Constructing of Statecharts from Flat State Machines

Research · January 2011

CITATIONS
0

READS
510

1 author:

Abdullah Alsaeedi
Taibah University
52 PUBLICATIONS **1,074** CITATIONS

SEE PROFILE

The University Of Sheffield.

Constructing of Statecharts from Flat State Machines

Author: *Abdullah Alsaeedi*

Registration Number: *100130052*

Course: *MSc in Advanced Software Engineering*

Supervisor: Dr. *Kirill Bogdanov*

(Dissertation Report)
30th August 2011

This dissertation is submitted in partial fulfilment of the requirement for the degree of MSc in Advanced Software Engineering.

Declaration

All sentences or passages quoted in this dissertation from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). I understand that failure to do these amounts to plagiarism and will be considered grounds for failure in this dissertation and the degree examination as a whole.

Name: Abdullah Ahmad Alsaeedi

Signature:

Date: 30th August 2011

Abstract

In software development, UML (unified modelling languages) notation is widely used to allow developers to represent systems analysis, design, and viewing software systems from different aspects. Behavioural modelling is describing using state machines in object oriented design. statecharts are more suitable to represent dynamic behaviour of reactive systems than finite state machines (FSM) which have many limitations compared with statecharts. There are many tools to construct statecharts from different UML models such as synthesis statechart from scenarios, and so on. Transformation of FSMs into statecharts diagram can be done by introducing hierarchy and concurrency into FSMs to enhance modelling reactive systems to overcome restrictions in FSMs.

This works aims to develop a tool to construct statecharts from FSMs; this was based on studying existing techniques and possible approaches of implementing this conversion, and observing the impact of generating composite states, nested states and overlapping between them into the generated statecharts on the number of transitions. Deciding which of different techniques are suitable to construct hierarchy and concurrency has derived depending on study case and measuring the obtained results using some metrics.

Of all the metrics that can be used to compare statecharts and FSMs, the number of transitions seemed most suitable as a guide to the introduction of higher-level states. This tool has been implemented and evaluated using 727 random FSMs, this has demonstrated that described approach is applicable to a range of FSMs but the reduction of the number of transitions has been rather limited.

Acknowledgments

First and foremost, I would like to thank God for helping me to finish this work and for giving me patience to complete this dissertation.

I would like to express my sincere thanks and gratitude to my supervisor, Dr. Kirill Bogdanov, for his help, advice, support, constructive feedbacks, and enthusiasm throughout this project.

I sincerely thank my parents, and my family on supporting me during my study period. This dissertation would not have been possible without their encouragements.

Thanks to all my friends for their moral support.

Finally, I would like to thank the Saudi Government and Taibah University for giving me the opportunity to study at the University of Sheffield.

Table of Contents

Table of Figures

List of Table

Chapter 1: Introduction

The Unified Modelling Language (UML) is used widely as a standard notation to represent the object-oriented software in different ways: Graphical representation (UML notations), or textual language (XMI language). There are different diagrams used to model a system during the life cycle of development systems.

In recent years, reactive systems have frequently been modelled using different kinds of state machines. Finite State Machines (FSM) is considered easy to use; it has faced several issues in modelling complex reactive systems. The lack of supporting concurrency has led to a state explosion problem. The complexity of FSMs has increased by representing multiple inputs and outputs, while the number of variables increases.

To overcome the complexity expressed in a FSM, statechart formalisms are used to enhance an FSM by concurrency, hierarchy, and state history. Different papers discuss the semantics of statecharts; Harel [16] is one of those concerned with statechart semantics. A statechart diagram can be used to model the event-state systems with a reduction of the complexities that were found in the state transition diagram.

Transformations between UML models have been introduced for different aspects such as obtaining other diagrams in different phases in software development to represent information in different views, and reducing complexities in modelling difficult systems. The complexity of modelling reactive system behaviours has motivated developers to reduce it; Ashish [23] is one of those concerned in extracting statecharts from FSMs by using techniques introduced by Harel [16,18] to simplify its representations.

2.1 Objectives and Aims

The main objective of this project is to develop a system that constructs simplest statecharts from flat state machines by introducing (concurrency and state hierarchy) to an FSM while preserving system behaviour. The developed system is able to load a valid state machine, (which is drawn and exporter form ArgoUML), and generate statecharts with concurrency and hierarchy.

In addition, experiments will be used to find out efficient techniques to achieve our objective. Evaluating outcomes of the developed system is one of our aims to observe whether the implemented methods succeed in reducing complexities or not.

Chapter 2: Literature Review

This section introduces the common aspects of statecharts and state machines. In addition, some disadvantages of flat state machines will be described. In addition, this section summarises some tools that will be used later in the research.

2.2 Concepts

2.2.1 State Machine

A state machine consists of states that represent system behaviours and activities as input and output pairs. As a result of the lack of concurrency in state machines, the number of states and transitions increases with the difficulties in modelling complex systems [6]. The cause of increasing number of transitions and states in state machine is a system might have multiple variables n with different values z be able to have z^n states.

An FSM is a model used to represent systems behaviour using states and transitions between them. An FSM is flat state machine in structure which means does not support concurrency in states and there is no hierarchical representing as well. An FSM is shown using state-transition diagram, which is a more complicated representation to understand the system's behaviour. This is especially true with complex systems, which require more states and transitions to represent the system correctly. Figure 2.1 shows a simple state machine for a cafe machine a series of states that cafe system might be in. The complexity of an FSM model increases according to the number of states in it in order to huge number of variables, which leads to some limitations [5, 16].

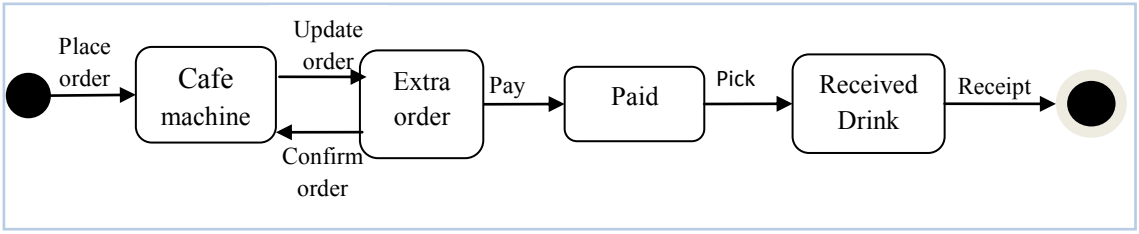


Figure 2.1: simple example finite state machine

State-transition diagram (state diagram) is the directed graph that represents state machine with circled nodes representing states and directed arrows showing transitions, labelled with events and conditions.

