any guard conditions. Again, by default, a generic name is generated, and the guard conditions are set to null. Editing the guard state will invoke a check of valid Boolean logic.

Generic Names:

When a new state is created, a quick query of the Names data structure will be made. This allows the Names structure to generate a new generic name for the state (The generic Name is the first letter of the name of the FSM followed by 2 digits 0-99, in the event of more than 100 states, The first letter of the FSM will be doubled. EX. (A01…A99, AA01…). Upon generation of the name, the state item will be pointed to by a list in the Names structure, allowing for easy access later.

Tree View:

On the right-hand side of the work area will be a splash screen of the Names data structure. The splash screen will contain a tree view of all the names of FSMs, States, and Transitions. Clicking on an item in this view will cause the item to be located in the work area and highlighted. From this selection you can edit any of the previously noted properties.

Guards:

When a guard is created, the string will be parsed to check for correctness of Boolean logic. If the logic is correct, the guard will be accepted as part of the transition. If even one transition is set to have a guard then the FSM will go into logical mode. In logical mode, each transition will be required to have a guard and each state will be required to have logic. An example of guard logic would be x==2, meaning that the transition could be used only if the variable x is equal to 2.

State Logic:

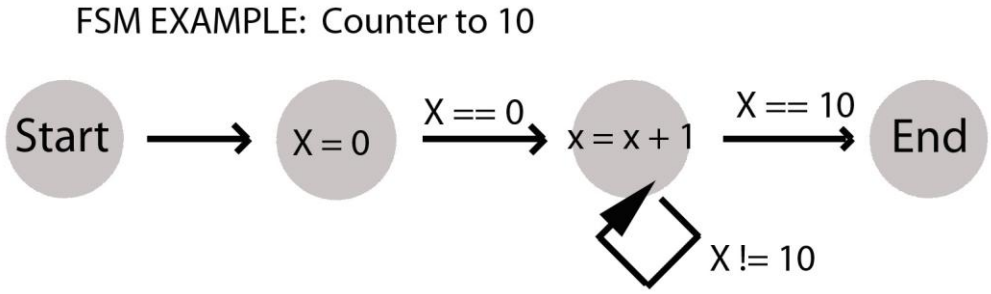
When state logic is created, the string will be parsed for errors and then added to the state. At current build, we are planning to only allow for for simple manipulation during states. An example would be x = x + 1. This state logic would take the variable x, add 1 to it, and store it back into the variable x.

The Run Command:

To RUN the FSM the program will check the FSM for completeness. It will throw errors if the FSM is not complete (That being unreachable states, No Start State, or No end state.) If the FSM is in logic mode, it will then check to make sure that each state has state logic and that each transition has Guards. After Checking, if the FSM is in logic mode, it will traverse the FSM, one item at a time at a pace set by the user, be it 1 second per

transition or x seconds per transition.
An output will be generated for a logically run FSM that will show the variables at each transition state and the overall ending of the FSM running.

Created FSM Example:



This is an example FSM with logic that a user could create on our editor. Notice that each transition (with the exception of the start transition) has a guard, and that each state is either a start state, an end state, or a logic state. When running this FSM, the error checker would first check that each transition has a correct guard, and that each state has a valid logic statement. The Editor would then step from the start state to the first state and set a variable *x* equal to 0. It would then check the guard state to move. The guard state would be satisfied (since *x* is indeed equal to zero) and the FSM would make the transition to the next state. In the next state, x would be set to *x* + 1. In this case, *x* would become 1. The guard states would then be checked. *x* equal to 10 would fail and its transition would not be followed. It would then check the next guard state. X not equal to 10. This would satisfy the guard. The FSM would make the transition to the next state (which happens to be the previous state) and perform the state function again. It would repeat in that fashion until x equaled 10. When it did, it would move to the end state and the program would finish.