SLICING OF EXTENDED FINITE STATE MACHINES

by

KAUSHIK ATCHUTA

B.Tech., Jawaharlal Nehru Technological University, 2012

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2014

Approved by:

Major Professor
Dr. Torben Amtoft

Copyright

Abstract

An EFSM (Extended Finite State Machine) is a tuple (S, T, E, V) where S is a finite set of states, T is a finite set of transitions, E is a finite set of events, and V is a finite set of variables.
Every transition t in T has a source state and a target state, both in S.

There is a need to develop a GUI which aids in building such machines and simulating them so that a slicing algorithm can be implemented on such graphs. This was the main idea of Dr. Torben Amtoft, who has actually written the slicing algorithm and wanted this to be implemented in code.

The project aims at implementing a GUI which is effective to simulate and build the graph with minimum user effort. Poor design often fails to attract users. So, the initial effort is to build a simple and effective GUI which serves the purpose of taking input from the user, building graphs and simulating it.

The scope of this project is to build and implement an interface so that the users can do the following in an effective way:

- Input a specification of an EFSM
- Store and later retrieve EFSMs
- Displaying an EFSM in a graphical form
- Simulating the EFSM
- Modify an EFSM
- Implement the slicing algorithm

All the above mentioned features must be integrated into the GUI and it should only fail if the input specification is wrong.

Table of Contents

List of Figures

Acknowledgements

I would like to express my sincere gratitude to my major professor Dr. Torben Amtoft for trusting in my abilities and providing me with an opportunity to work under his guidance.

I extend my thanks to my committee members Dr. Daniel Andresen and Dr. Mitchell Neilsen for their kind assistance and constant guidance.

I am especially grateful to my family and friends for all their love, encouragement and support.

Finally, I bow in reverence to the almighty but for whose blessings nothing can turn into reality.

Chapter 1 - Project Description

Introduction

An extended finite state machine is generally associated with states, transitions and a set of conditions. An EFSM (M) is a tuple (S, T, E, V) where S is a finite set of states, T is a finite set of transitions, E is a finite set of events, and V is a finite set of variables. A transition can be termed as an "if condition". The transition is fired only when the condition is satisfied. This actually transforms the machine from current state to the next state. Thus, every transition t which $\in$ T has both a source state and a target state. Both the source state S(t) and the target state T(t) $\in$ S. All the trigger conditions of the corresponding transitions have boolean results. The boolean expression of a transition is called a *guard* which is denoted as G(t). It's enabling events is denoted E(t) which is either a singleton or an empty set.

Terms in EFSM

All states in the set S are atomic. A *self-looping* transition is defined as a transition 't' whose source state and destination state are the same i.e. source(t) = destination(t).

An EFSM may have multiple transitions that have the same source state. All such transitions are termed as *siblings*. A transition is said to be a successor of another transition if the source state of the first transition is the target state of the second transition. This can be simply termed as follows - a transition t' is a successor of the transition t if source(t') = target(t).

A state *s* is said to be an exit state if it has no further outgoing transitions. A transition *t* is said to be the final transition if the target state of this transition is the exit state.

A transition t is said to be an ε-transition if it doesn't have a label i.e. it should neither have a guard nor an action. During simulation of an EFSM, we assume that the system remains unchanged when the evaluating expression is not satisfied.

Thus, computations within an EFSM occur on transitions rather than on states.

The EFSM has a *store*, which maps variables to values. The domain of these values is not specified.

Motivation

The need for implementing this project is to have a tool which generates a finite state machine in graphical form taking the input from the user. There are tools which generate graphs based on given input but what makes this problem more interesting is that all such tools take files as input and generate graphs but here, the user gives the input dynamically through the GUI. The problem occurs while building the graph. We have to check for all the triggered conditions before simulating the machine as a graph.

The current solution to this problem is also improvised. Besides generating the graph, the tool simulates and also implements the slicing algorithm on the finite state machine.

Chapter 2 - Literature Survey

Introduction

[1] An EFSM is a graphical representation of a system that has distinct states and a set of transitions between those states. The system is in exactly one state at any given time. The transition brings the system from one valid state to another i.e. the transitions are either completely executed or nothing at all. No partial fulfillment of transitions. If a transition occurs, then it brings the system from an existing valid state to a new valid state. If the transition fails then the system is left unchanged.

EFSMs are generally viewed as Non-deterministic Finite Automata (NFA) or Finite State Automaton (FSA) but EFSMs are different from FSAs. EFSMs have stores that maps variables to values. The labels of an EFSM are way more complex than that of a FSA. The labels is an event, condition or guard in addition to the actions.

Uses of an EFSM

[1] EFSMs are widely used to model system behavior at a higher level of abstraction. They are used to model dynamic behavior of applications. They are extensively used to model completely executable systems. There are many embedded systems whose behavior is fully specified using EFSMs.

Slicing

The process of slicing out a sub-EFSM or sub-component of an EFSM to isolate that portion of the EFSM is termed as slicing.

Slicing has been in research for more than 30 years now and is used in many software engineering applications. Though it has been in research for a considerable time now, there has been very little progress in the field of Slicing EFSMs.

Slicing an EFSM rewires the EFSM. Program slicing takes an EFSM and a slicing criteria as input and generates a sliced EFSM as output.

Program slicing removes as many states or transitions as possible by respecting the specified slicing criteria.

The effectiveness of slicing can be determined using the metric 'length of the text'. We may count the number of states but since the computation on EFSMs takes place on transitions, me count the number of transitions. Sometimes, the number of unique transitions may also serve as a very good metric.

[1] Dependence may be defined using the concepts of -

- Maximal Path
- Sink-bounded Path

In the paper by Kelly Androutsopoulos et al, Maximal Path, Sink-bounded path and Control Sink were defined as:

"A maximal path is any path that terminates in a final transition, or is infinite.

A path π is a sink-bounded path if either π contains a final transition or there exists a control sink K such that π contains every transition from K infinitely often.

A Control Sink in an EFSM is a set of transitions K that forms a strongly connected component (SCC) such that, for each transition t in K each successor of t is also in K."

But for this project Dr. Amtoft takes another approach.

Amorphous Slicing

The slices of an EFSM constructed using the amorphous slicing depends on the Dependence analysis and the slicing criteria. Amorphous slicing is also termed as a graph-based slicing that eliminates unnecessary transitions and nodes. This slicing generates an output slice that is not the sub-graph of the original.

In the paper by Kelly Androutsopoulos et al, the slicing criterion is defined as follows:

"A slicing criterion for an EFSM is a pair (t, V) where transition $t \in T$ and a variable set V which is a subset of Var. It refers to the store value immediately after the execution of the action contained in transition t."

Chapter 3 - Description Of Approaches To Solve The Problem

To achieve the above goals, I started off by building a Form application in C#. All the code is written in C#. I have chosen C# because it is relatively easy to build a GUI in C# using the Visual Studio IDE.

First Approach

The initial GUI design is shown below:

Figure 3.1 Initial Design

With this design, the user had to specify all the conditions in the text box including the number of states, transitions, guards and actions. This was not friendly enough as it involves lot of manual work from the user. The user has also been provided with another option to build the graph. The "File" menu item has a sub-menu item called "Open". When the user clicks on "Open", it prompts the user to select a document from the file browser. It has to be a text document and all the text in the document must be in the format which is similar to that of text input so that it can be read and fed as input to the machine. This was not very feasible because all the text files had to have text in the following format.

The numbers below denote the line numbers in a text file:

1. Number of states
2. Names of all the states (comma separated fields)

3. Initial State
4. Number of transitions
5. Name of each transition (comma separated fields)
6. Condition
7. Final State

I had used the above format so that I could easily parse the input and build my adjacency list. The adjacency list has the source state as the key and the rest as a list of strings.

Second Approach

It is not the best option to take the input as a text file. The GUI should also allow user to enter the input. Then, I worked further on the GUI. As a result, I have made several changes to the GUI to facilitate the user.

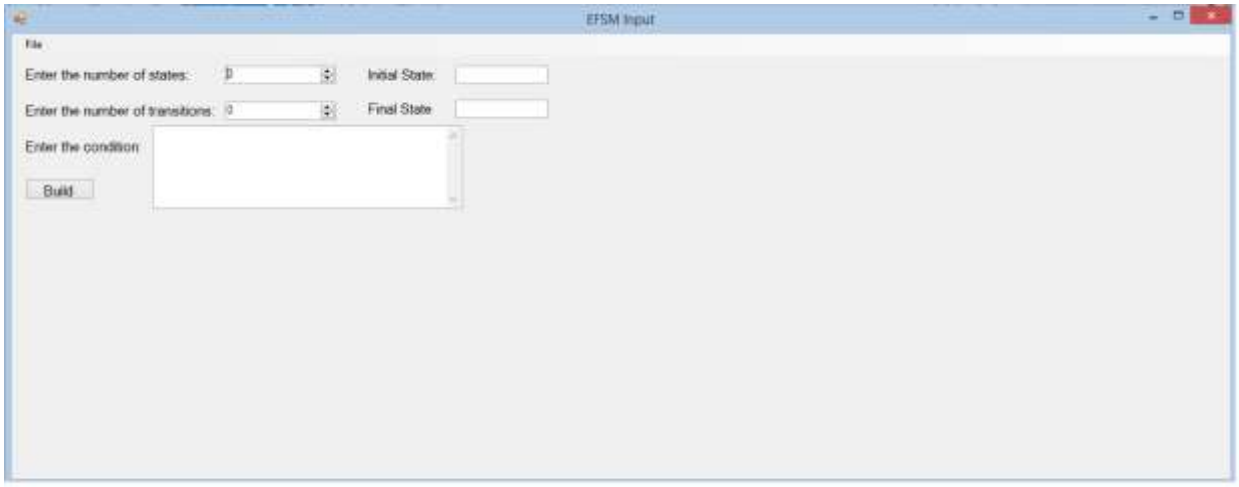

Figure 3.2 GUI Improvement

This GUI has two numeric up down fields, three text boxes, one menu item and a button.

The "File" menu item has two sub-menu items namely "Open" and "Save". When the "Open" file is clicked, it prompts you to select a file from the file browser. When a text document is selected, all the fields are auto filled with the text in the file and generating the graph is just a click away.

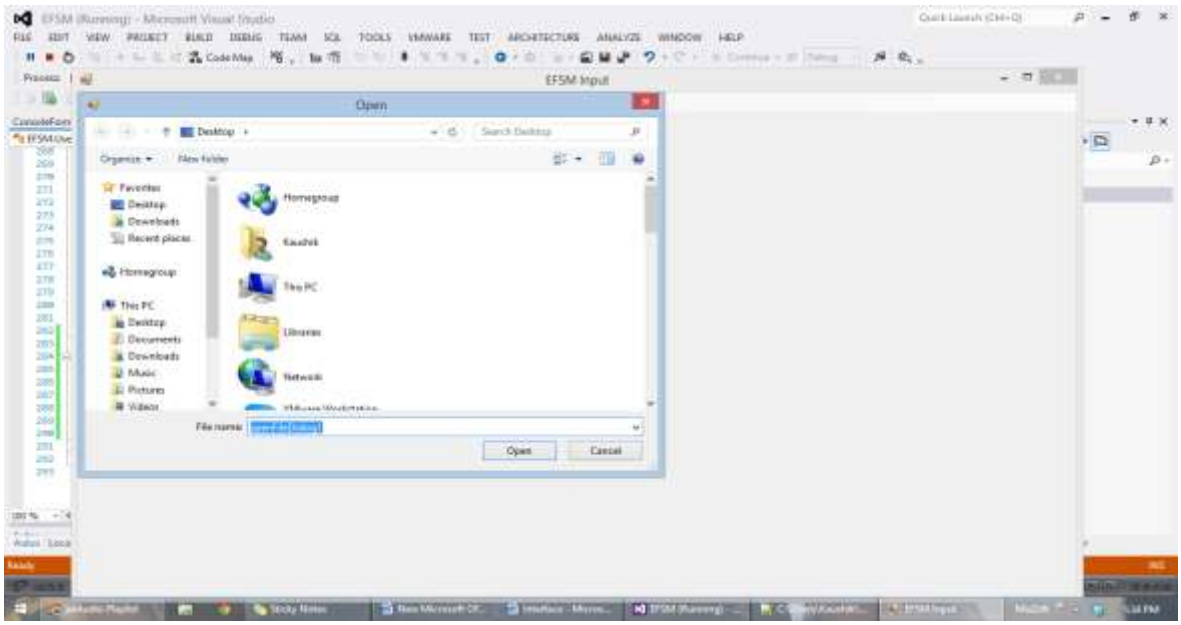

Figure 3.3 File Input

The other option to build a graph is to feed in input through the GUI directly. The "Save" button prompts the user to save all of its contents in a text file in the desired location.

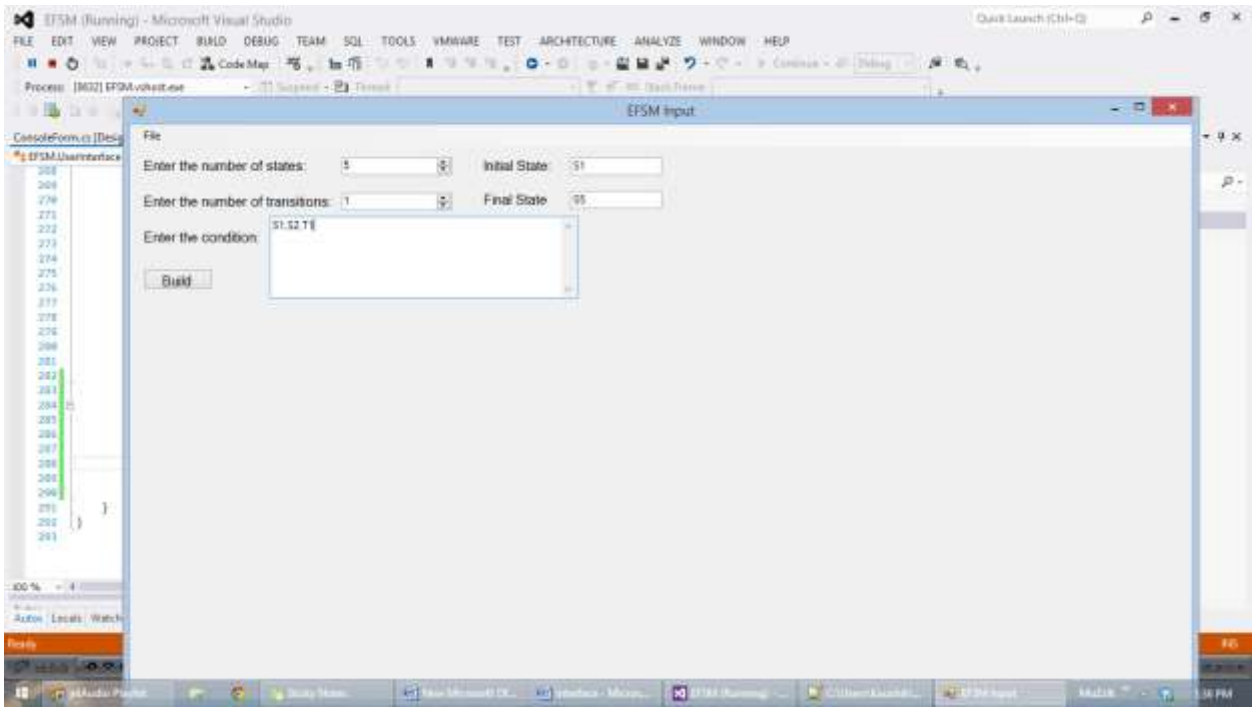

Figure 3.4 Input through the GUI

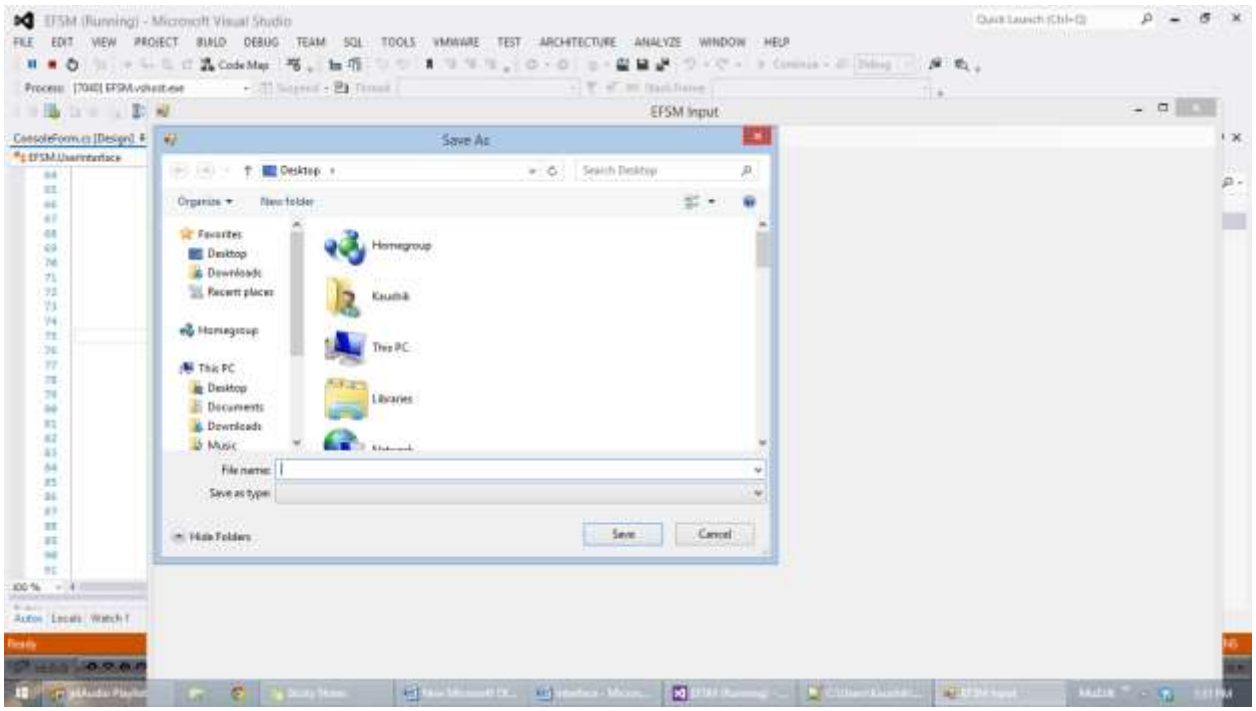

Figure 3.5 Saving the file

All the contents are saved in the text file. Below is the screen shot on how the given input is saved as a text file.