An FSM diagram is represented with M =(S, S0, T, δ), where 'S' is the collection of states and 'S0' is the initial state. In addition, 'T' is the set of transitions in the model and δ: T x S is the next state function, which determines the next state by giving it an input and the current state.

Figure 2.2 shows an FSM with six states and fifteen transitions, labelled as $t_i \in T$, as events that cause the state to be changed. On the other hand, Figure 2.2 illustrates that a state machine has increased its complexity, while the number of transitions can be reduced by introducing statechart hierarchy and concurrency [6].

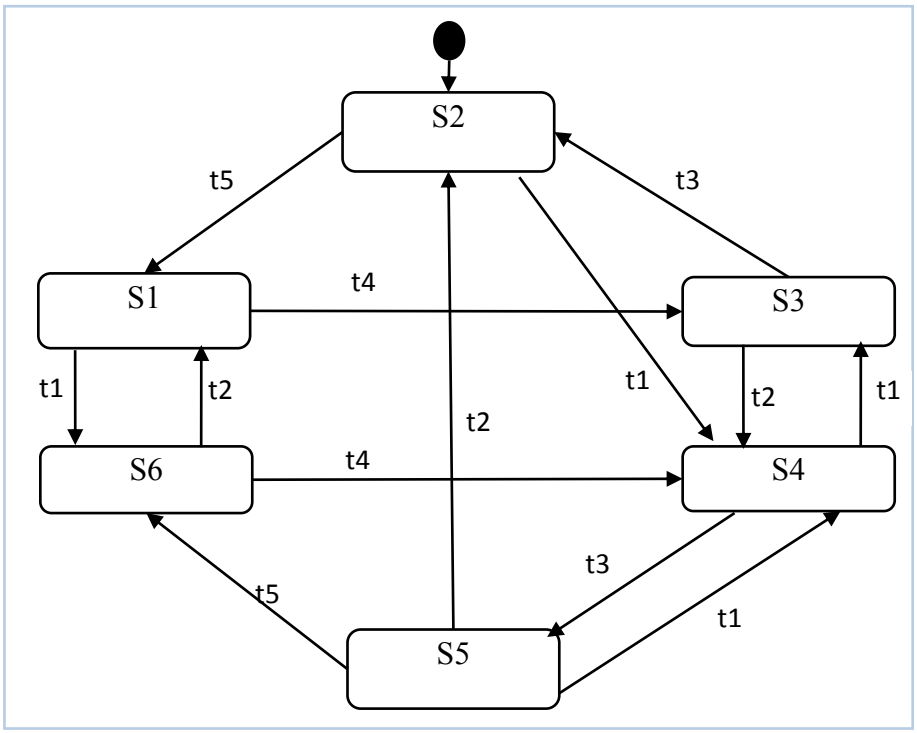


Figure 2.2: an example of a finite state machine

2.2.1.1 State-Transition Table

State-transition table is a scheduler representation of FSM which depends on states, transitions, and events. The following table demonstrates the states-transitions diagram shown in Figure 2.2.

Table 1: State-Transition Table of FSM

	S1	**S2**	**S3**	**S4**	**S5**	**S6**
T1	S6	S4	-	S3	S4	-
T2	-	-	S4	-	S2	S1
T3	-	-	S2	S5	-	-
T4	S3	-	-	-	-	S4
T5	-	S1	-	-	S6	-

2.2.1.2 Reachable and Unreachable States

One of the main properties in validating an FSM is reachability which is mean each state has a path from its initial state to ensure every state in an FSM has visited. Figure 2.2 demonstrates the concept of reachability of FSM where all its states are reachable because they have a path from its initial state. However, Figure 2.3 shows an example of unreachable states; state *S2* is unreachable state, no path from the initial state to state *S2*, while it has incoming and outgoing transitions.

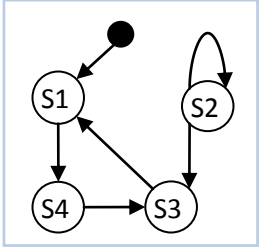


Figure 2.3: unreachable state

In addition, another crucial validation of FSMs is each state should not have more than one outgoing transition with the same label.

2.2.1.3 Limitations of State Machines

A flat state machine has a lot of limitations, which are mentioned in [6, 16]. Firstly, a huge number of states and transitions are shown in a complex system, increasing the visual complexity of the system; this makes reading, managing, visualising, scaling and modelling system behaviour more complicated. Finally, the number of transitions can be reduced by introducing hierarchy and concurrency to statecharts [9, 19]. Douglass [6] has described the drawbacks related to a FSM; it has a lack of support for concurrency, which leads to increased numbers of states and transitions, and it does not have a hierarchy in states where all states are on the same level. It is difficult to maintain without knowing the design of state machine well.

2.2.2 Statechart

Statecharts diagram is one way of representing the behaviour of an object in any system by representing states and their interactions with others [16, 18]. Statecharts representation described by Harel [16] as a visual representation for states and transitions, with concerns for hierarchy and concurrency concepts.

Harel [16] summarised the concept of statechart using the following formula:

Statecharts = state diagram +depth (hierarchy approach) + orthogonality (concurrency) + broadcast_communication.

According to Harel's equation defined above, there are some additional elements to state diagrams that make statecharts used to model such systems [16].

Figure 2.4 shows a simple example of a statechart to represent a cafe machine, where concurrency concept is shown below such as adding extra items in ordering coffee requires updating the price concurrently. Diego *et al* [5] describe statecharts as a network of its states and events.

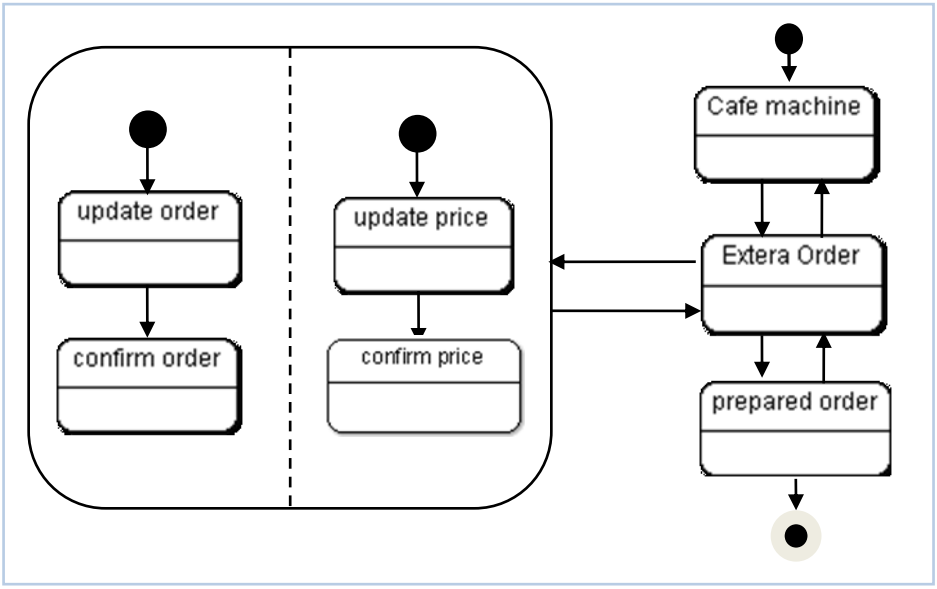


Figure 2.4: simple example of statecharts

2.2.2.1 Basic Components of Statecharts

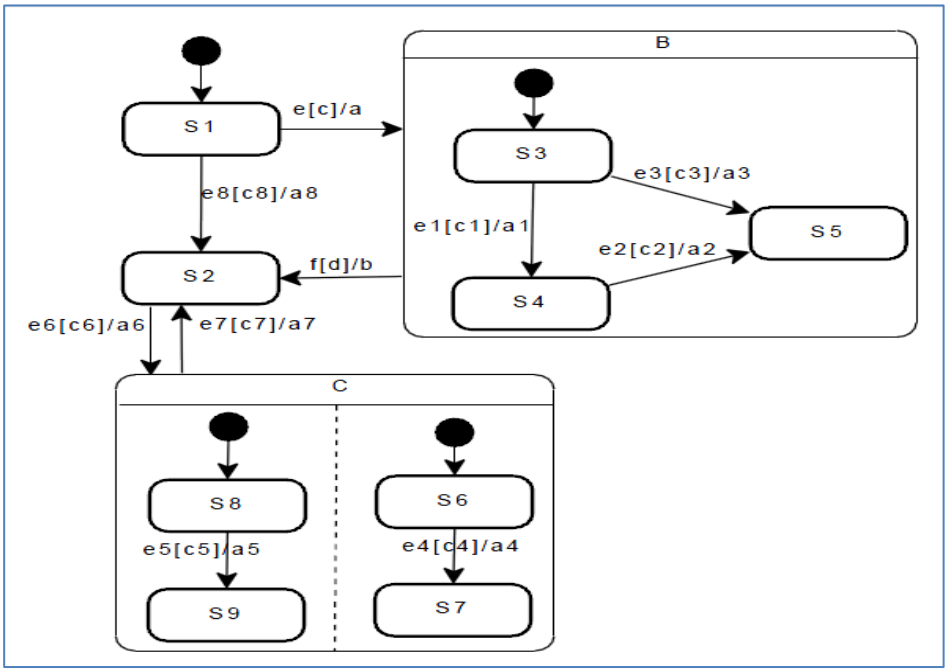


Figure 2.5: statechart components example [24]

2.2.2.2 States

A state represents the status of an object in a system during the life of objects. States are shown by rectangles in a statechart diagram. There are three types of states in a statechart diagram representation: OR-states, AND-states and basic states (simple states). In addition, there is a distinct state called the root state, which has no parent, and this kind of state is at the highest level. All these kinds of states are shown in Figure 2.5, e.g. state *S1* is a basic state and a root state. Different kinds of states are described below with details [10, 16].

2.2.2.2.1 AND-states

AND-states are those states that represent concurrent components, which might be OR states or simple statecharts within the concurrent state. A dashed line is used to represent the AND relation between states. Furthermore, if the AND-state status is active during the operation of any system, the sub-states that are in the AND-state are active simultaneously. State *C* in Figure 2.5 is an example of an AND-State [16, 18].

2.2.2.2.2 OR-states

An OR-state is a state that contains a number of states, only one of which may be active at the same time. This kind of state is very helpful in achieving the aim of constructing a statechart from a complex state machine. State *B* in Figure 2.5 is an example of this kind of state [16, 18].

2.2.2.2.3 Basic states

The basic state is a state that does not have sub-states. Such states can also be called simple states, such as state *S2* in Figure 2.5 [16].

2.2.2.2.4 Composite states

A composite state is either an AND-state or an OR-state. In addition, using the composite state allows statechart to be hierarchical and the number of basic (simple) states to be merged. States *C* and *B* are examples of composite state in Figure 2.5 [16, 18, 23]. The important and benefit of composite states is described by Cruz-Lemus *et al* [2], as the comprehensibility increases when using composite states in statechart diagrams and some experiments have shown how the composite state enhances the comprehensibility [5].

2.2.2.2.5 Initial and final states

A statechart has an initial state to represent the start of the system and an OR-state can have its initial state. It is shown as a filled circle. A final state is a state that is entered to indicate that a specific state has completed its execution. Figures 2.6 and 2.7 illustrate initial and final states as below.

Figure 2.6: initial state

Figure 2.7: final state

2.2.2.3 Transitions

Transitions show how a system transforms from one state to another according to the event that occurred. It is represented as an arrow with a label describing an event and a condition that have to be satisfied to make this transformation in system status. A transition label is expected to have a form of e[c]/b, where 'e' is the event and 'c' is the condition. When 'e' and 'c' is true, this cause the transition to be taken to reach the destination state. There are different forms of transition, such as simple, self, and compound transitions. Figure 2.8 shows how the simple transition will be made and actions will be executed; the system will enter a target state of the transition. A self-transition is a transition from a state to itself, such *t1* in Figure 2.8.