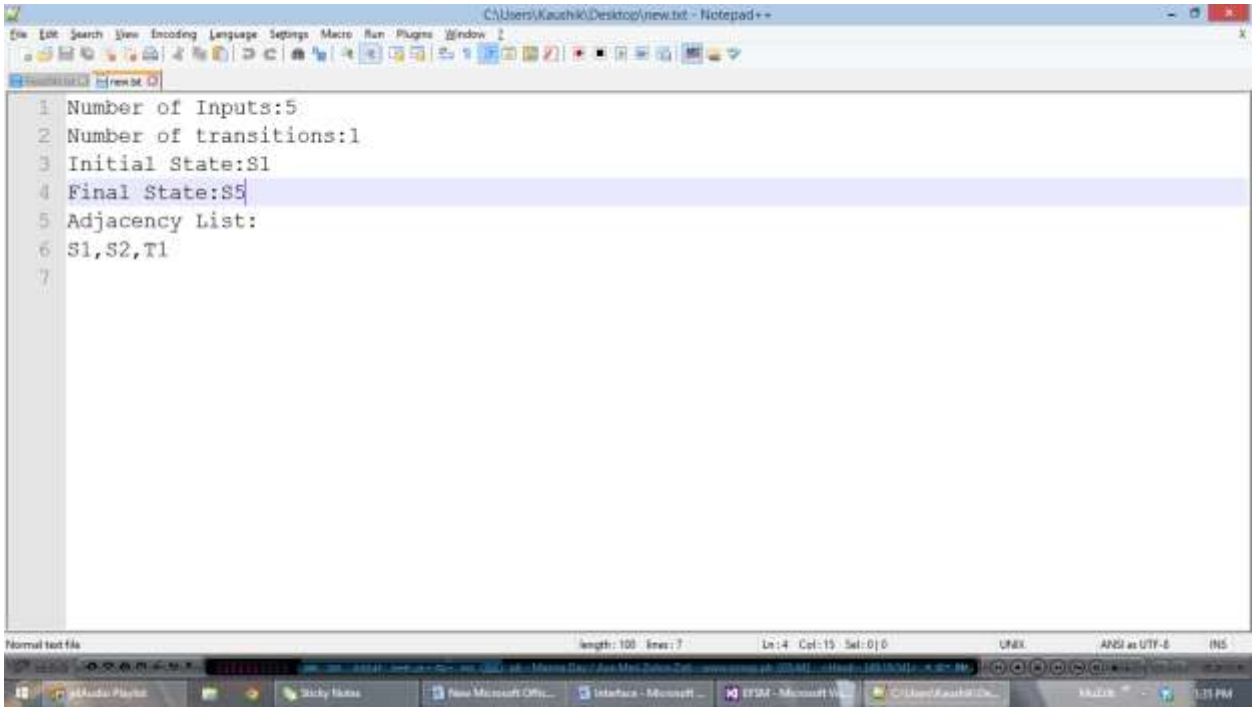

Figure 3.6 Saved text file

Simultaneously I was working on generating the graph. Initial attempt was to build or draw rectangles. I have been writing methods using the Graphics class to draw rectangles directly on the form which represent the states of the machine.

```
private void Rectangle(int x, int y, int width, int height)

{

        //int x = 200;

        Pen myPen;

        myPen = new Pen(Color.Black, 2);

        Graphics formGraphics = this.CreateGraphics();

        formGraphics.DrawRectangle(myPen, new Rectangle(x, y, width, height));

        myPen.Dispose();

        formGraphics.Dispose();

}
```

This method takes four parameters. The first two parameters are of type integer which determine the coordinates of the rectangle to be drawn. The next two parameters are the width and height of the rectangle which remain constant throughout.

The number of times this method is called is equal to the value given as input in the "Number of states" text box. This generates rectangles equal to the number of the states given as input by the user. Extra care has been taken regarding the positioning of such states or rectangles. All the rectangles are separated by an equal space. These rectangles are placed adjacent to each other and when it reaches the end of the form, it goes to the next line.

The next step is to represent state names within these rectangles. I have written a method to achieve this.

```
private void DrawString(int i, int x, int y)
{
        x += 4;
        y += 3;
        Graphics formGraphics = this.CreateGraphics();
        string drawString = "S" + i.ToString();
        Font drawFont = new Font("Verdana", 12);
        SolidBrush initialBrush = new SolidBrush(Color.Green);
        SolidBrush drawBrush = new SolidBrush(Color.Black);
        SolidBrush finalBrush = new SolidBrush(Color.Red);
        if (uxInitialState.Text == drawString)
        {
           Font initialFont = new Font("Verdana", 13, FontStyle.Bold);
           formGraphics.DrawString(drawString, initialFont, initialBrush, x, y);
        }
        else if (uxFinalState.Text == drawString)
        {
           Font finalFont = new Font("Verdana", 13, FontStyle.Bold);
           formGraphics.DrawString(drawString, finalFont, finalBrush, x, y);
        }
```

```
        else
        {
            //float a = 210;
            formGraphics.DrawString(drawString, drawFont, drawBrush, x, y);
        }
        drawFont.Dispose();
        initialBrush.Dispose();
        finalBrush.Dispose();
        drawBrush.Dispose();
        formGraphics.Dispose();
}
```

The above code gives the desired result. The string 'drawString' keeps track of all the names. All the state names begin with a 'S' and an integer starting from 1 is appended to S. The initial state name is written in Green, the final state in Red and all the other states in Black.

At this stage, we have all the states ready. The next and the most important task is to generate lines between these states whenever the condition satisfies. The following method is written to draw lines between the states but it has many flaws.

```
private void Line(int x, int y)
{ /*
        Pen myPen = new Pen(Color.Black);
        Graphics formGraphics = this.CreateGraphics();
```

```
            formGraphics.DrawLine(myPen, 245, 210, 320, 210);
            myPen.Dispose();
            formGraphics.Dispose();
           */
            using (var p = new Pen(Color.FromArgb(190, Color.Black)))
            {
                p.StartCap = LineCap.Round;
                p.EndCap = LineCap.ArrowAnchor;
                p.CustomEndCap = new AdjustableArrowCap(3, 3);
                //p.DashStyle = DashStyle.Dash;
                p.DashCap = DashCap.Triangle;
                var graph = this.CreateGraphics();
                graph.DrawLine(p, new Point(x, 210), new Point(y, 210));
            }
}
```

The above code draws directed horizontal lines but it is very tough to pass the co-ordinates of both starting and ending point for this method as I am not creating objects. The parameters 'x' and 'y' are the start and end points of the line. The parameter 'x' is the sum of the value of the starting coordinate of the rectangle and it's width. The parameter 'y' is the sum of the value of 'x' and the spacing between the adjacent rectangle. This method only generates straight lines and there will be many cases where we might need arcs. One other difficulty is that it strikes through the states that come in the way of these lines. To avoid this behavior, some code has to be written which takes care of this but this can be achieved only when I create objects for both drawing

rectangles and for writing it's name inside the rectangle. This has not been done as it takes lot of effort for a simple task.

At this stage, the GUI works properly and when the "Build" button is hit, the graph is generated. But as of now, the graph is generated based on the value of the number of states. I have written a method which draws a rectangle (representing a state). I have also written a method which actually writes or draws the name of the state within that rectangle. These methods work perfectly fine. I have also highlighted the text of the Initial State in "Green" and that of Final State in "Red" to distinguish from the rest of the states. Based on a few calculations, I am now always able to place the states in an orderly manner. The number of states per line doesn't exceed 5. The major challenge here is to draw lines between states which represent the transitions. I have also written a method which draws lines between states by keeping track of the coordinates but most of the times, we may need to draw arcs between states which are not adjacent to each other.

I had worked for more than two weeks to get this working but there were still issues that are not totally answered. I had to make sure that there is minimal intersection between transitions. Also that the transitions should not pass thorough the states. This was something more challenging because I haven't created objects for each state instead I have been drawing them whenever necessary. One solution which I figured out for this is that the transitions should automatically deviate when they encounter a state. This was something which I attempted in JavaScript but this was more challenging in C#.

Then I was trying to find a possible solution for this. I was looking at all the possible methods to draw graphs. Then I have found something useful. There is a tool called GraphViz. This tool actually takes in a file with a '.gv' extension and generates a graph. This is as simple as running one command in the terminal.

So, I have made changes to my code accordingly. I haven't disturbed the GUI. The GUI still remains the same but what significantly has changed is how we store the information in the text file. Initially, I have stored all the information given to the GUI in a Dictionary of string and List. I was storing the same in the text file. The source state being the key of the Dictionary.

Now, for example consider the following example where we give the following input to the GUI to build the graph.

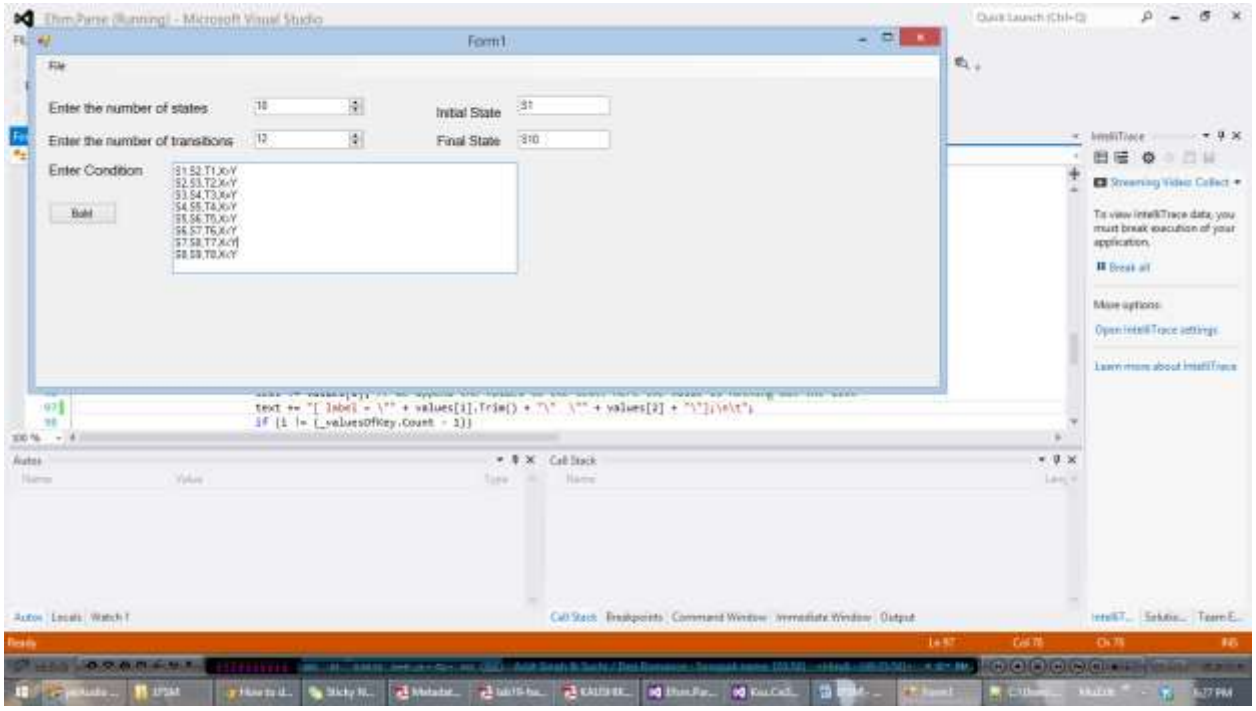

Figure 3.7 GUI Input

Once, the input is given and the build button is clicked, it prompts you to save the file with the given contents. Make sure that you save it with a ".gv" extension as shown below.

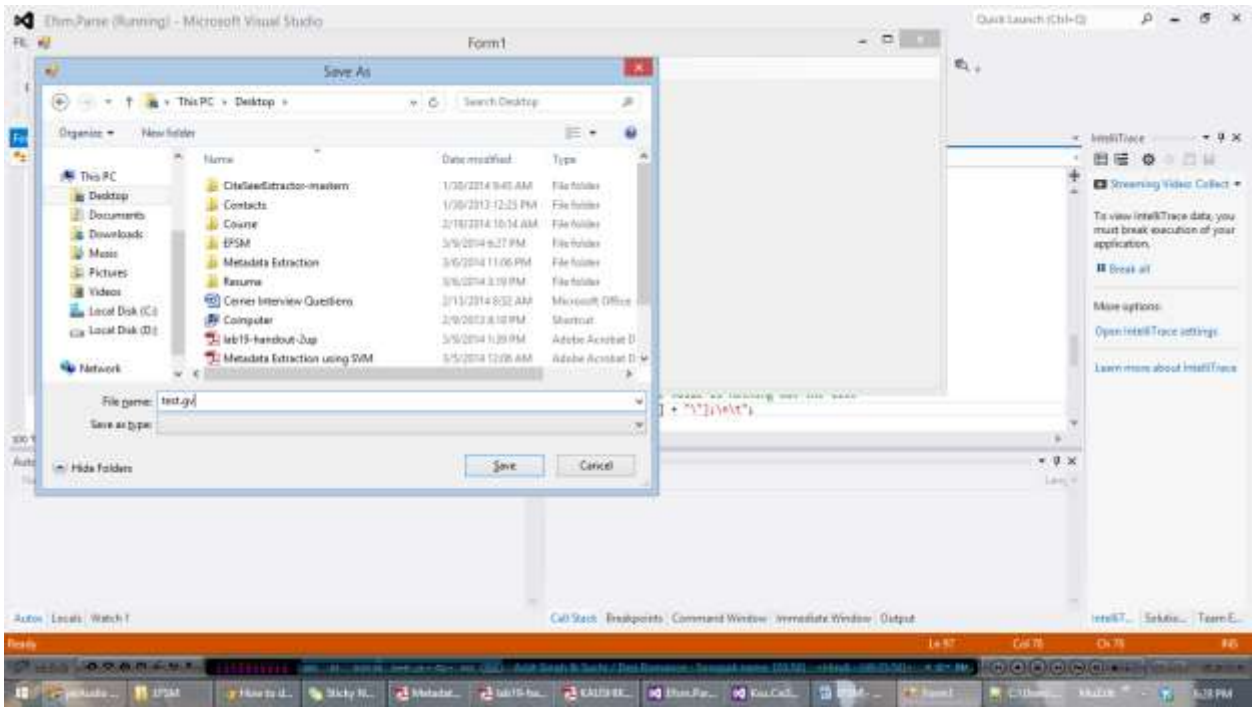

Figure 3.8 Saving the file with '.gv' extension

When you save the file with a .gv extension, all the input given to the GUI is stored in the following format.

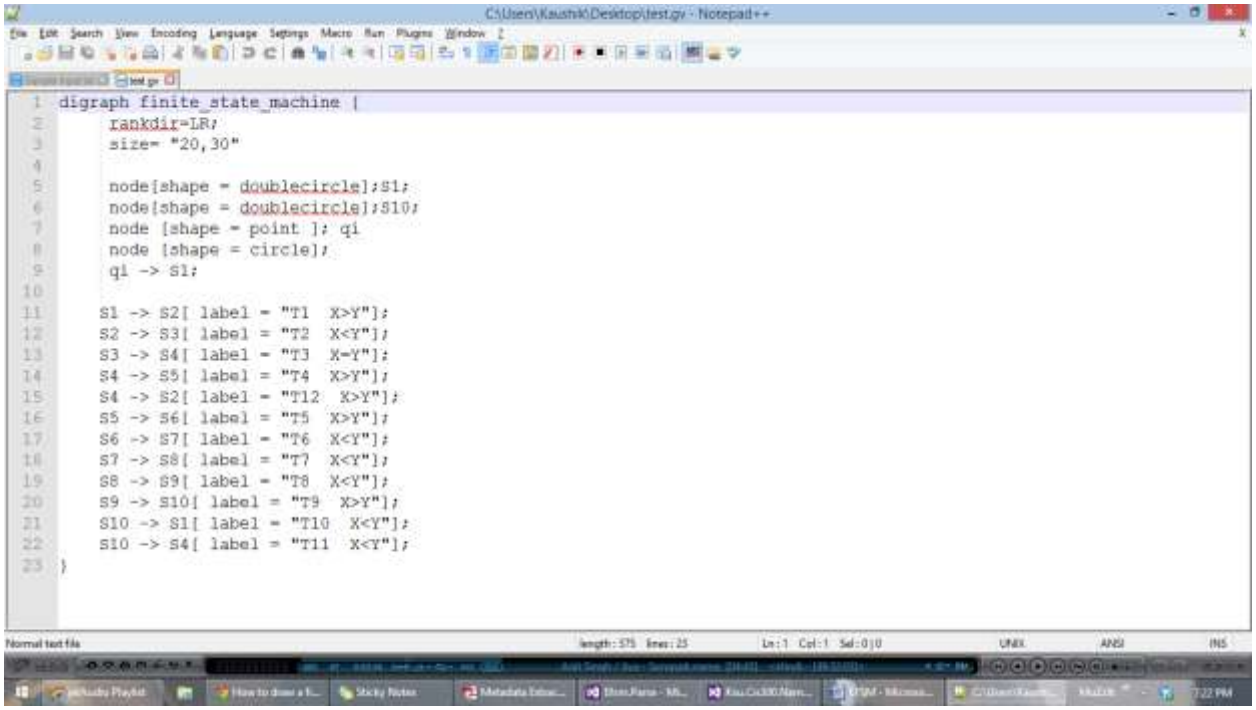

Figure 3.9 An example of '.gv' file

The GraphViz tool takes only '.gv' files as input. The input for the file has to be in the above specified format for it to generate the graph.

Now this file is given as input to the GraphViz tool (See User Manual for installation).

Figure 3.10 GraphViz command

This would generate a graph. The graph is generated & stored in '.png' format. The graph for the above given example would be as shown below:

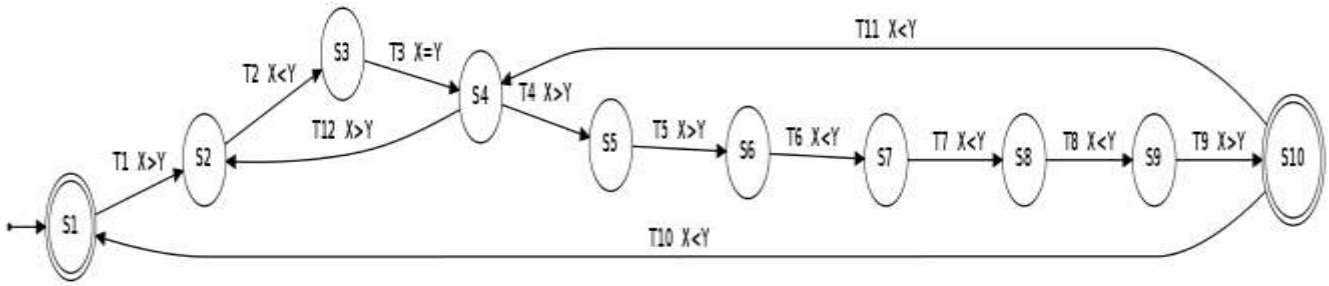

Figure 3.11 Sample Graph

Thus, this tool is very much useful in building graphs effectively. It has minimum intersection between transitions. One loop hole with this tool is that it doesn't work on Windows platform and I have been writing my code using the Microsoft Visual Studio IDE which works only on Windows. The solution to this problem is using the GraphViz API for C#.