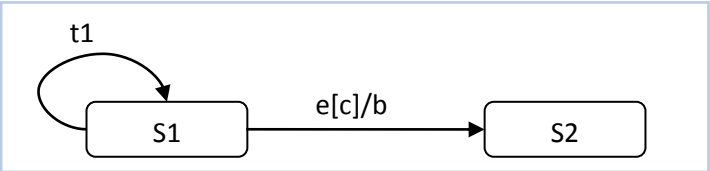


Figure 2.8: simple and self-transitions

Figure 2.9 shows a compound transition such as that between *t1* and *t2* linked by connector to form a single transition; *t1* cannot be executed without executing *t2* or *t3* [16,18,14]. There are different forms of compound transition such as AND, OR compound transitions.

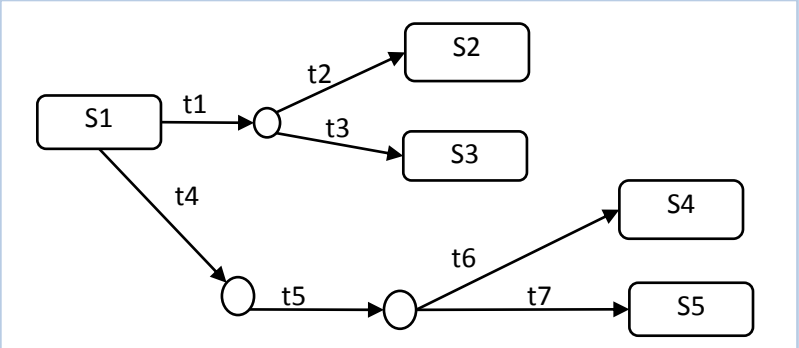


Figure 2.9: compound transition

2.2.2.4 Events

The event ‘e’ in any transition labelled by e[c]/b is the cause that makes a system switches from one state to another. Events could be internal or external; the internal events are those that pass among instances of the system (such as call events within the same instance), whereas external events relate to when the whole system receives an event from an external instance within the same environment. There are different types of event depending on the effect occurring, such as **time event** is caused by after the passage of certain time after particular event and a **call event** is used to invoke a method to execute a specific operation [18, 29].

2.2.2.5 Conditions

A condition is a boolean expression that might be true or false. It is shown within brackets in transition label [c] and is used to allow the transitions to be taken when ‘c’ value is true and event ‘e’ occurs [18].

2.2.2.6 Actions

Actions are executed when a transition is taken. There are two kinds of action to represent the way of executing; an **entry action** is executed when the current state is entered and an **exit action** is performed if state is exited [33].

2.2.2.7 History

A composite state might contains a history state which is used to resume a state that had visited before system switches to another composite state rather than starting again form the default state. Figure 2.10 shows a history component in the composite state *S6*: this history is applied to the hierarchy at the same state level. [16].

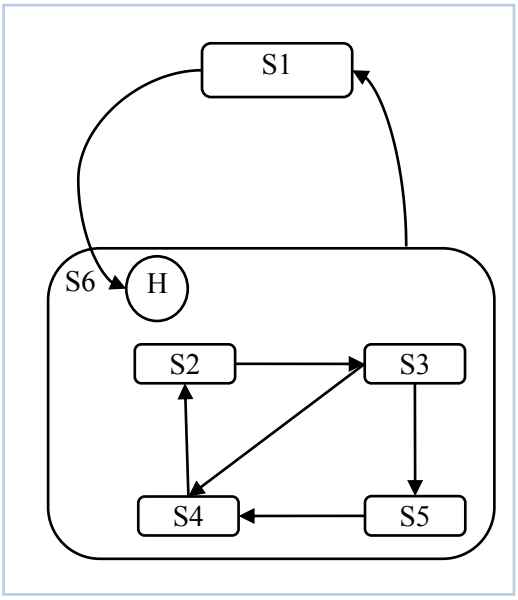


Figure 2.10: history state in statecharts

2.2.2.8 Connectors

There are two kinds of connectors: ‘AND’ and ‘OR’ connectors. They are used to link compound transitions. Once a compound transition is linked with a series of transitions by an AND-connector, any transition connected with the compound transition is linked with all that participate in it. An OR-connector is linked to a compound transition with at least one

transition in its segment. Figure 2.12 shows an example of OR-connector where *t2* is follow by *t3* or *t4* to be executed; however AND-connector shown in Figure 2.11 should be composed of *t3* and *t4* to be carried out [19,23].

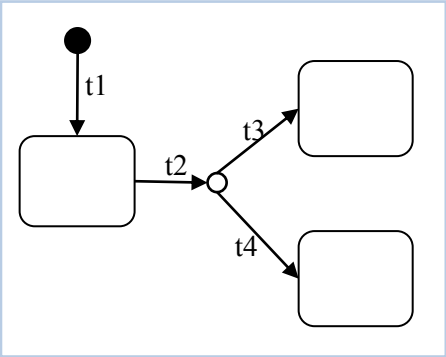


Figure 2.12: OR-connector

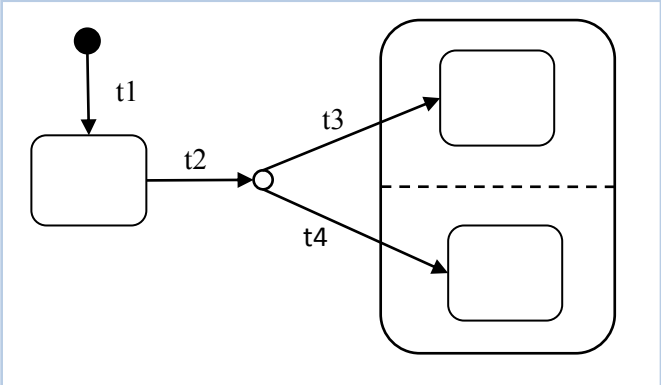


Figure 2.11: AND-connector

2.2.3 Concurrency and Hierarchy of Statecharts

Hierarchy is introduced into a statechart by representing a composite state; in Figure 2.5, composite state *C* is higher than state *S7* in the hierarchy. The purpose of using this kind of hierarchy is to make statechart more elegant, providing the ability to maintain and understand statechart design easily, especially when it becomes complex [31, 9, 5]. Moreover, the benefit of applying hierarchy or depth in abstraction is to provide an easy way to manage the design of complex systems [19]. Defining the number of nested state (the depth of hierarchy) to avoid unreadable statecharts.

Concurrency means that multiple transitions with different actions are taken concurrently [7]. Concurrency within statecharts provides the ability to manage many states simultaneously. In addition, concurrency is represented with AND-states, where its components are concurrent objects in the system's behaviour. The method of managing the number of states is the use of concurrent states to group related states together [35, 8].

Moreover, as described above, the importance of statecharts is to reduce the visual complexity of state machines by introducing concurrency and hierarchy. Harel [16, 18] showed the impact of using the AND and OR composite states on improving state machine; Figure 2.5 shows an example of a statechart using the AND-state and OR-state. The state *B* is an OR-State and the state *C* is an AND-state with a dashed line to represent it [16].

Horrocks [20] describes the importance of statecharts in different aspects such as statechart has ability in avoiding duplication in transition compared with state machine diagram. Moreover, states in statecharts are represented in hierarchal form with consideration to levels of abstraction, where the number of states in statecharts is moderate (unlike in state machines) [20].

2.2.4 Clustering

Clustering is shown by Harel [16] to reduce the number of transitions and cluster states into a composite state. For example, in Figure 2.13 there is an event that causes the system to move from state *S1* to state *S2* and *t1* takes the system to state *S3* from *S1* or *S2*, clustering states *S1* and *S2* to a new super-state *S4* leads to the common transitions *t1* collapsing into a single transition for the composite state (see Figure 2.14). A transition that leaves the clustered state is referring to all states within clustering such as *t1* in the Figure 2.14. Clustering might be helpful to achieve this research aim of constructing statecharts from FSMs.

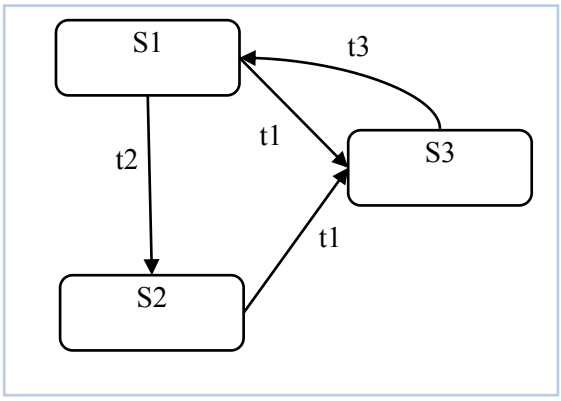


Figure 2.13: statechart without clustering

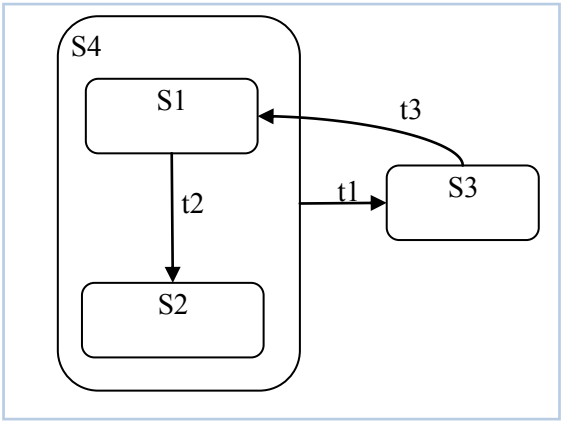


Figure 2.14: statechart with clustering

There is a special way of forming clustering that might occur in statecharts which is called recursive clustering. Harel introduced this concept in [16] to construct composite states and therefore hierarchy are introduced. Figure 2.15 show an example of a recursion clustering such as 'cluster 2' and 'cluster 3' are recursive clusters for 'cluster 1'. The importance of using recursive clustering in our project is to achieve the aim of reducing the number of transitions in the generated statechart from FSM.

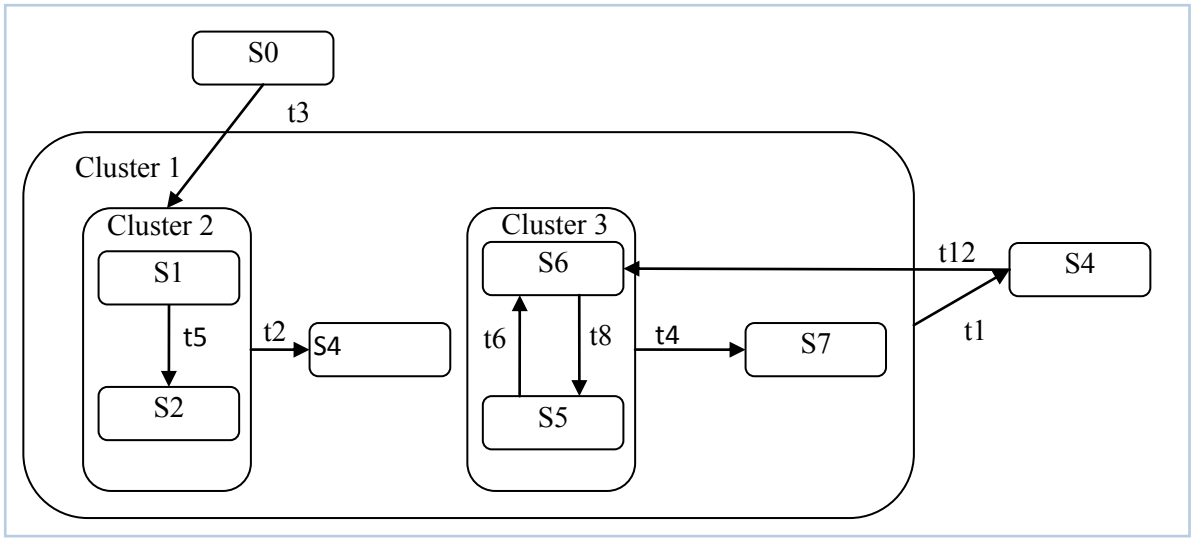


Figure 2.15: statechart with recursive clustering

Overlapping

Overlapping state is a state which is present in multiple composite states such as state *S2* in the Figure 2.16 that is an overlap between two composite states, whereby reducing the number of transition is shown. Overlapping is introduced to avoid duplicated statecharts inside composite

state. Overlapping is helpful for setting priorities among states and clustering states by common events. Entering overlapping to statecharts makes it more compact. The intersection method introduced by Harel [17].