Now that the input specification, storing and modifying an EFSM are working for sample input. The next big task is to check for the conditions given as input and then simulate the graph. In addition to that, I have also been researching on tools and APIs that facilitate graph building in Visual Studio.

Several major modifications:

I have tweaked the GUI a bit to facilitate graph building on the form. The GUI now has an additional numeric up down and an additional text box. The numeric up down is used to input the number of variables and the text box is used to input the initial values to the variables. This would be a comma separated field. The user doesn't have to worry about the names for the variables, they start from 'A' by default.

Initially, the GUI is used to generate a ".gv" file which is fed as input to the GraphViz tool.
Now, the graph is shown on the GUI itself. For this, I have included a picture box in the GUI and used the GraphViz wrapper API for C#. The GraphViz doesn't work with Windows, so I have chosen to use the API.

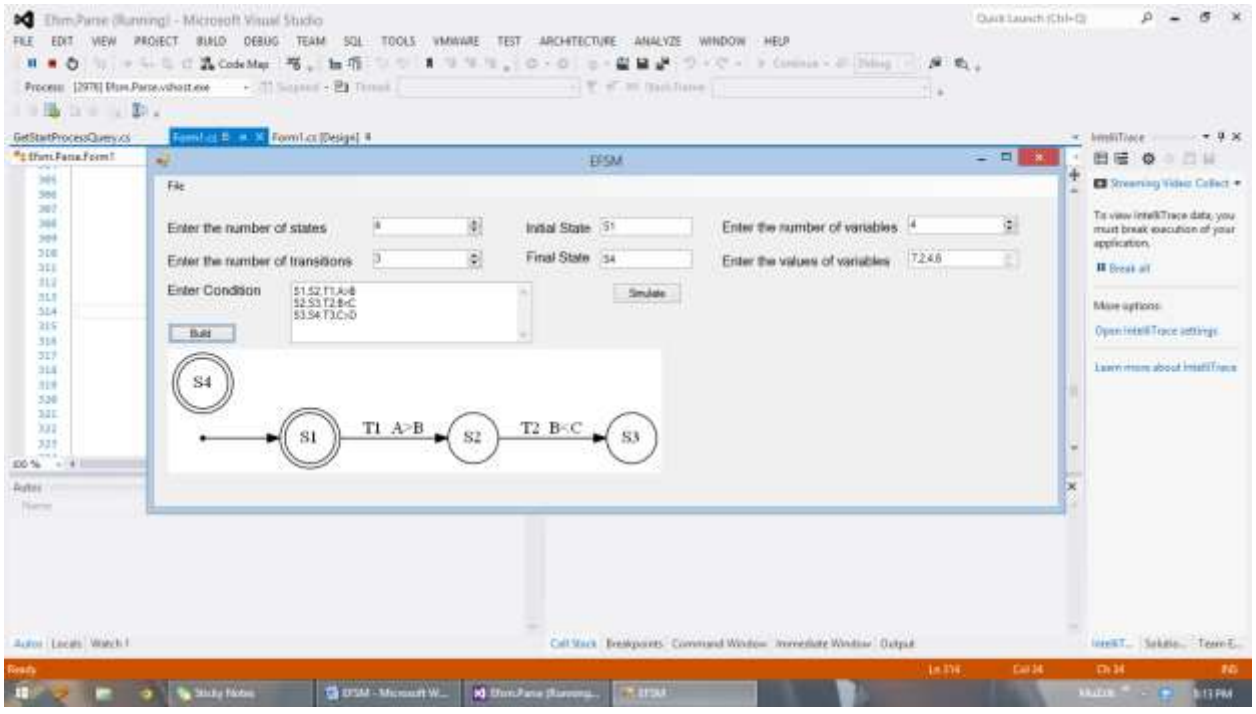

Figure 3.12 Graph built on the form

GraphViz API:

I used the GraphViz C# library to achieve the above. It fundamentally uses the GraphViz (dot) in my C# project. By default, *dot* is basically the dynamic library gvc.dll and a bunch of plug-ins that actually do the layout and renderings. It's basically a few lines of code that reads the graph and calls the layout and rendering algorithms for each. This constructs a representation of a graph in the *dot* language.

The program computes the position information for the graph, attaches the desired attributes and returns the graph back to the application through a file or pipe. The application then reads the graph and apply the geometric information as necessary.

To build the graph directly on the form, I have attached a picture box to the form. This facilitates the API to draw the graph directly on the picture box.

Simulation of a graph now seems a possible task. Simulation shows each transition between states sequentially. To achieve this a new button "Simulate" has been added to the GUI. The event handler of this button checks if there is any text in the text box. If at all there is some text, it builds the dictionary first. It first trims the whole text and stores it in a string variable. It then splits the string using the End Of Line character and stores the resultant strings in an array of strings. It is further split using the character 'comma'. We then check for all the required conditions before we simulate the graph.

Figure 3.13, figure 3.14 and figure 3.15 describe simulation for the same input as given above.

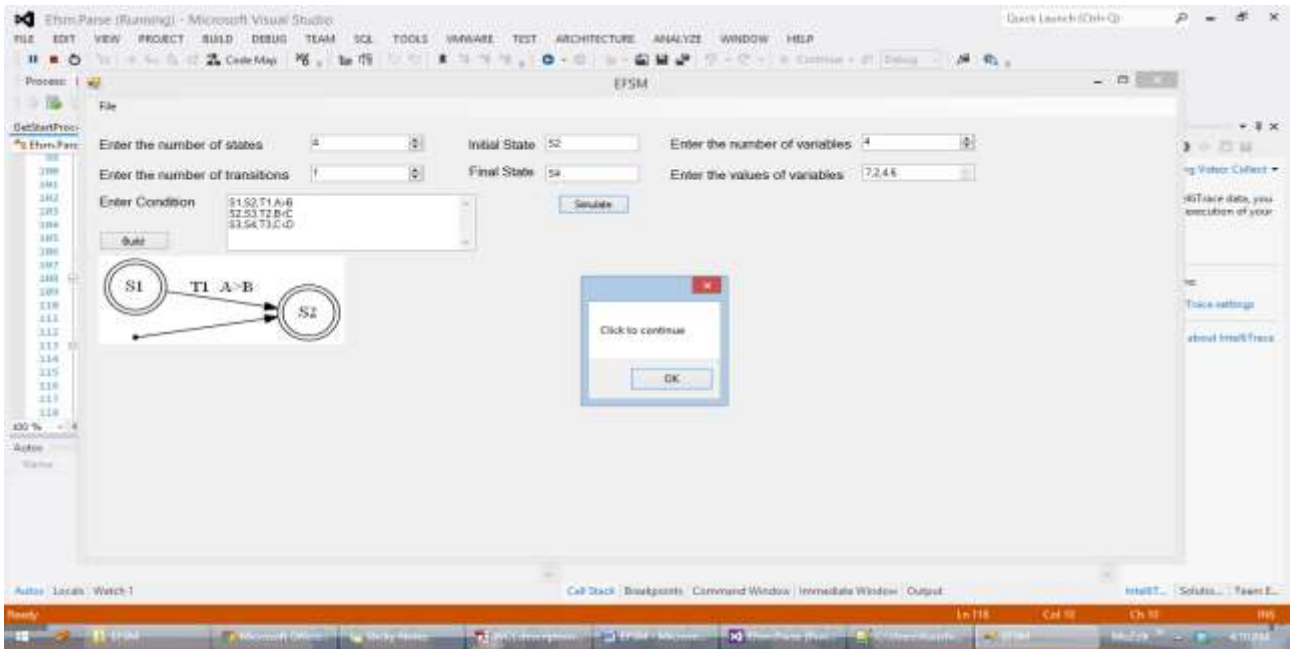

Figure 3.13 First step of simulation

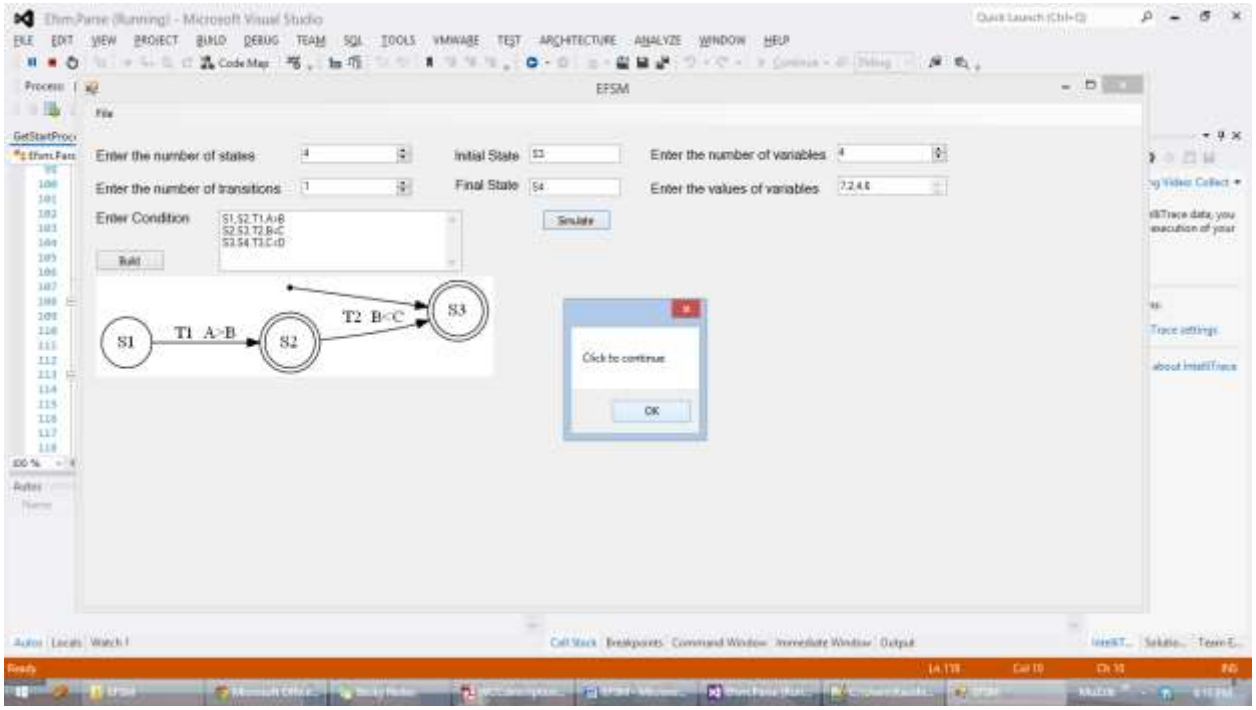

Figure 3.14 Second step of simulation

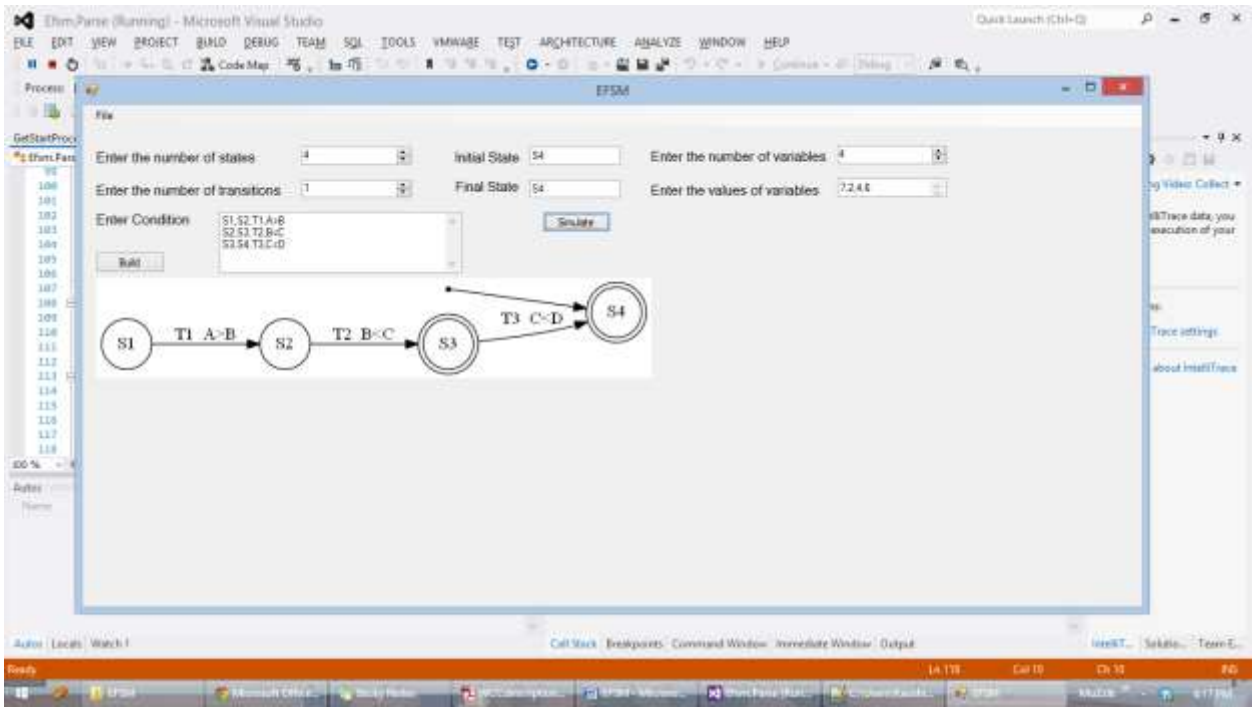

Figure 3.15 Third step of simulation

Since all the given conditions in the above input are true, this can be considered to be the best case. The above screen shots show a working demonstration of a simple graph simulation.

The next phase is to implement randomization. This is the most crucial part of the system. The main task of randomization is to pick one transition and fire it based on all the conditions or expressions that evaluate to true.

Thus, when we have multiple transitions from a single state whose expressions evaluate to true, the tools picks one such state randomly using the random function. So the simulation varies for same input when you try to do it multiple times.

Chapter 4 - Data Dependence and Slicing Algorithm

Data Dependence

Now we go to phase two where we check whether the given set of transitions are data dependent.

To calculate data dependency between the given input transitions, we store all the input transitions. Then we check for all number possible combinations in the given. We then calculate D(t) and U(t'). D(t) is either empty or a singleton set. D(t) contains the variables modified by the first transition. U(t') is the union of the variables contained in both the evaluating expression (or the condition) and in the enabling events of the second transition.

The standard definition of data dependence is as follows:

We say that transition t' is data dependent on transition t, written $t \rightarrow_{dd} t'$*, if and only if there exists a variable* $v \in D(t) \cap U(t')$ *and a path* $[t_1...t_k]$ $(k >= 0)$ *from T(t) to S(t') such that for all* $j \in 1...k$*, v does not belong to* $D(t_j)$*.*

We then calculate the intersection of both of these i.e. D(t) ∩ U(t'). If at all if the intersection is not empty then we find the paths. We start by finding the source state of t' (S(t')) and target state of t (T(t)). Once we have them, we check for all possible paths from T(t) to S(t') using breadth first search. From the set of possible paths we look for the path/paths with transitions that do not have the same variable as in D(t) ∩ U(t') in their enabling events. If at least one such path exists then t' is data dependent on t i.e. $t \rightarrow_{dd} t'$.

The implementation code for data dependency works well. It has been checked for various inputs. One example that vividly shows that the code for data dependency is in place is described.

Consider the case where the GUI is fed with the following input.

S1,S2,T1,A>B,B=4
S2,S3,T2,B<C,D=6
S2,S4,T3,A>C

S2,S6,T4,D<E

S3,S4,T6,A>D

S3,S6,T5,A<E,B=7

S4,S5,T7,A<E

S4,S6,T8,C>B

S5,S6,T9,D<C,B=1

Figure 4.1 shows the graph for the above input.

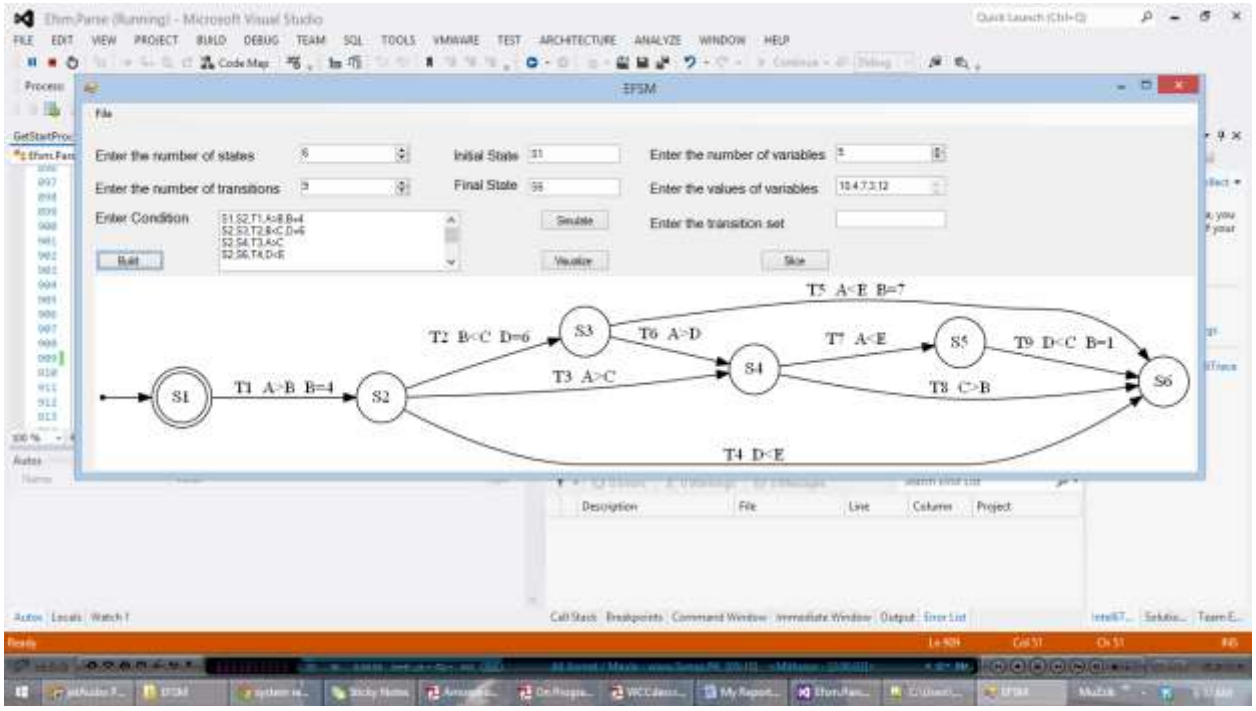

Figure 4.1 Building Graph for Data Dependency

The same input is used as input to test data dependency. It checks for all possible combinations in the given set of transitions. For each such combination, D(t) and U(t') are calculated. For example if the pair of transitions is (T1, T9). Then D(T1) ∩ U(T9) is calculated. We find all possible paths from target of T1 i.e. S2 to source of T9 i.e. S5 using Breadth First Search (BFS). Then, we add to the queue only the paths with transitions that do not have the variable v ∈ D(T1) ∩ U(T9) in their enabling events. One such path exists from S2 to S5. Hence transition T9 is data dependent on T1.