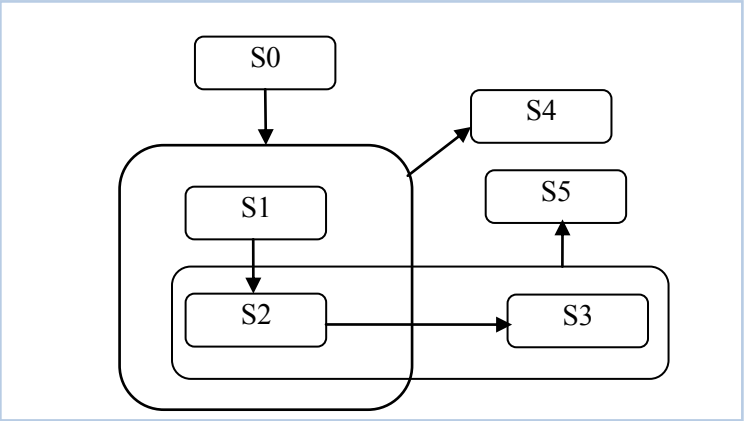


Figure 2.16: statechart with overlapping

2.2.5 Language for Modelling Reactive Systems

STATEMATE is a language for representing reactive systems described by Harel and Naamad [18]. STATEMATE shows system behaviours in sequence executions as sets of runs; each run is represented as a series of snapshots called **status**, where system status is represented. Status in this language is similar to states in Statecharts with extra information, such as conditions, events and history. Changing from one status to another is represented by executing a **step**. Figure 2.17 shows this series of status and steps [18].

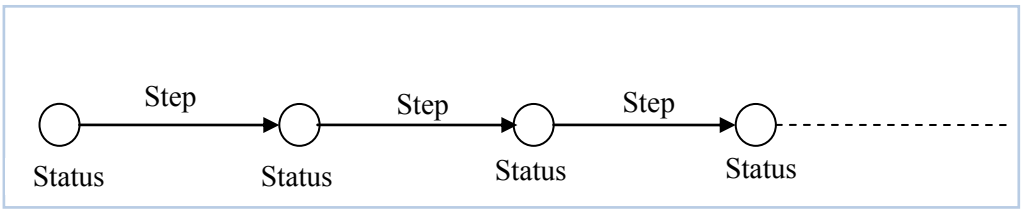


Figure 2.17: series of statuses and steps

2.2.6 Configuration

Statecharts configuration is the maximum number of states that might be in the system simultaneously [18]. Given a root state R and a configuration is a set of states C, the following rules related to configuration.

- C contains R.
- If statecharts has an OR-state (A), it must contain only one state at least of its sub-states (A's).
- If statecharts has an AND-state (A), it must contain all its sub-states (A's).
- The basic configuration is referred to the number of simple states simultaneously in a statecharts.
- If the system is in any state (A) mentioned above, it must also in the root of sub-states (A's).

2.2.7 The Scope and Priority of Transitions

The scope of a transition is described as the common ancestor of source and target states of a transition. In Figure 2.18, the scope of transitions *t1* and *t2* is 'A'; the scope of transition *t3* is 'C', and so on. Figure 2.18 demonstrates that priority is given to a transition over another in order of its scope in hierarchy, thus *t1* and *t2* has the same priority, and *t3* has a higher priority over transitions *t1*and *t2* [18].

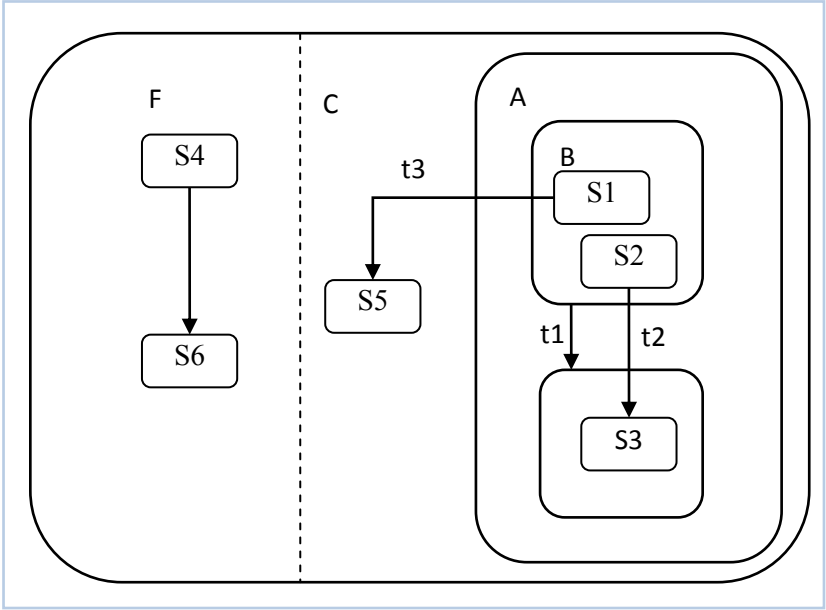


Figure 2.18: the scope of transitions and priorities

2.2.8 Transitions Conflicting

The conflict between two transitions occurs if there is a common state, which is sources of conflicted transitions, which will be exited if any one of these transitions is taken. In Figure 2.19, there is a conflict between transitions (*t1, t2*) because both of them have the same source state. If both transitions have the same source states, it means *nondeterminism*. If transition *t4* is taken, another conflict occurs with transitions *t1*, *t2*, and *t3*; however, this is not implying *nondeterminism*.