Data dependence for the given input transition set is shown in figures 4.2 and 4.3.

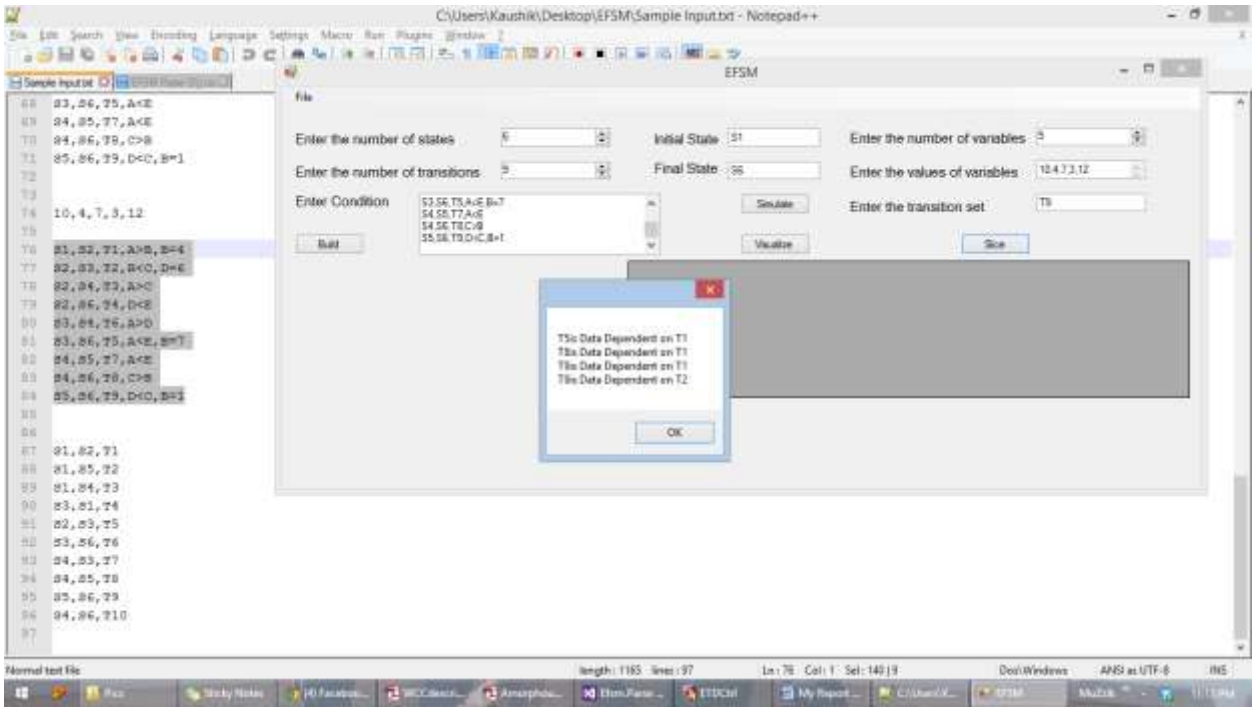

Figure 4.2 Data Dependency (1)

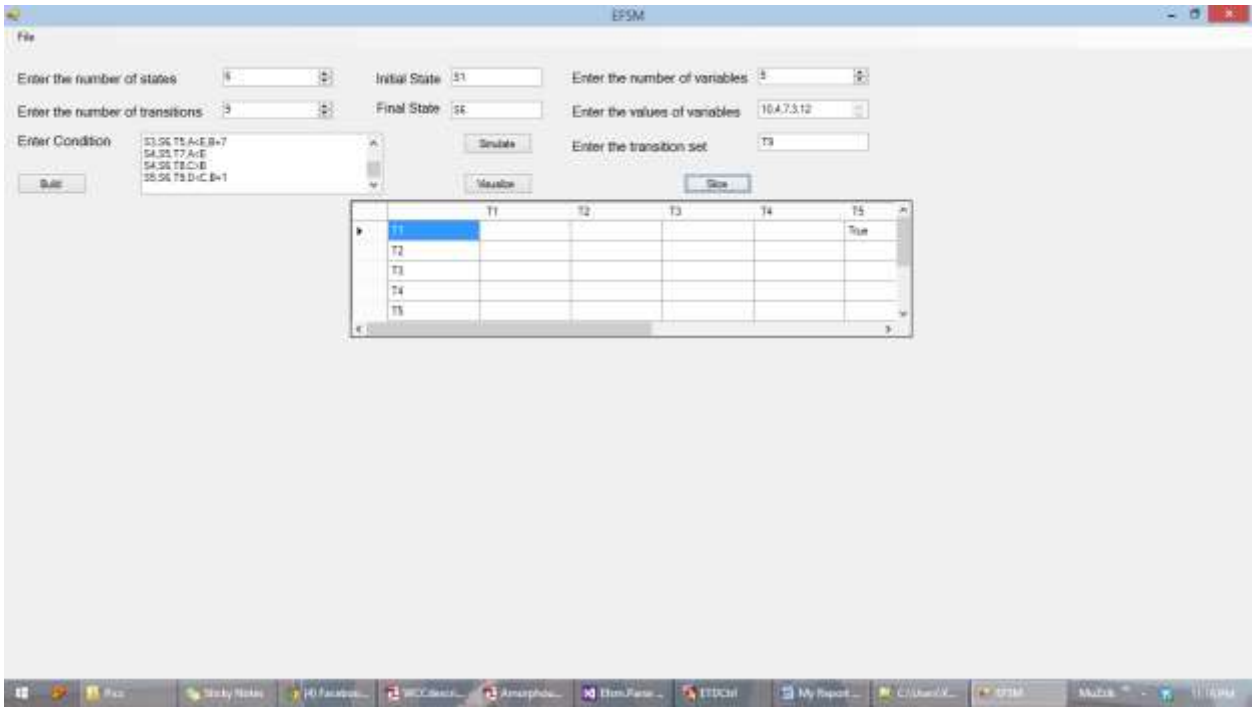

Figure 4.3 Data Dependency (2)

Once data dependencies have been calculated, we find the transitive closure.

For example, if we have the following pair of transitions that are data dependent (T1, T9), (T9, T3), (T9, T6) then the transitive closure also includes (T1, T3), (T1, T6) in addition to the above pair of transitions.

The next goal is to build the table DDStar. This is a two dimensional table where each row and column is represented by the transition names. This is constructed using the Data Dependencies and the transitive closure. The table displays "True" if two transitions are data dependent. For example, in figure 4.3, T1 and T5 are data dependent thus we display "True" in the corresponding field in the table.

Now the last phase is to implement the slicing algorithm.

Slicing Algorithm

The algorithm found in [2] produces a set of sliced transitions. It takes a set of transitions as input from the user. Then the following algorithm is applied:

The set of transitions given as input by the user to slice is stored in a HashSet "L".
All the transitions is stored in a structure say X.

```
for each t ∈ L
  for each u does not belong to L (i.e. for each u in X - L)
        if DDStar(u, t)
          L := L ∪ {u}
repeat
  Lnew := ∅
  B := {n | ∃t ∈ L : n = S(t)}
  for each n ∈ B do
        obs[n] := n
  V := B
  C := B
  while C ≠ ∅ and Lnew = ∅ do
        Cnew := ∅
        for each m ∈ C do
```

```
        for each transition t does not belong to L with T(t) = m do
            n := S(t)
            if n ∈ V
               if obs[n] ≠ obs[m]
                     Lnew := Lnew ∪ {t}
            else
               V := V ∪ {n}
                Cnew := Cnew ∪ {n}
                obs[n] := obs[m]
    C := Cnew
L :=  L ∪ Lnew
for each t ∈ Lnew do
        for each u does not belong to L  do
            if DDStar(u, t)
               L := L ∪ {u}
until Lnew = ∅
```

This algorithm returns the new transition set Lnew.

See figures 5.25 and 5.26 for the implementation of the algorithm.

Chapter 5 - User Manual

Read Me

Running the tool is pretty simple. Follow the below the steps.

1. Open Visual Studio 2010 or latest
2. Click on 'File' -> 'Open Project'. Browse through the windows explorer and select the folder 'Efsm.Parse' and then select the Visual Studio Solution.
3. Once the project is loaded, click on 'Start'.
4. Input all the specifications based on the requirement.
5. The available options are to build, simulate and slice. You can choose any one of these just by clicking on the corresponding button.
6. The 'Build' and 'Simulate' button generate graphs
7. The slice button gives you a set of transitions.

System Requirements

Operating System: Windows XP, Windows 7, Windows 8, Windows 8.1

RAM: 512 MB or more

IDE: Visual Studio 2010 or latest

GraphViz Installation in Linux

To install GraphViz, type in the following command.

sudo apt -get install graphviz

Open terminal and type in the following command.

dot -Tpng filename.gv -o filename.png

This command takes a '.gv' file as input and generates a '.png' file as output which has the graph for the given input.

Chapter 6 - Testing The System

The system has been tested for various inputs to check if it works best for all such inputs. For all the inputs mentioned below, the initial values of the variables are as follows:

A = 7, B = 2, C = 4, D = 6

First Test

The input given below is one of the cases where all the expressions evaluate to true. This test case is chosen because the graph simulation takes into consideration all the transitions.

S1,S2,T1,A>B

S1,S3,T2,B<C

S3,S4,T3,C<D

The graph is built in the following way when this is given as input to the system.

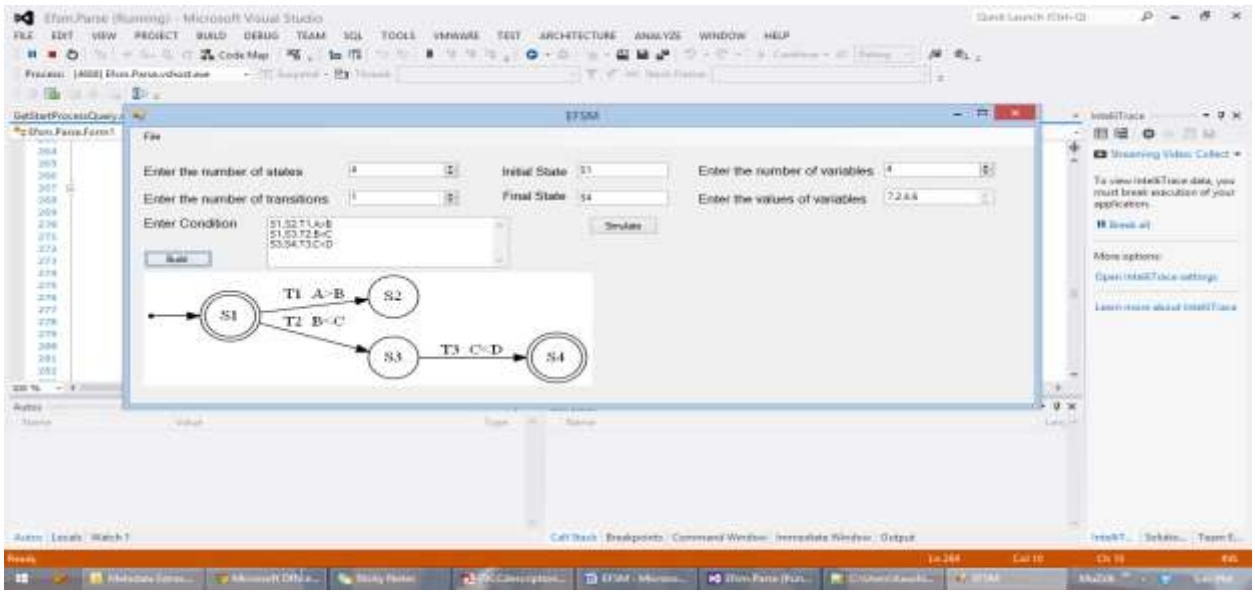

Figure 6.1 Building Graph (Input 1)

The simulation for the above input has two cases. The first case is when the random function fires transition T1 which brings the machine to S2 and the program stops there as we do not have any path further from S2. So, the system throws an exception saying "No path ahead". The

second case is that the system fires transition T2 which fetches the machine to S3 and then it fires transition T3 which brings the machine to the exit state i.e. S4 and the execution stops.

The random simulation for the above input is shown in figure 6.2 and figure 6.3 .

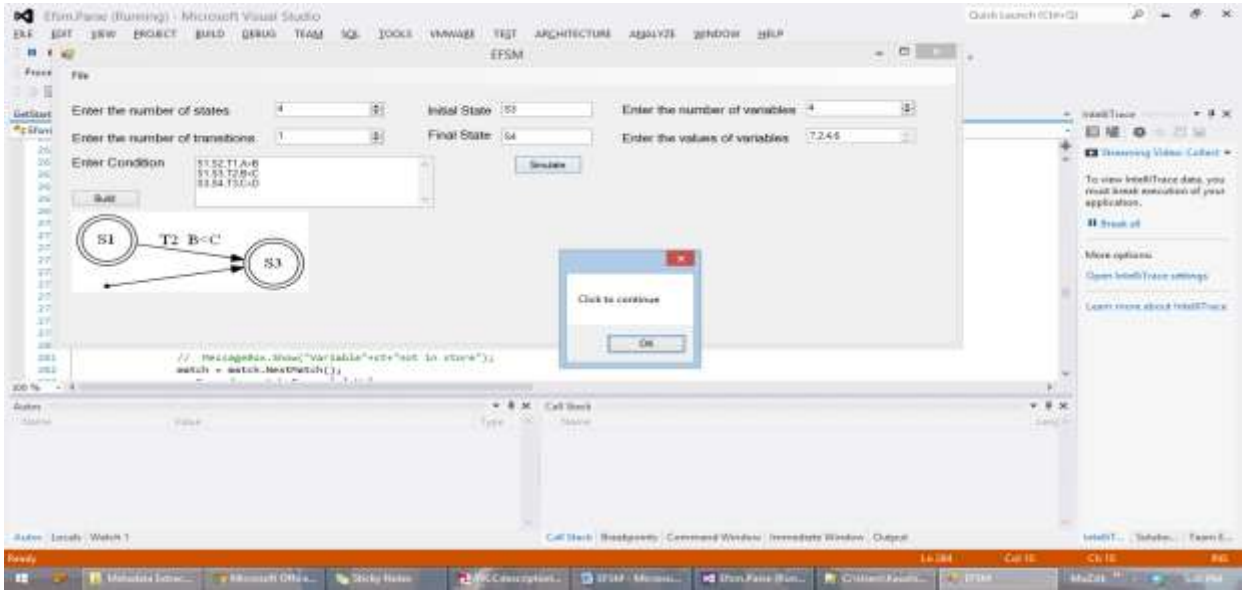

Figure 6.2 Random Simulation Step 1 (Input 1)

As described above, the system randomly picked transition T2. This leads to the final state in the next step.

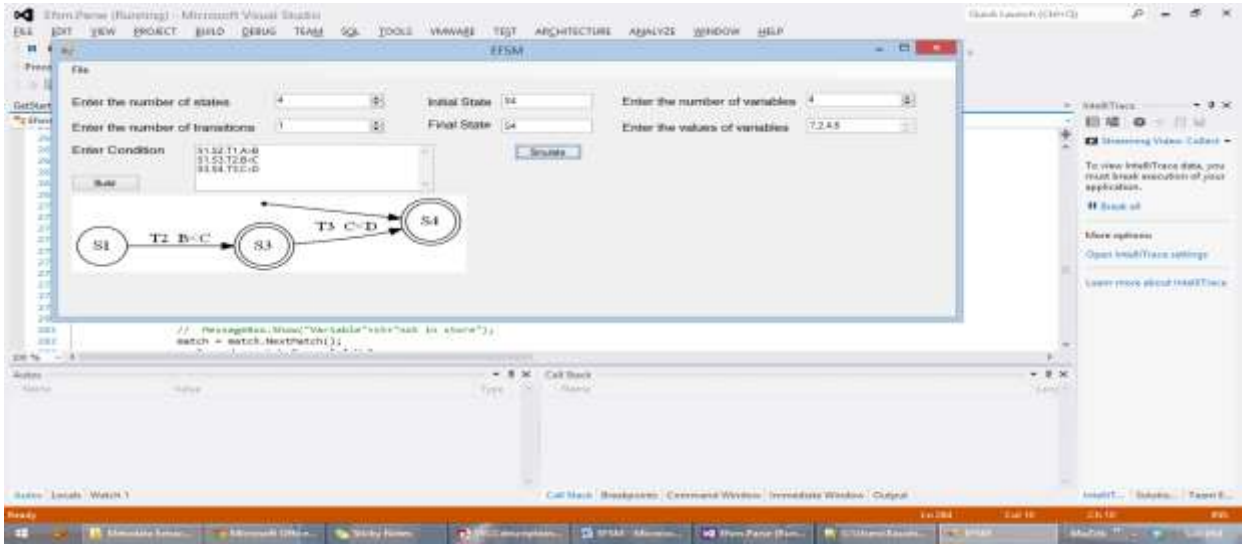

Figure 6.3 Random Simulation Step 2 (Input 1)

Second Test

Consider the following test input where the random simulation has a choice to make between the transitions. This is similar to the first test case except that it has a self looping transition.