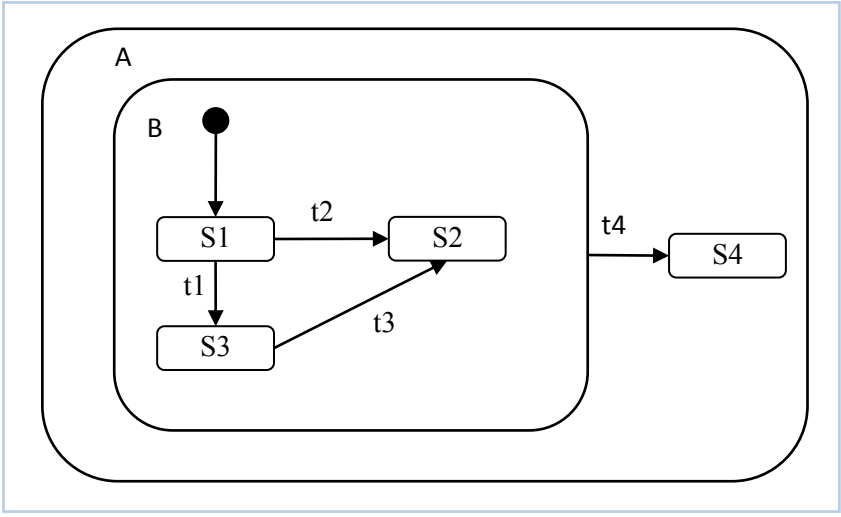


Figure 2.19: Conflicting between transitions

Treating these two kinds of conflict separately; if *t1* and *t2* are in the same step, preferring one of them is the only possible option of treating conflicts to deal with the conflicting detected between them. In the second case of conflict is treated by taken transitions with higher priorities in the same step; transition labelled with *t4* has a higher priority over all other transitions, hence *t4* does not conflict with other transitions in the same step [18].

2.2.9 Assumptions

In this project, an FSM is assumed to be consisting of initial state, simple states, and transitions linked between states. A statechart is assumed that consisting of simple states, OR-states, and AND-states. A transition in FSMs and statecharts is assumed to be consisting of events only without actions and condition.

2.3 Tools and Languages

In this section, some tools and languages that play a vital role in implementing this project will be discussed.

2.3.1 ArgoUML

ArgoUML is an open source tool used to model UML diagrams; it runs on the Java platform. In addition, ArgoUML provides a way to generate an XMI format. It has the ability to import and export an XMI format for any state machine [28]. Figure 2.20 shows the interface of ArgoUML tool for drawing statecharts and state machines, with the ability to export state machine in XMI format.

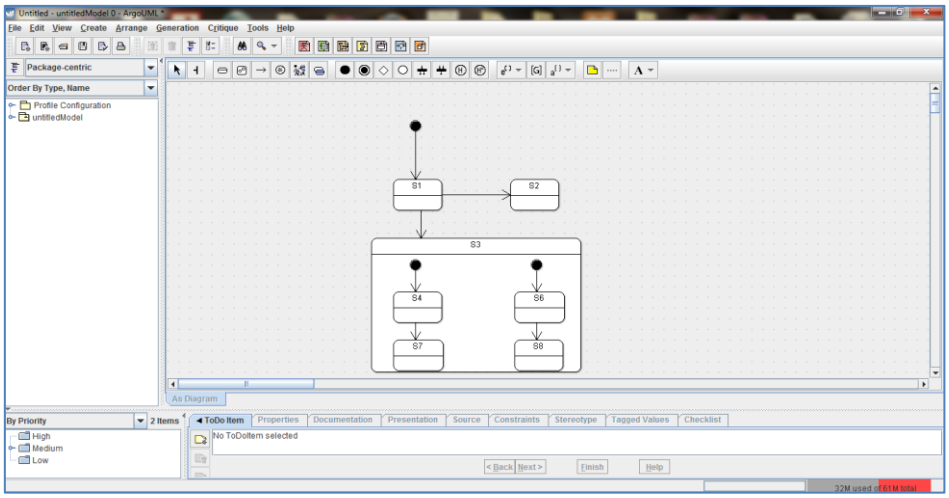


Figure 2.20: ArgoUML tool

2.3.2 XMI

XMI (XML Metadata Interchange) is a language that allows the developer to interact with objects as XML syntax. XMI is designed to represent UML diagrams using XML syntax. This language will be used as input in constructing these kinds of statecharts from flat state machines [15]. Figure 2.21 shows this meta language similarities to XML syntax.

```
<UML:StateMachine.top>
  <UML:CompositeState xmi.id = '-64--88-56-1--115ce67e:12ecefea205:-8000:0000000000000867'
    name = 'top' isSpecification = 'false' isConcurrent = 'false'>
    <UML:CompositeState.subvertex>
      <UML:SimpleState xmi.id = '-64--88-56-1--115ce67e:12ecefea205:-8000:0000000000000868'
        name = 'A' isSpecification = 'false'>
        <UML:StateVertex.outgoing>
          <UML:Transition xmi.idref = '-64--88-56-1--115ce67e:12ecefea205:-8000:000000000000086A'/>
        </UML:StateVertex.outgoing>
      </UML:SimpleState>
      <UML:SimpleState xmi.id = '-64--88-56-1--115ce67e:12ecefea205:-8000:0000000000000869'
```

Figure 2.21: XMI format of state machine

2.3.3 JDOM

The JDOM (Java Document Object Model) is an open source API used to make the interaction with XML easy; it contains different methods that allow a Java programmer to access the data and manipulate it with a small amount of coding. The JDOM will be used in our project to access states and transitions elements in XMI file easily [20, 21].