S1,S1,T1,A>B

S1,S3,T2,B<C

S3,S4,T3,C<D

The graph for the above input is generated as follows:

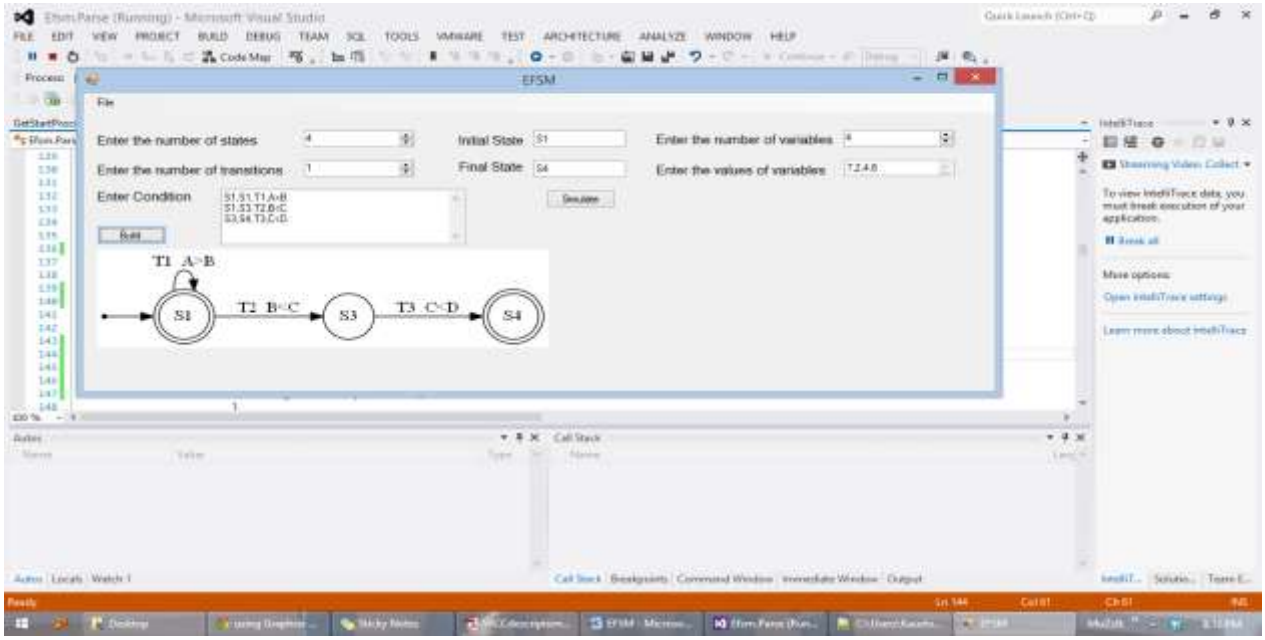

Figure 6.4 Building Graph (Input 2)

Simulation for the above input is shown in Figure 6.5

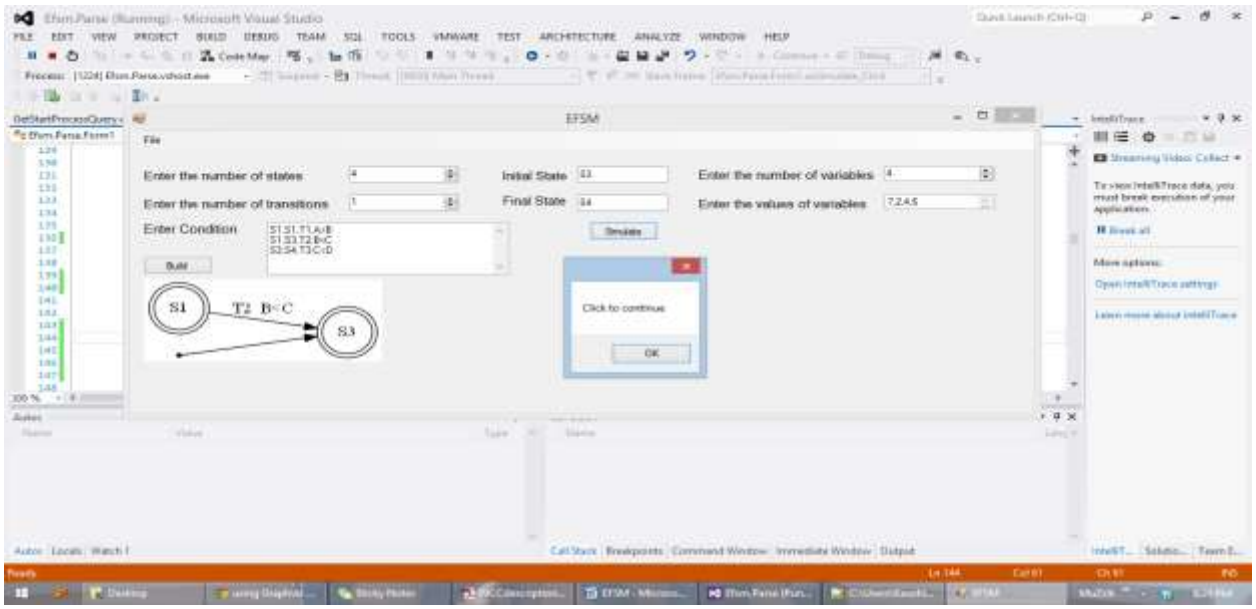

Figure 6.5 Random Simulation Step 1 (Input 2)

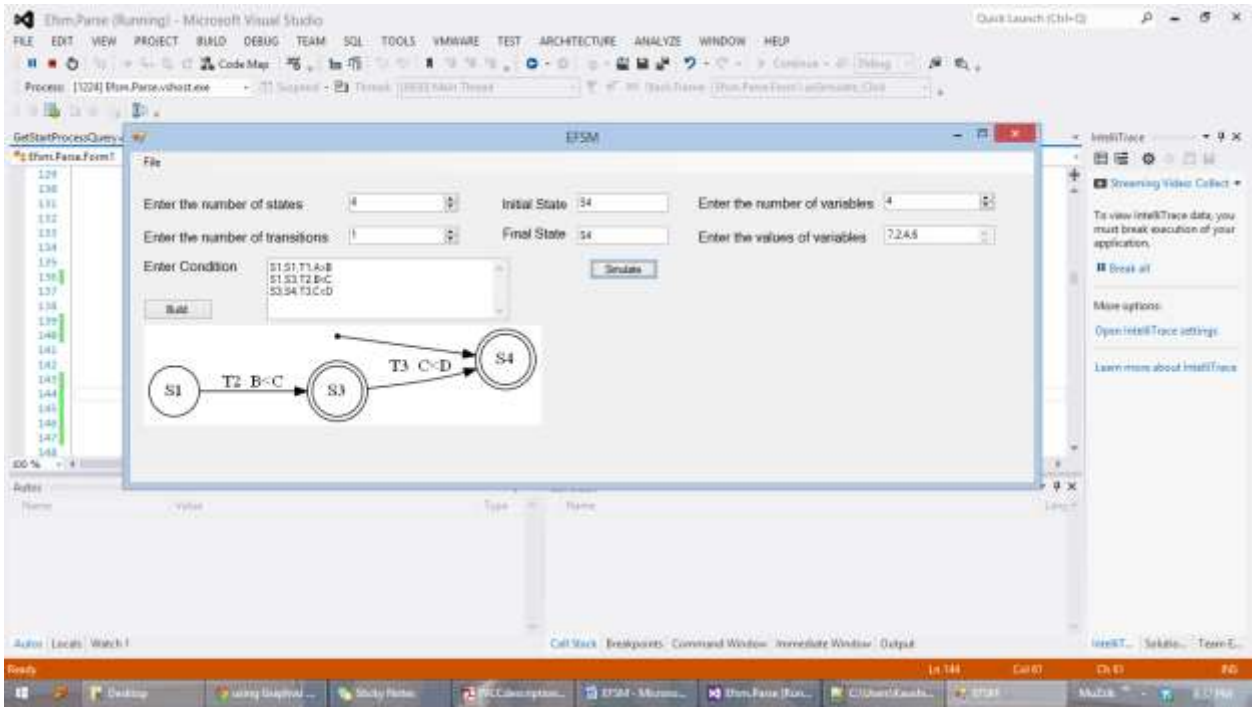

Figure 6.6 Random Simulation Step 2 (Input 2)

Third Test

Consider the following input

S1,S2,T1,A>B

S1,S3,T2,B<C

S1,S4,T3,C<D

S2,S4,T4,C<D

S2,S3,T5,A>B

S3,S4,T6,C<D

S3,S1,T7,B<C

The above input has multiple transitions from a single state. This test case has been selected to check the random simulation. This kind of input is also essential to check for graph building because a single source state has multiple transitions that are fired which brings the machine to different target states i.e. a single state has multiple target states based results of the evaluated expressions. Figure 6.7 shows the graph generated by the system for the above input.

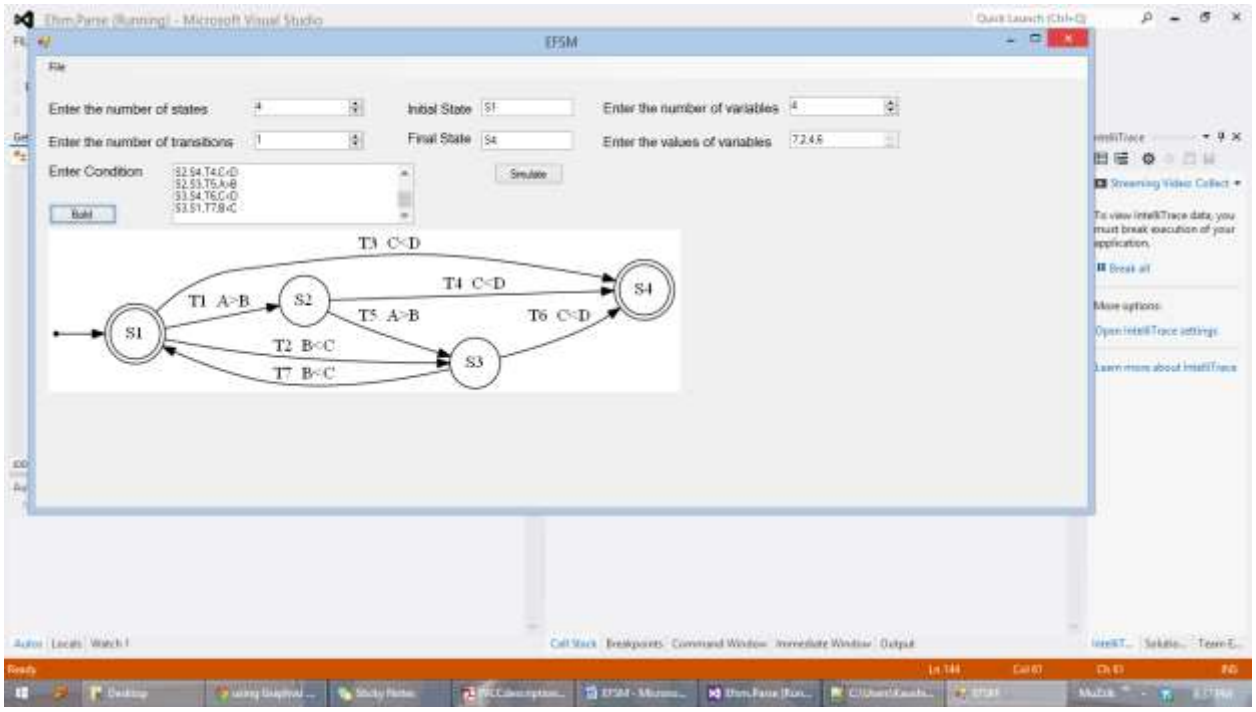

Figure 6.7 Building Graph (Input 3)

When we try to simulate the graph, there are many different possibilities as the system chooses a state randomly.

Simulation for the above input is shown in figure 6.8

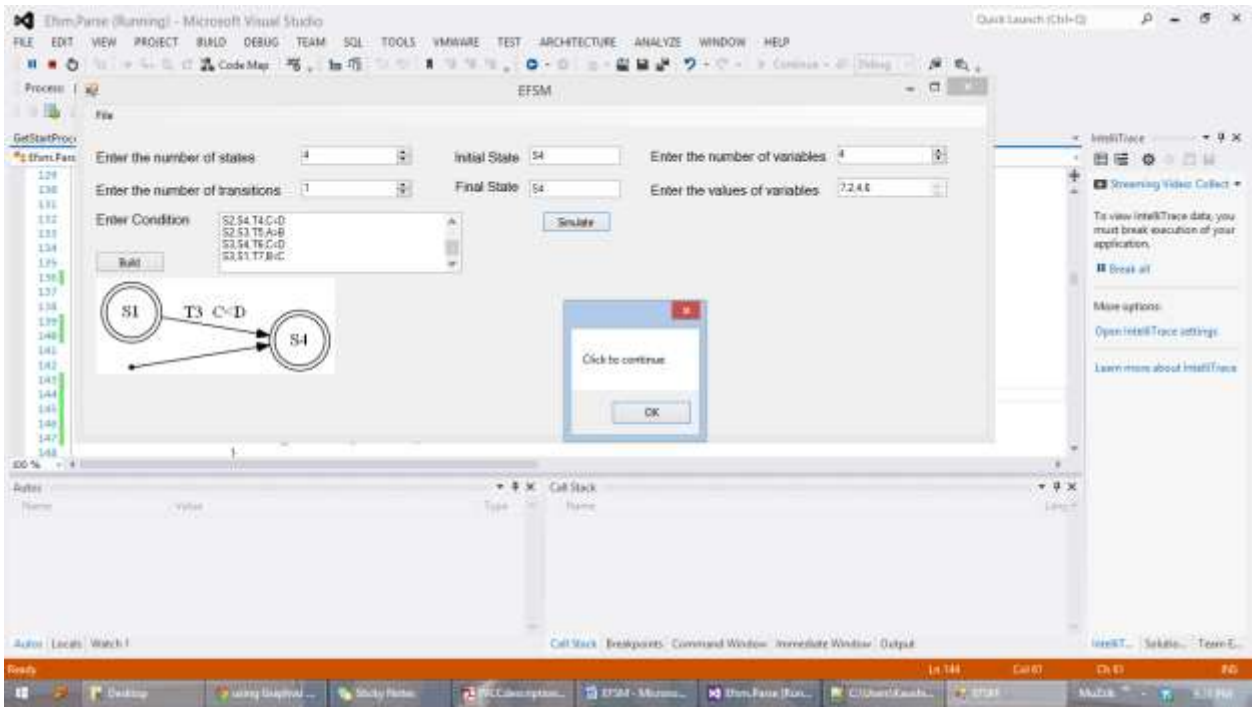

Figure 6.8 Random Simulation First Step (Input 3)

Simulation stops here as there is no other transition to fire from the current active state S4 but if S1 picks S2 or S3 then the simulation runs for more transitions.

Fourth Test

Consider the following input, where one of the expressions evaluate to false.

S1,S2,T1,A>B

S2,S3,T2,B>C

S3,S4,T3,C<D

This is how the graph is built.

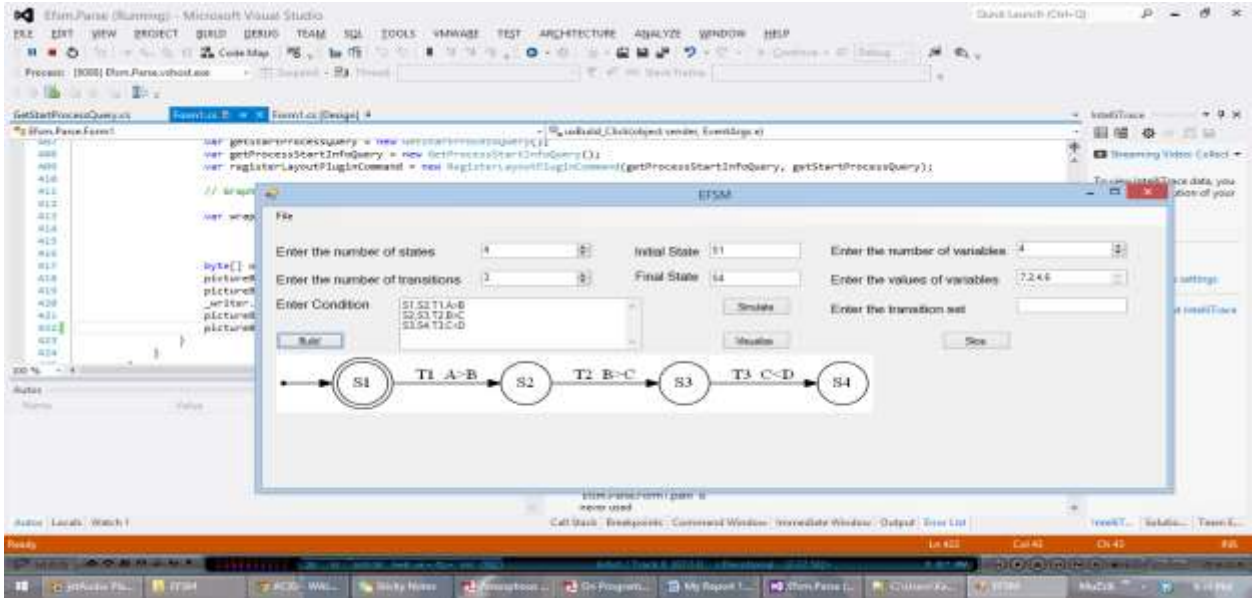

Figure 6.9 Building Graph (Input 4)

Simulating the above input would result in the following

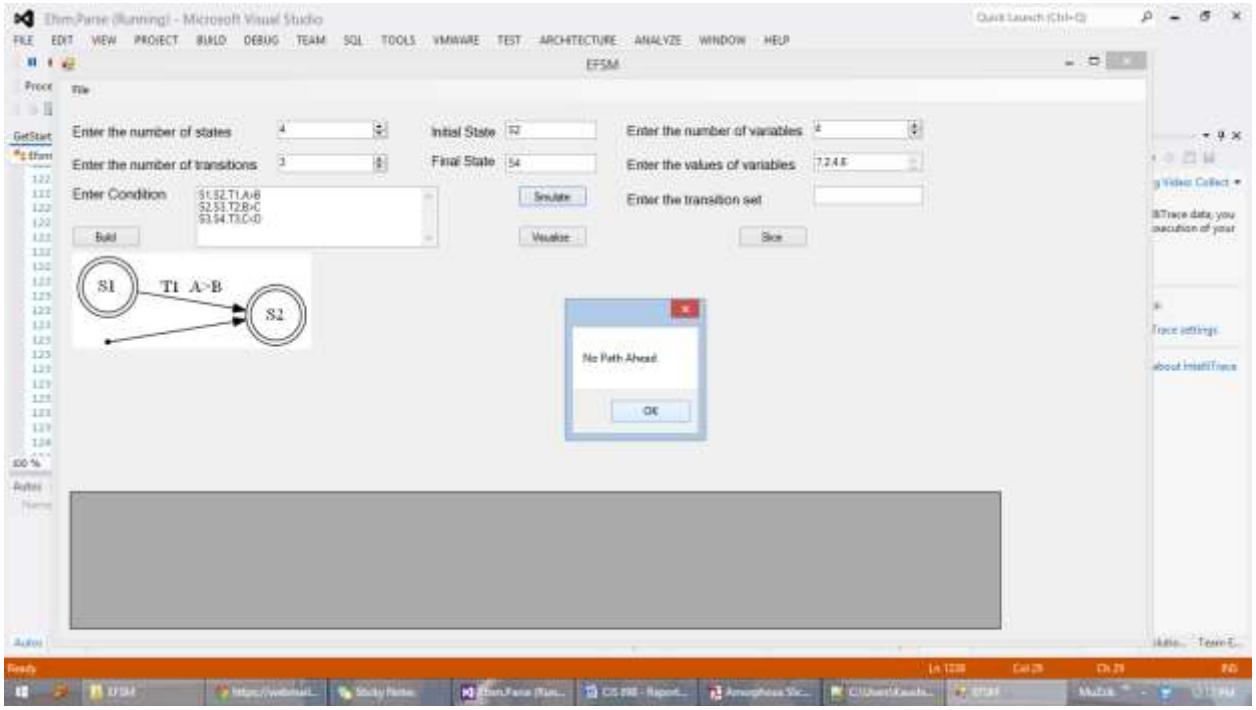

Figure 6.10 Random Simulation (Input 4)

Fifth Test

Consider the below given input where there is scope for random simulation.

S1,S2,T1,A>B

S1,S3,T2,B<C

S3,S4,T3,C<D

S2,S4,T4,C<D

Figure 6.11 represents how the graph is generated.

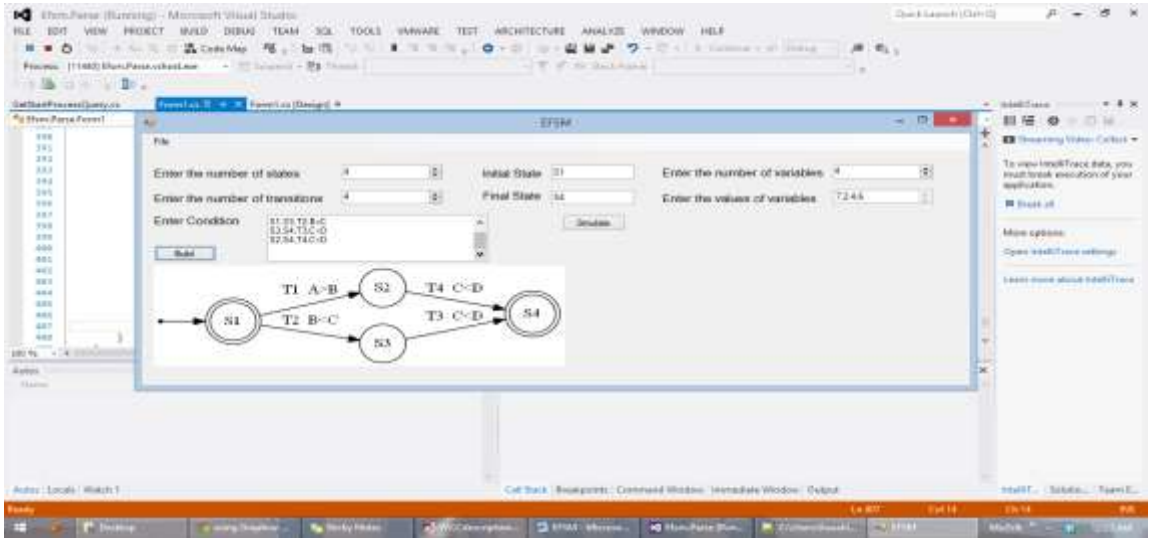

Figure 6.11 Building Graph (Input 5)

Figure 6.12 and Figure 6.13 represents the simulation.

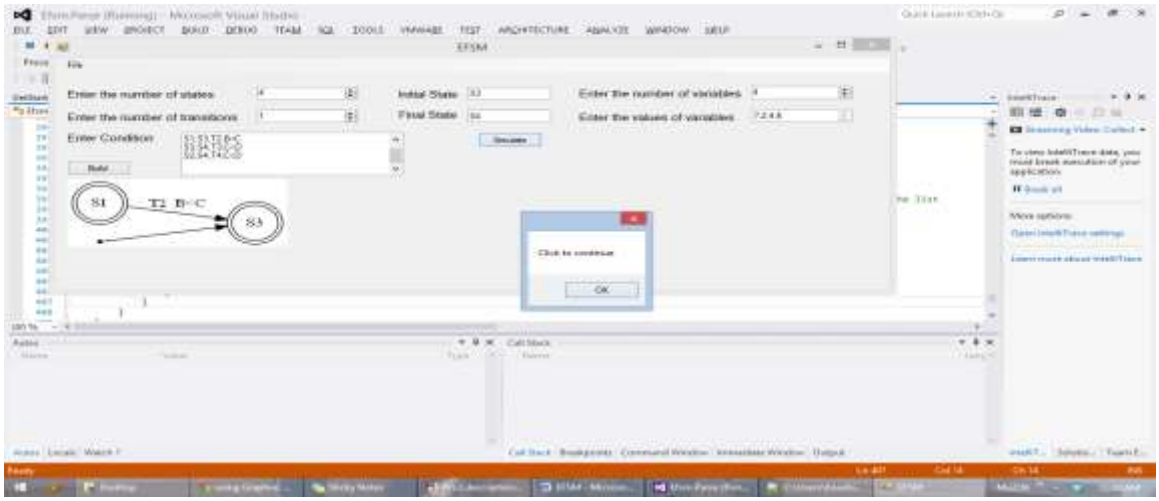

Figure 6.12 Random Simulation Step 1 (Input 5)

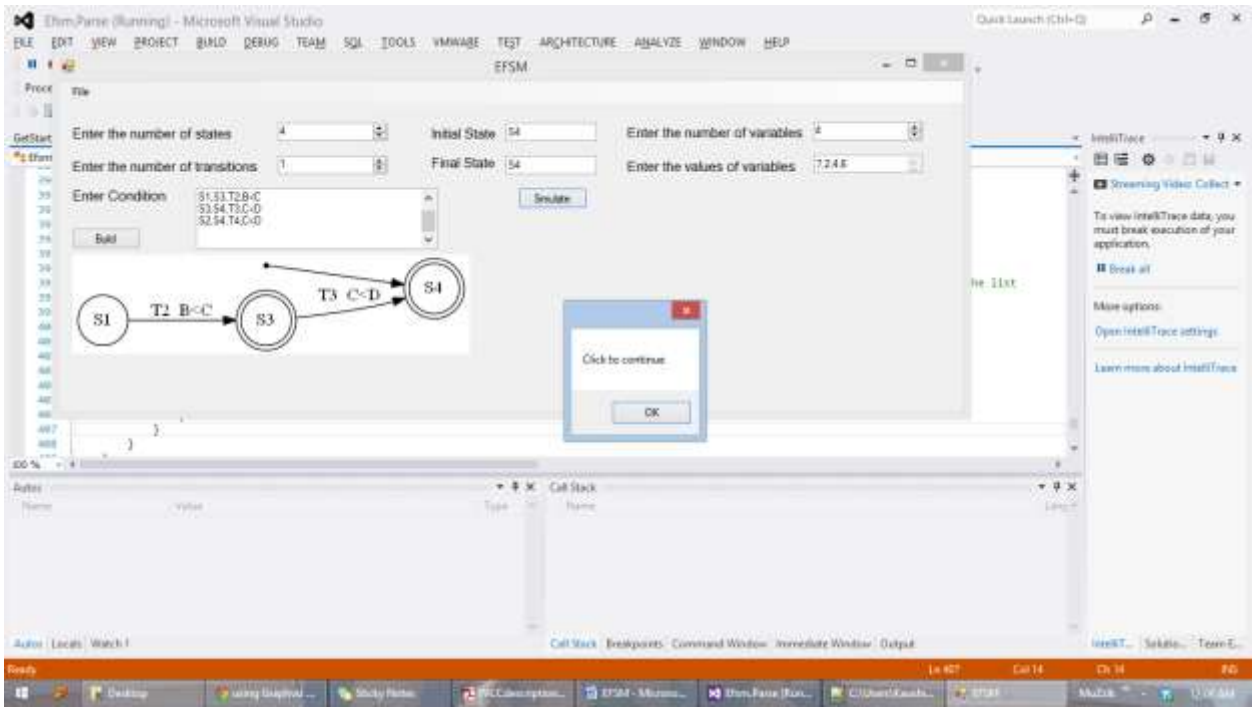

Figure 6.13 Random Simulation Step 2 (Input 5)

Sixth Test

Consider the below given input which is an example of ε-transitions.

S1,S2,T1

S2,S3,T2

S3,S5,T6

S3,S4,T3

S4,S5,T4

S5,S6,T5

Figure 6.14 shows the visual graph for the given input

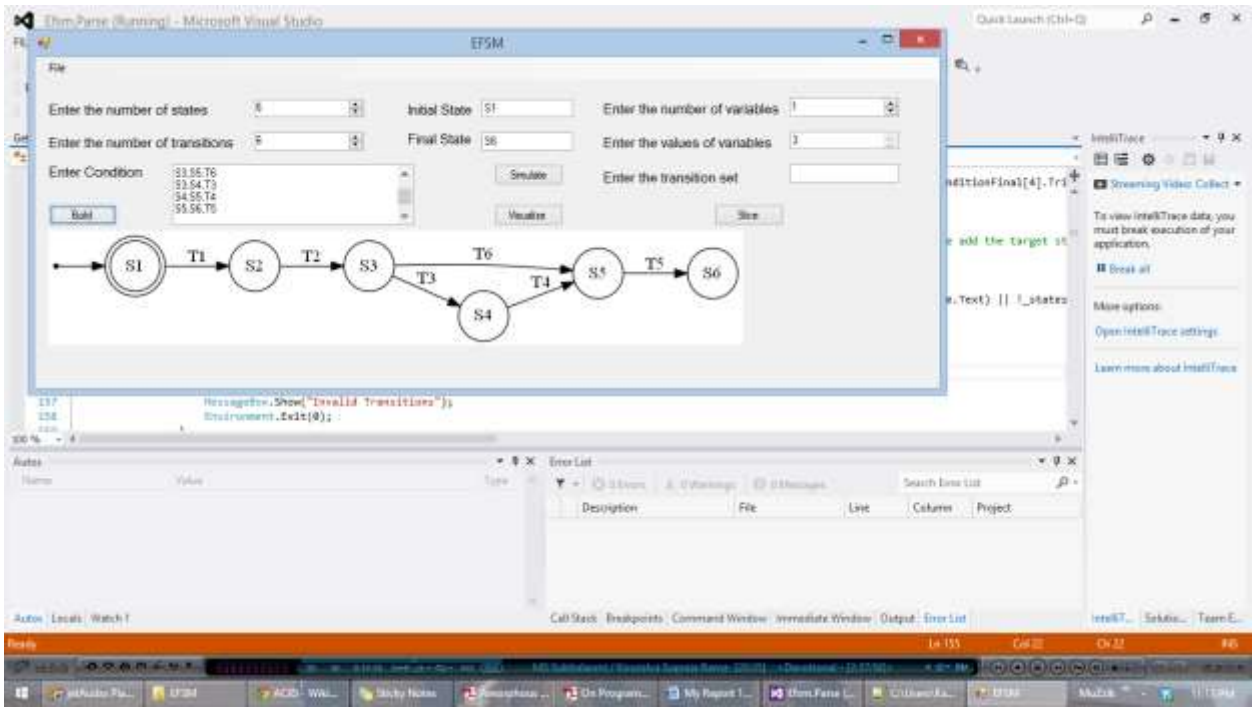

Figure 6.14 Building Graph (Input 6)

Seventh Test

Consider the following input where a few transitions have actions and a few don't.

The initial values of the variables are A = 10, B = 4, C = 7, D = 3, E = 12

The graph is generated without checking for any of the evaluating expressions or the enabling events.

Simulation checks for all the evaluating expressions and enabling events.

S1,S2,T1,A>B,B=4

S2,S3,T2,B<C,D=6

S2,S4,T3,A>C

S2,S6,T4,D<E

S3,S4,T6,A>D

S3,S6,T5,A<E,B=7

S4,S5,T7,A<E

S4,S6,T8,C>B

S5,S6,T9,D<C,B=1

Figure 6.15 shows the graph for this input.

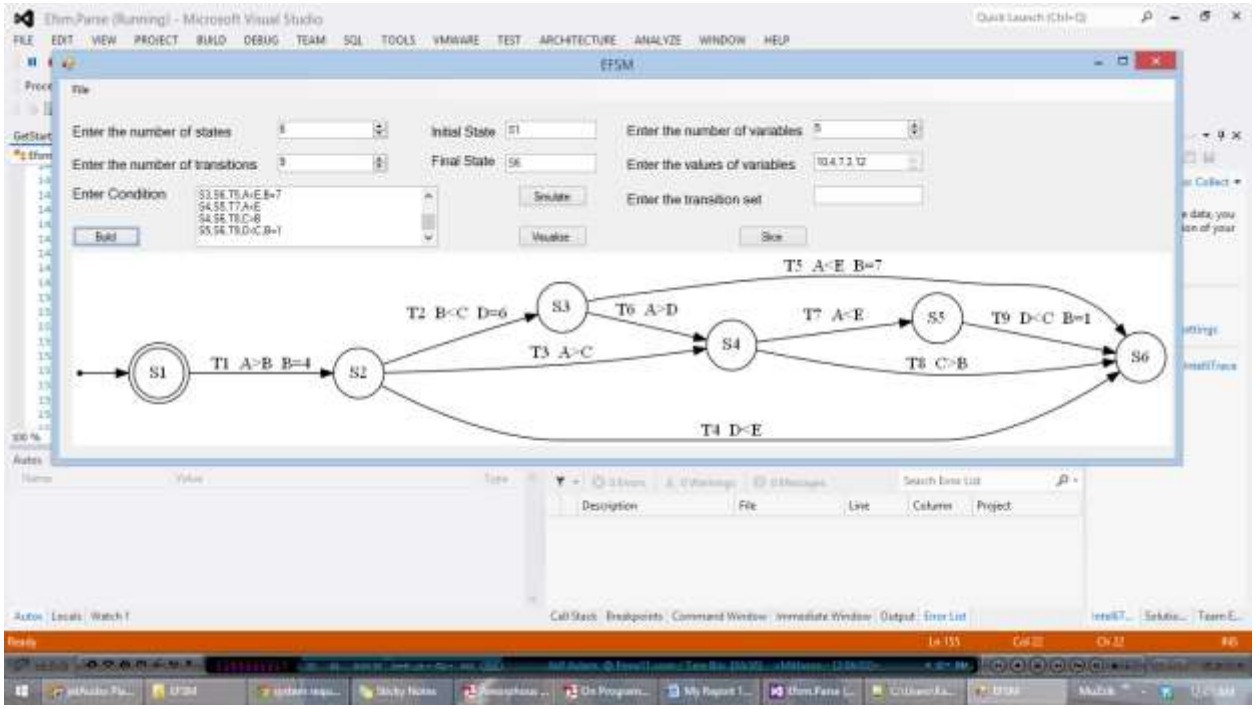

Figure 6.15 Building Graph (Input 7)

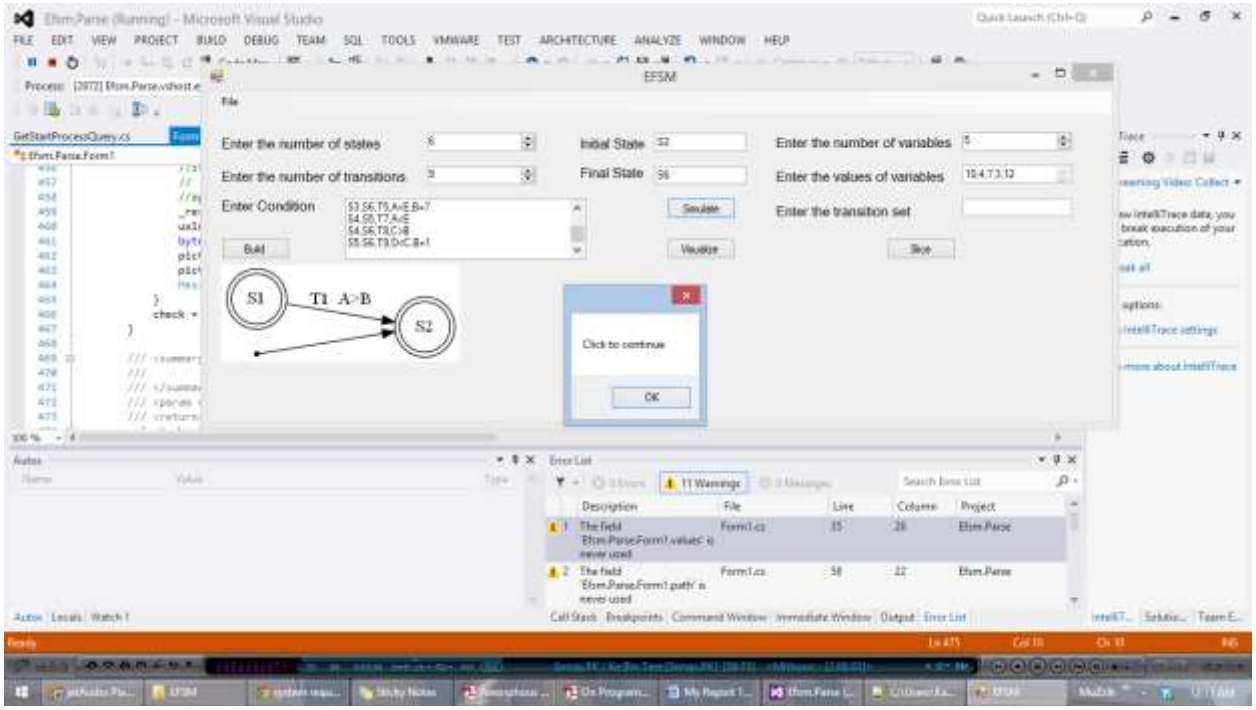

Figure 6.16 Simulation Step 1 (Input 7)

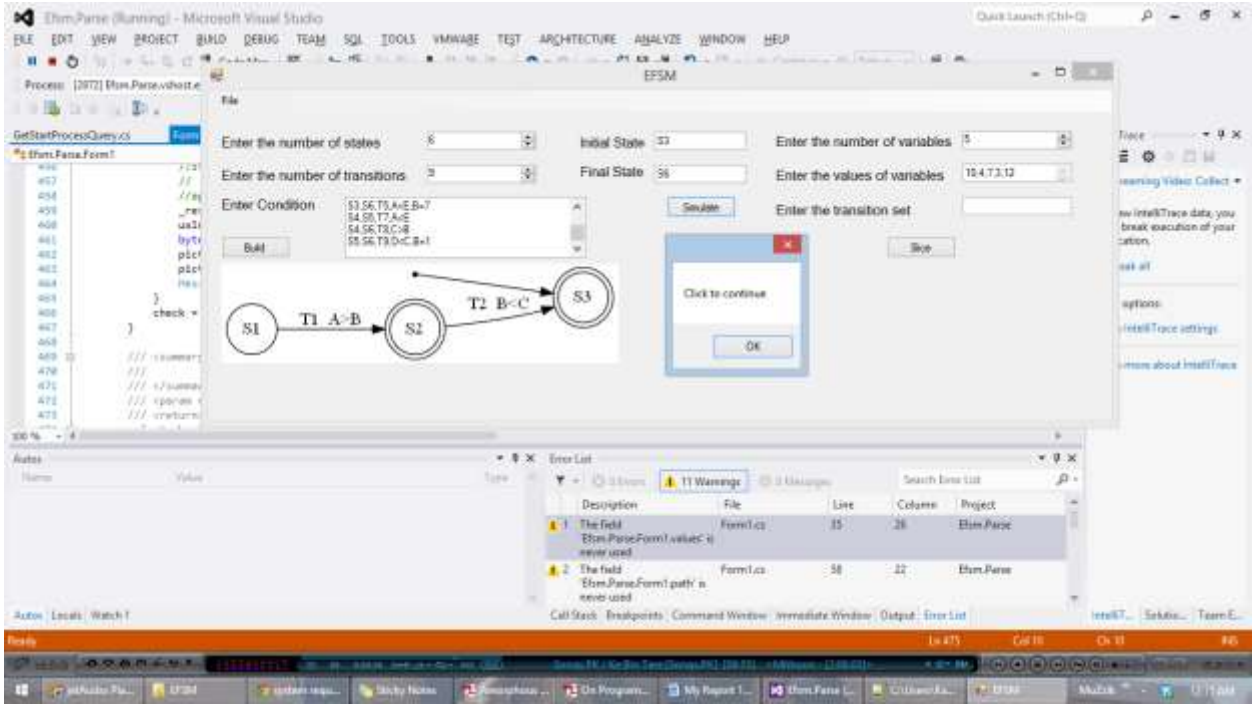

Figure 6.17 Simulation Step 2 (Input 7)

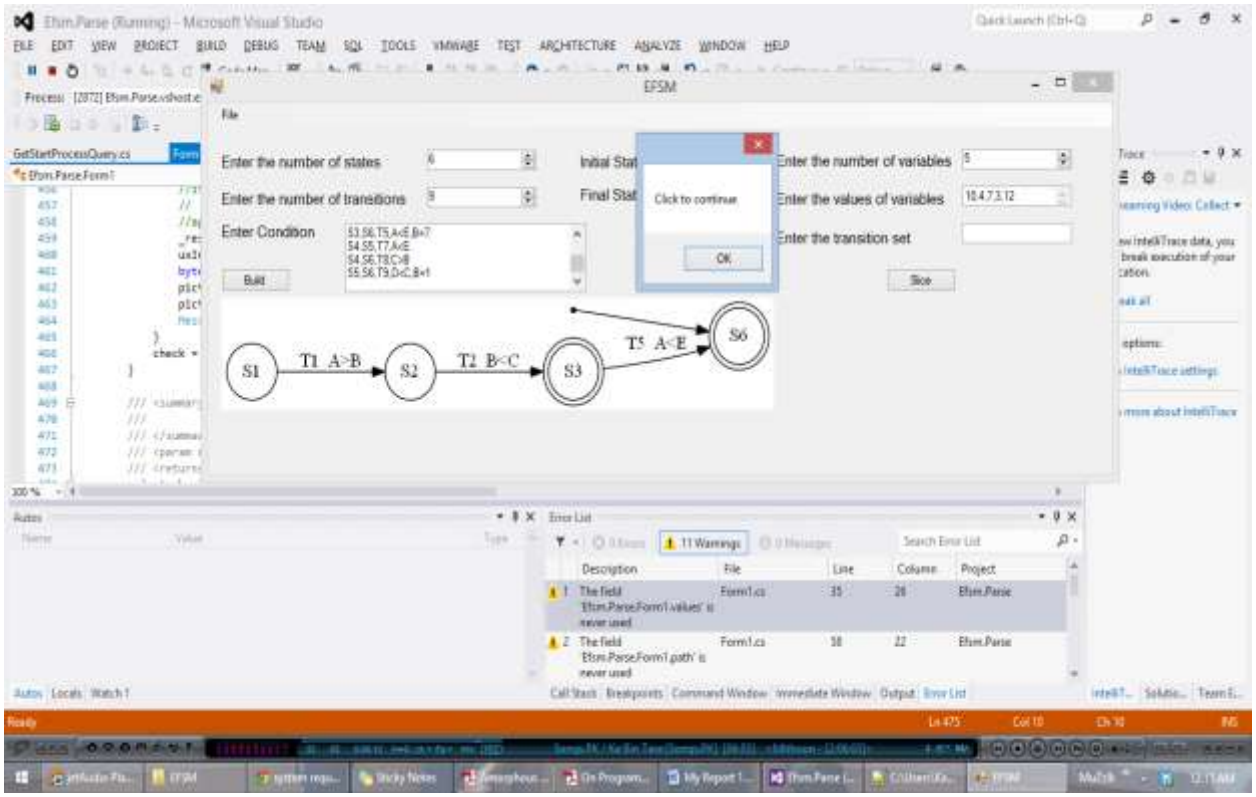

Figure 6.18 Simulation Step 3 (Input 7)

Eighth Test

Consider the case where the user enters the value '2' in the number of states field and enters more than two states in the "condition" field. The GUI throws an exception. It displays a message box saying "Invalid States" and exits from the application.

Figure 6.19 shows the exception thrown by the GUI.

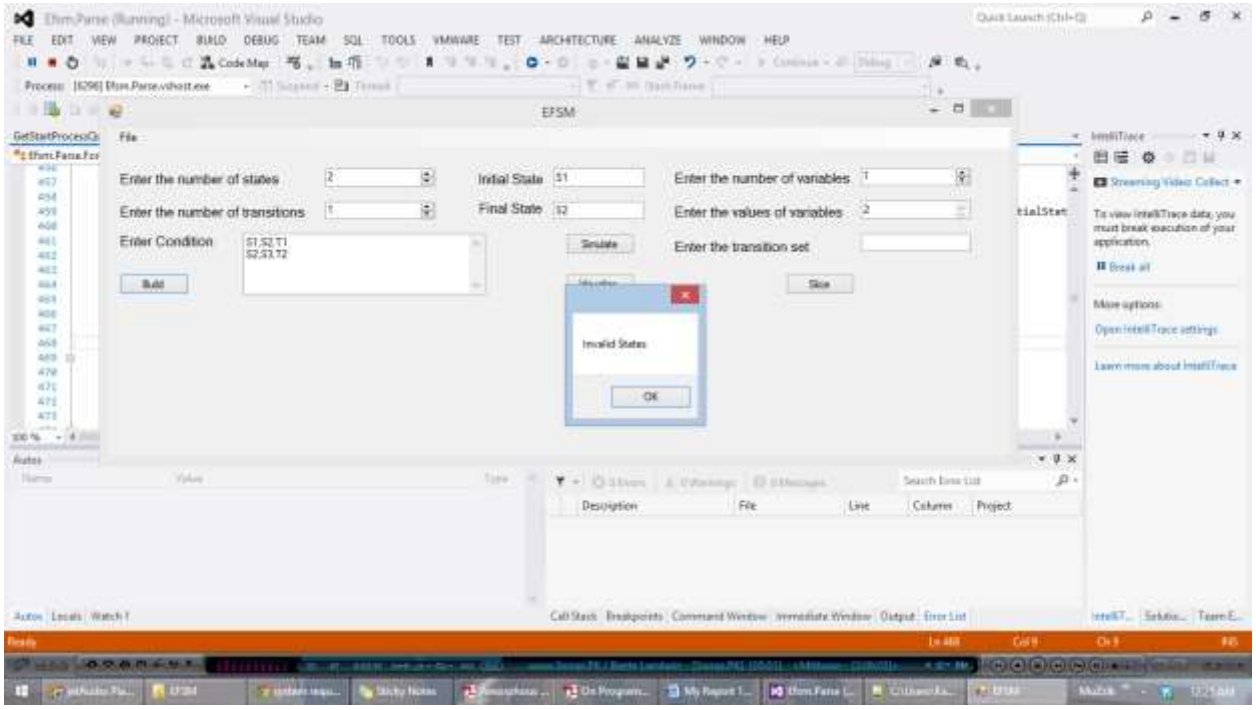

Figure 6.19 Invalid Number of States Exception (Input 8)

Ninth Test

Consider the case when the user enters a value in the number of transitions field and enters more number of transitions than mentioned in the "condition" field. The GUI throws an exception. Figure 6.20 shows the exception.

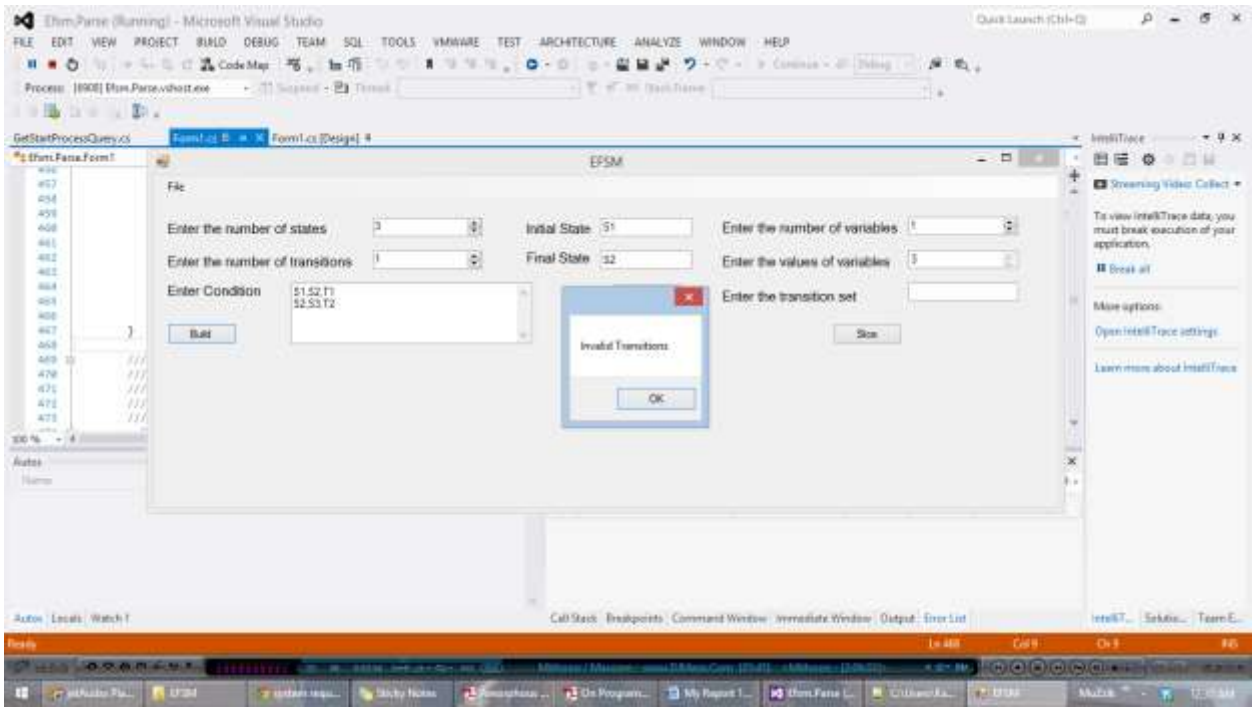

Figure 6.20 Invalid Number of Transitions Exception (Input 9)

Tenth Test

The GUI throws an exception when the initial state field is left blank.

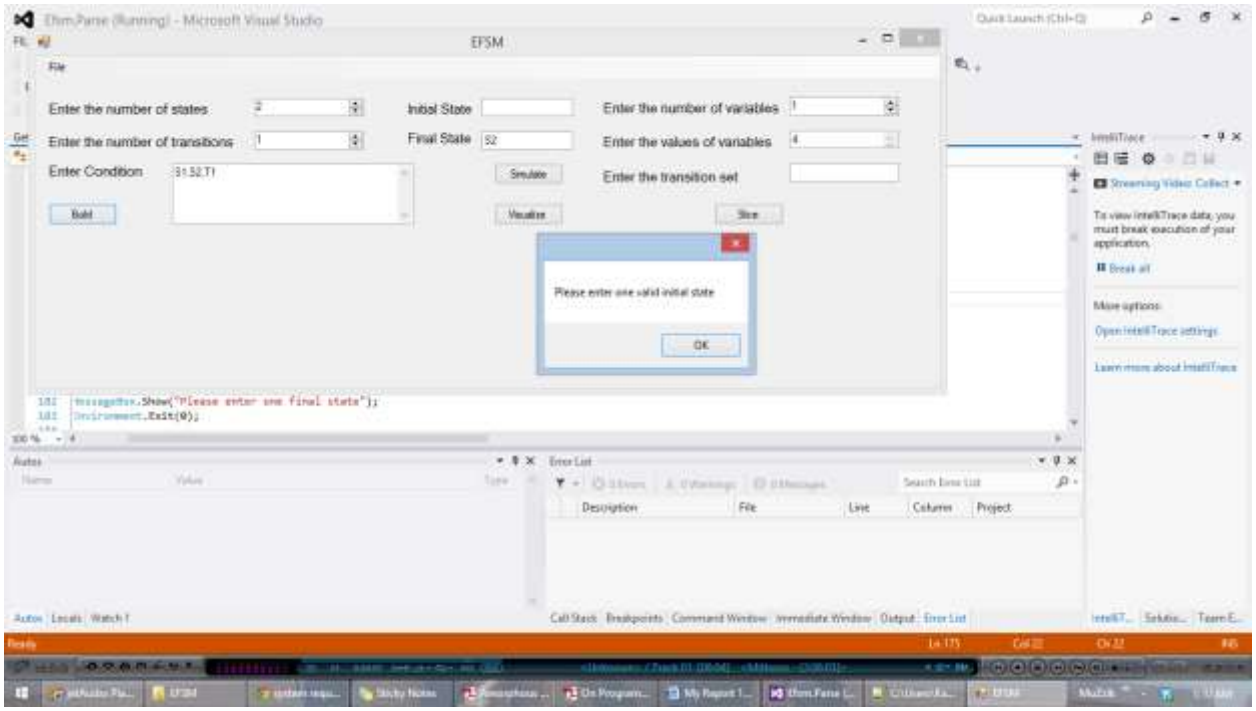

Figure 6.21 Initial State Exception (Input 10)

Similarly, the GUI throws an exception when the final state field is left blank.

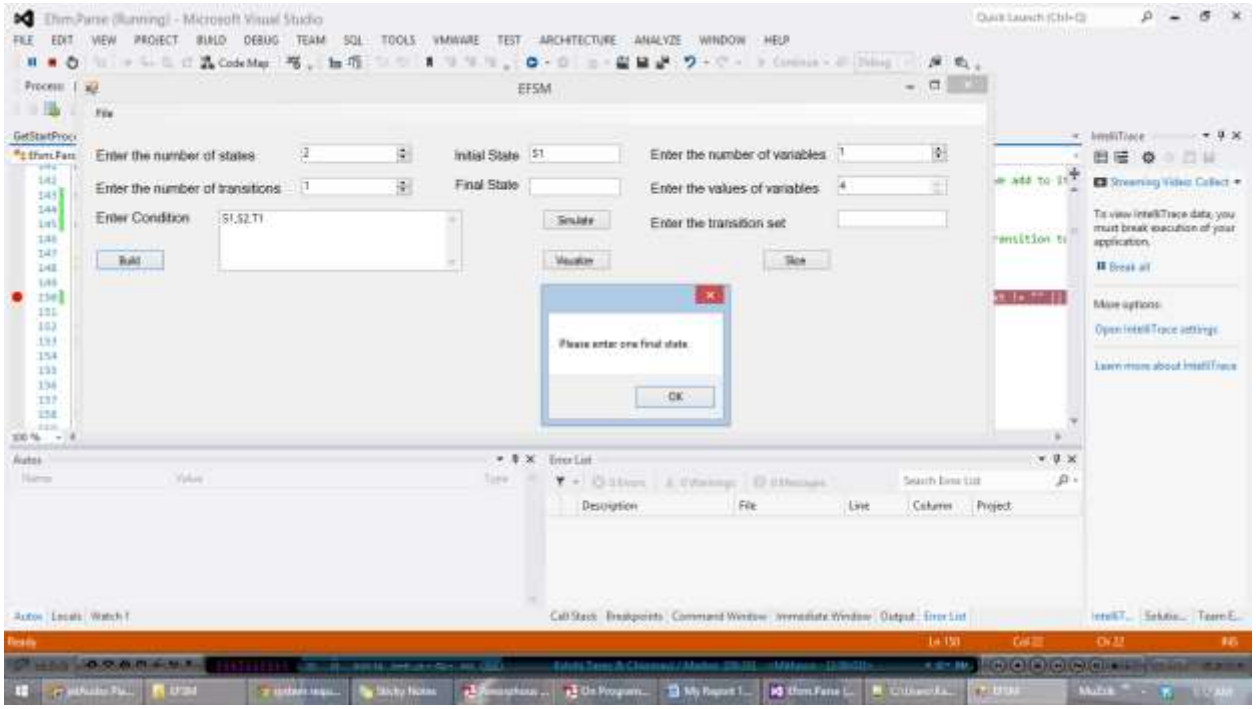

Figure 6.22 Final State Exception (Input 10)

The GUI throws the following exception when both initial and final state fields are left empty.

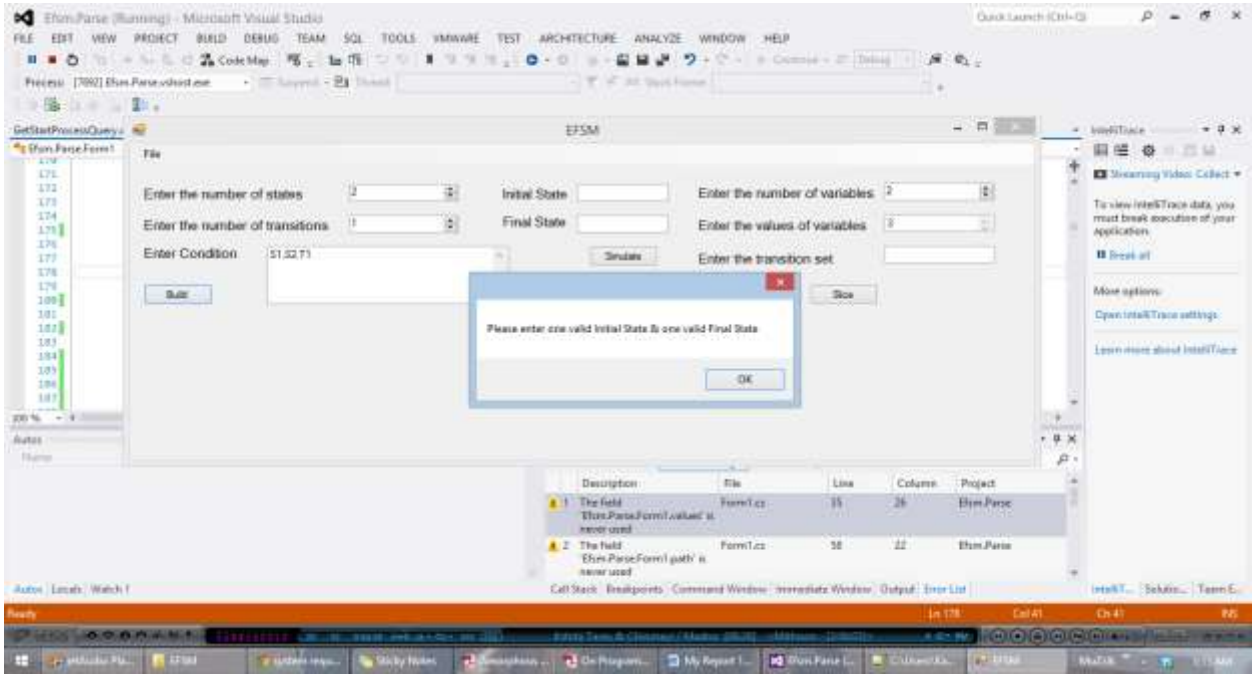

Figure 6.23 Initial & Final State Exception (Input 10)

Eleventh Test

Consider the following input:

S1,S2,T1

S1,S5,T2

S1,S4,T3

S3,S1,T4

S2,S3,T5

S3,S6,T6

S4,S3,T7

S4,S5,T8

S5,S6,T9

S4,S6,T10

Figure 6.24 shows the graph for the input

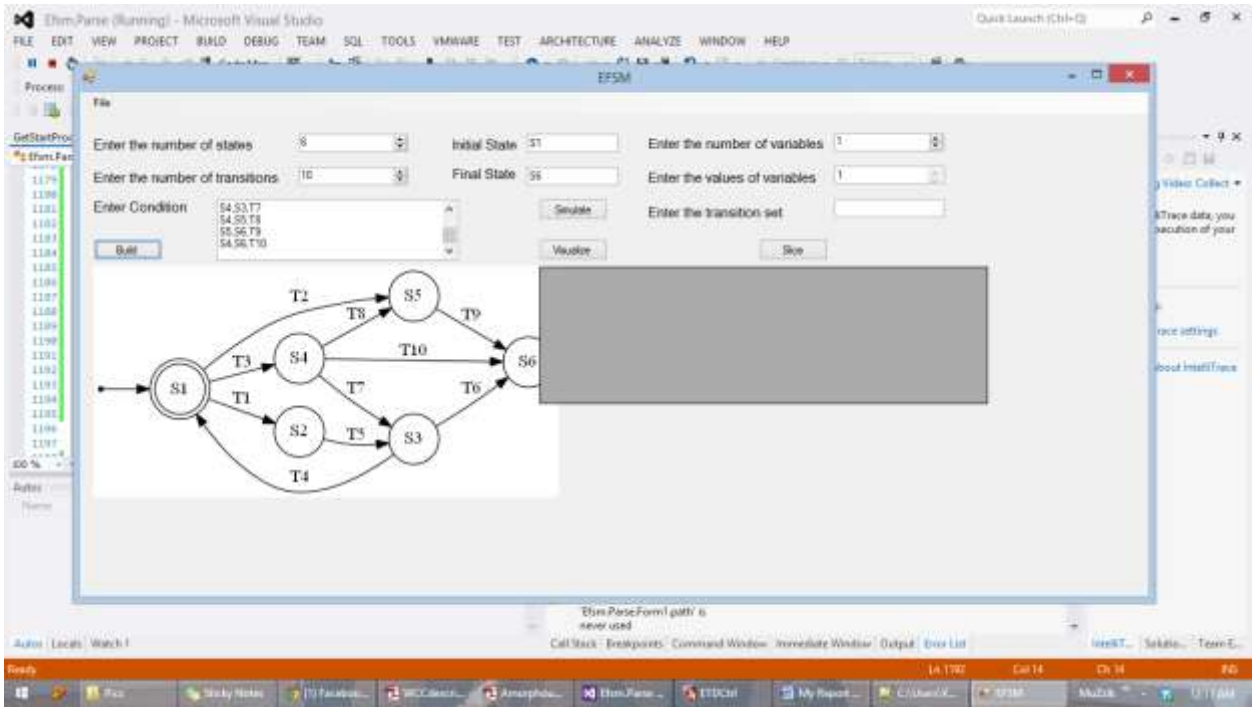

Figure 6.24 Building Graph (Input 11)

See figures 6.25 and 6.26 for data dependencies between transitions, the DDStar table and the sliced transitions.

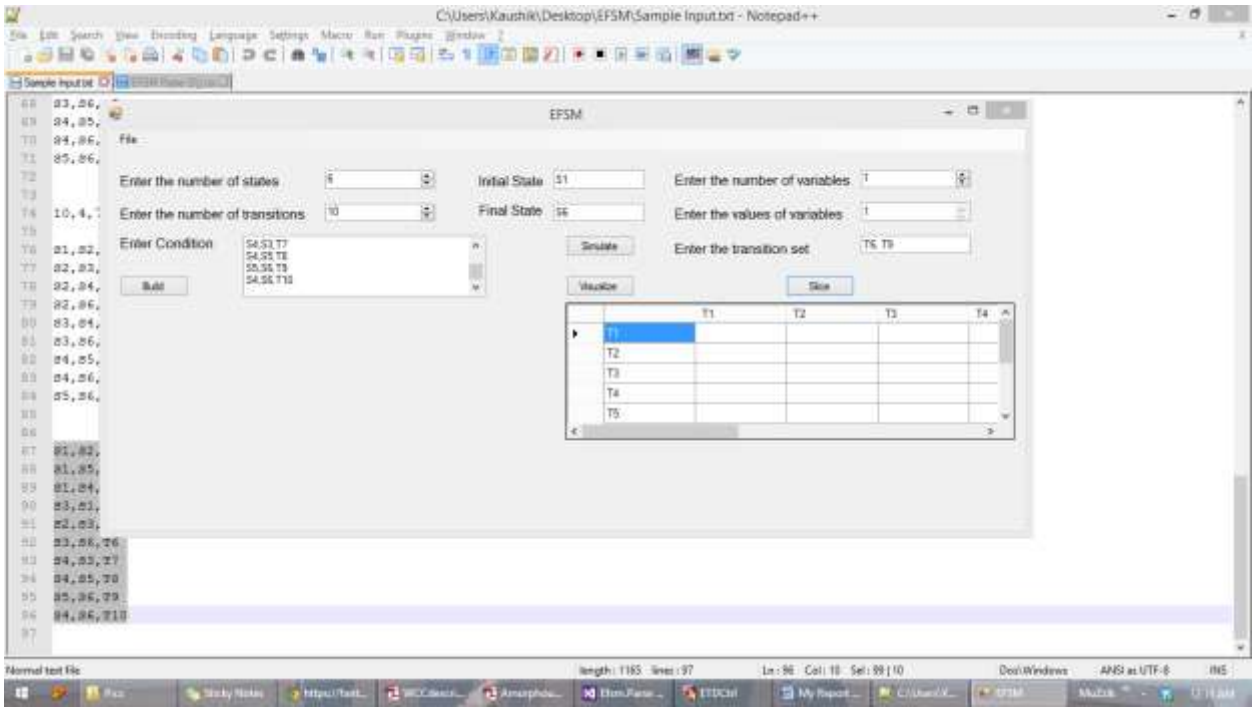

Figure 6.25 DDStar (Input 11)

The table DDStar is empty as there are no data dependencies between transitions.

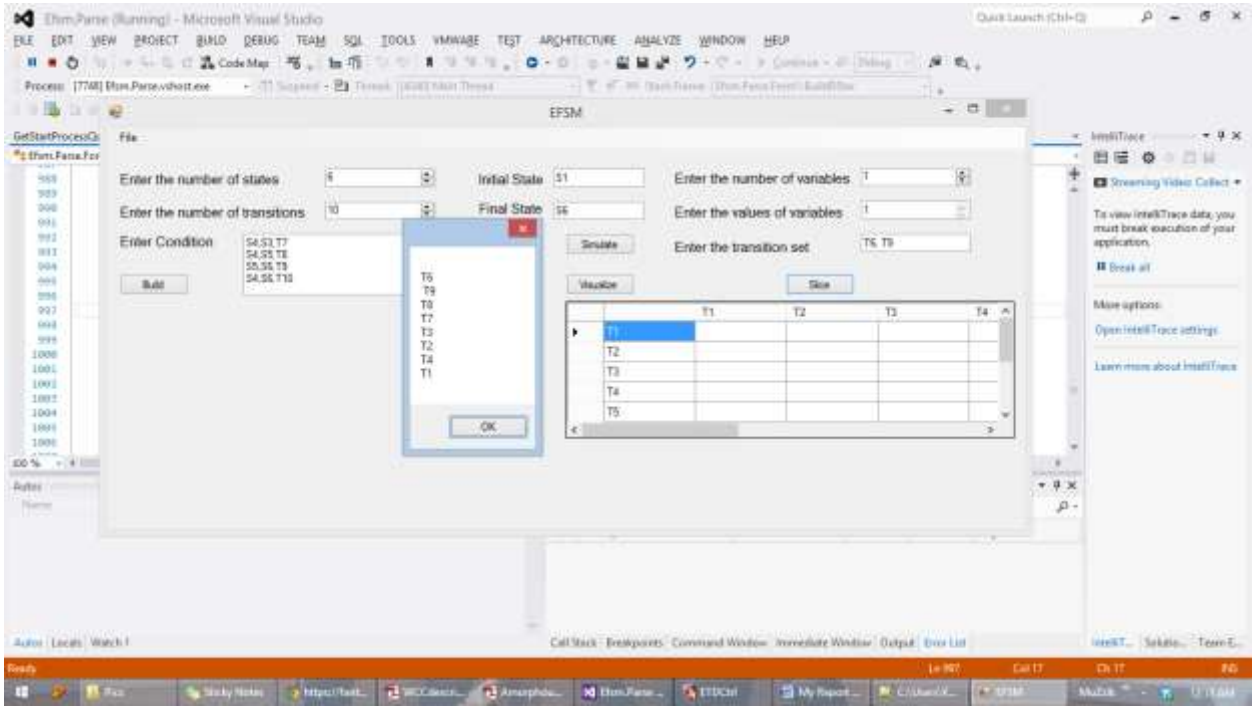

Figure 6.26 Sliced Transitions (Input 11)

The output produced is correct because it is similar to the result mentioned in [2].

Twelfth Test

Consider the following input where data dependence and DDStar are computed and the slicing algorithm is then applied.

The initial values of the variables are A = 10, B = 4, C = 7, D = 3, E = 12

S1,S2,T1,A>B,B=4

S2,S3,T2,B<C,D=6

S2,S4,T3,A>C

S2,S6,T4,D<E

S3,S4,T6,A>D

S3,S6,T5,A<E,B=7

S4,S5,T7,A<E

S4,S6,T8,C>B

S5,S6,T9,D<C,B=1

Figure 6.27 shows all the data dependent trasnitions.

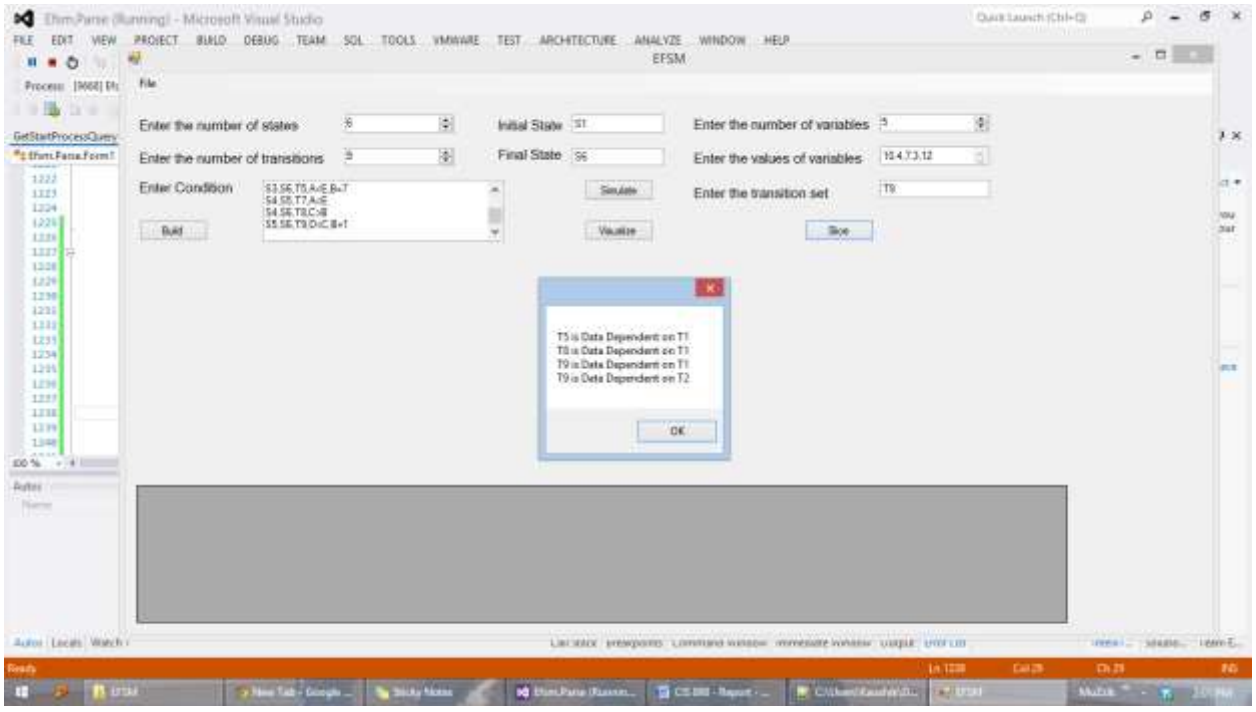

Figure 6.27 Data Dependent Transitions (Input 11)

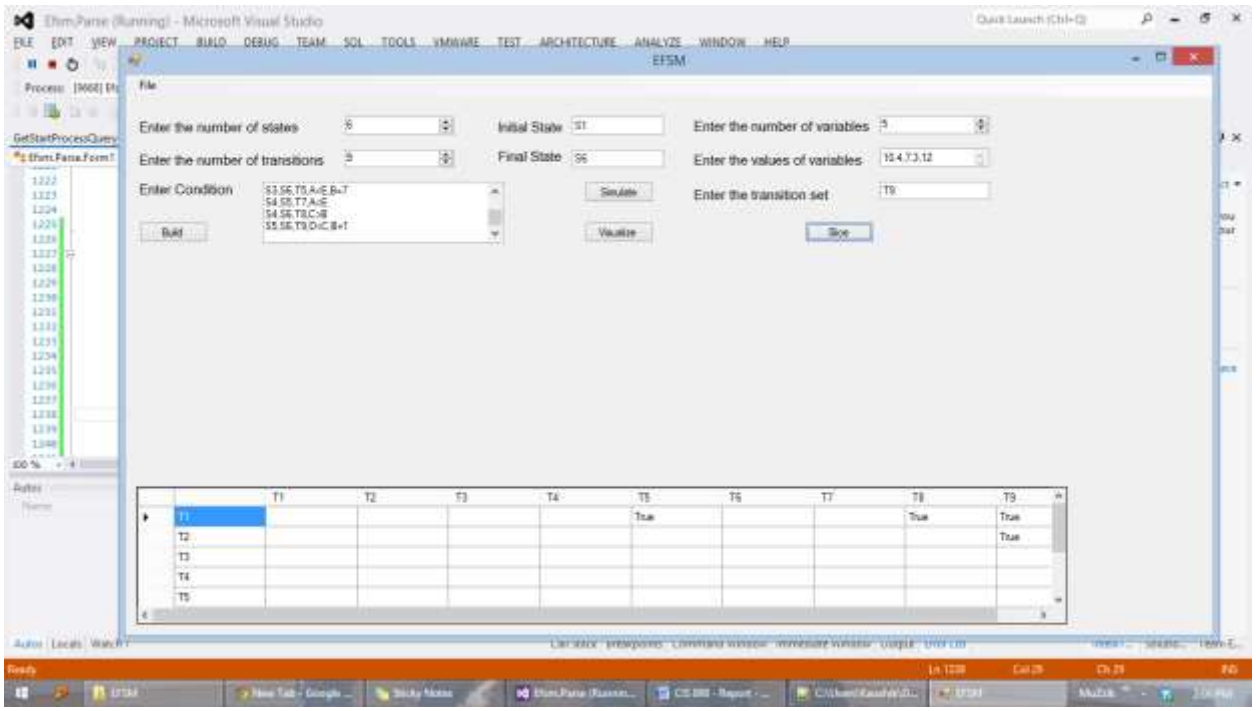

Figure 6.28 Table DDStar (Input 12)

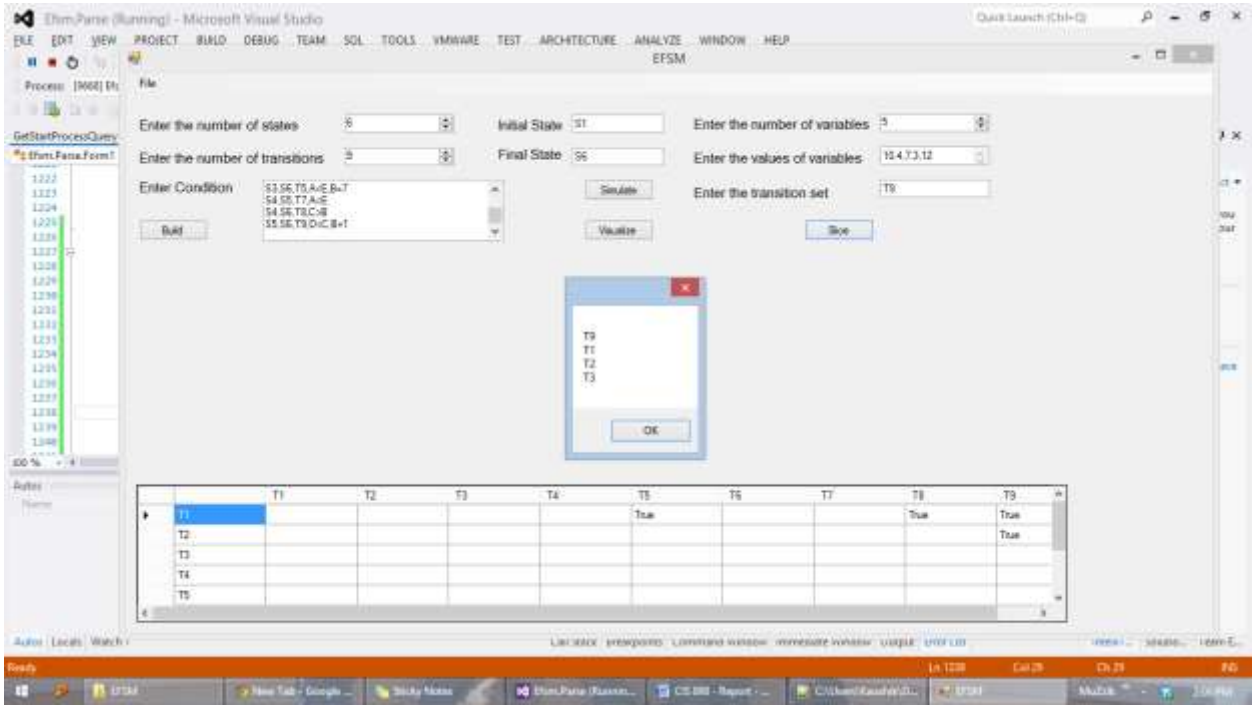

Figure 6.29 Sliced Transitions (Input 12)

References Or Bibliography

1. Kelly Androutsopoulos, David Clark, Mark Harman, Robert M. Hierons, Zheng Li, Laurence Tratt, Amorphous Slicing of Extended Finite State Machines
2. Torben Amtoft, Slicing of Extended Finite State Machines (Extract from a technical paper)
3. http://en.wikipedia.org/wiki/Extended_finite-state_machine : Extended Finite State Machine (Wikipedia)