2.3.4 DOT

Graphviz is a graph visualisation tool used to produce graphs in different formats, such as JPEG and PNG, from a DOT input [34, 13]. DOT is a language for describing graphs as plain text; it is used to draw undirected and directed graphs with hierarchy. It has a lot of attributes related to each notation such as colour and shape for edges and nodes [34]. Figure 2.22 illustrates how an FSM is represented as DOT languages.

```
digraph finite_state_machine {
      node [shape = circle];
        initial -> S2[ label = t2 ];
        initial -> S1[ label = t1 ];
        initial -> S9[ label = t7 ];
        S5 -> S3[ label = t3 ];
        S2 -> S3[ label = t3 ];
        S6 -> S1[ label = t6 ];
        S9 -> S6[ label = t5 ];
        S3 -> S2[ label = t9 ];
        S1 -> S6[ label = t6 ];
        S2 -> S5[ label = t4 ];
       }
```

Figure 2.22 DOT syntax of FSM

2.4 Evaluation

In this section, metrics will be illustrated to show how they might be used in this research to evaluate the complexity of statecharts. Some of them will be used to measure the generated statecharts, comparing them with state machines they were built from. The following list shows metrics for measuring statecharts generated from state machines:

1- Number of entry actions in a state machine, which is performed by comparing those in state machine with those in the corresponding statecharts.
2- Number of states in a statechart.
3- Number of transitions in a statechart, including the compound transitions.
4- Number of conditions and events in statechart.
5- For measuring the complexity of statechart, there are several known metrics, such as McCabe Cyclomatic complexity. This is defined by the number of simple states (NSS) and transitions (NT) as the following: NSS – NT +2 [12, 25].

6- Number of simple states inside a composites state.
7- Number of transitions that might be reduced after constructing composite states.
8- There is a new metrics are found by Cruiz [3] called Nesting level in composite states (NLCS) to indicate the number of composite states nesting in statechart diagram.

These metrics might be used in different parts of this project:

- To evaluate the complexity of a statechart produced by the converter compared to original state machine diagram.
- Finding the possible ways of merging states to form composite states using the number of transition that might be reduced with each option of merging states.

Here, an example of how these metrics might be helpful in measuring the efficiency of converting FSMs into statecharts, suppose we have the following FSM and corresponding statechart are shown in Figure 2.23.

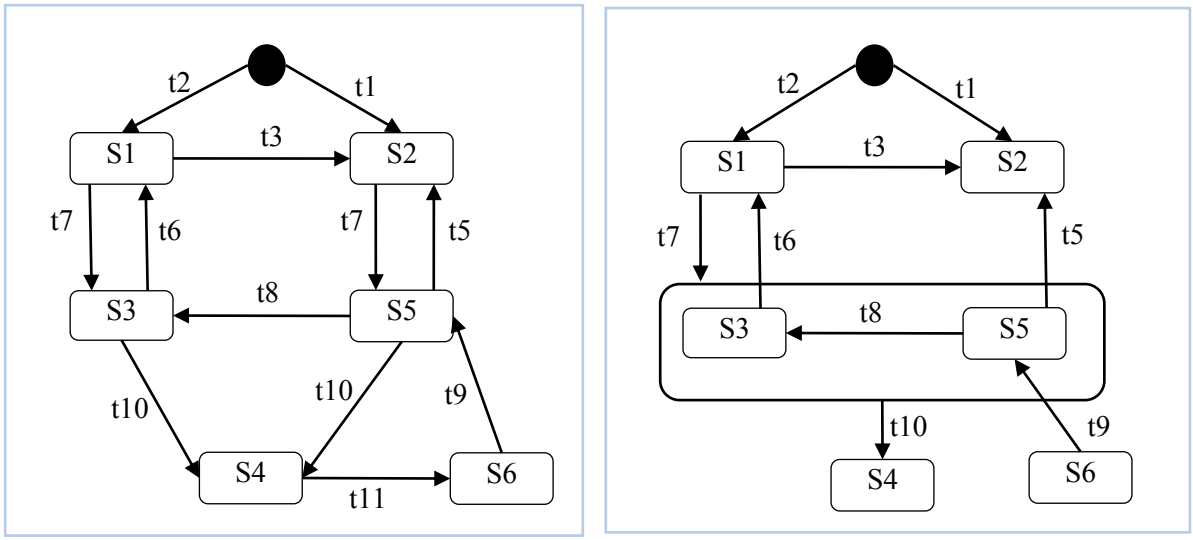


Figure 2.23: an FSM vs. a statechart for metrics

The number of states and transitions are the appropriate metrics alongside with the McCabe Cyclcomatic complexity to evaluate structural complexity, whereas other metrics not applicable according to our assumption.

Table 2: Metrics to evaluate structural complexity of FSM and corresponding statechart

	Number of simple states	Number of transitions	McCabe Cyclomatic complexity
FSM	6	12	6-12+2= -4
Statechart	6	9	6-9+2= -1

According to information presenting in Table 2, a statechart has a slightly lower complexity compared to a FSM; the number of transitions in an FSM is twelve compared to nine in the generated statechart. In addition, McCabe Cyclcomatic complexity demonstrates how the FSM has a structural complexity more than statecharts.

Chapter 3: Related Works and Case Study

3.1 Related Works

Hua Chu [1] demonstrates the technique of converting sequence diagrams to statecharts using the BK-Algorithm. The BK-Algorithm is used to reverse of state machine from scenario models. Finding program traces from sequence diagrams and then representing them as set of trace items in (a_i, e_i) format, where a_i is the sending messages between objects and e_i is the receiving message. Each trace item (a_i, e_i) is corresponding to a state in a state machine; a_i is a state action and e_i is a state event [1, 30]. After that, composite states were introduced to statechart diagrams to introduce hierarchy. The concept of clustering introduced by Harel [16] has been implemented in this work to produce composite states and this is shown successful results in introducing hierarchy [1].

Whittle and Schumann [30] introduced a way of generating statecharts from scenarios; sequence diagrams are generated to represent objects and related functions to each object according to scenario. Semantic information has been added to overcome conflicts in interpreting customer requirements as scenario has been mentioned. The important factor in this research is that generating statecharts is done by generating an FSM from multiple sequence diagrams after detecting conflicts between them. The approach is described by Whittle and Schumann [30] as below:

- Each FSM represents each sequence diagram.
- Margining multiple FSMs into one FSM by taking the union of those FSMs; this phase produces a lot of unnecessary and similar nodes.
- Merging these nodes to reduce them to get the correct FSMs without hierarchy.
- Two ways have been mentioned to introduce hierarchy to the generated statecharts [30]. One of these way will be described below:

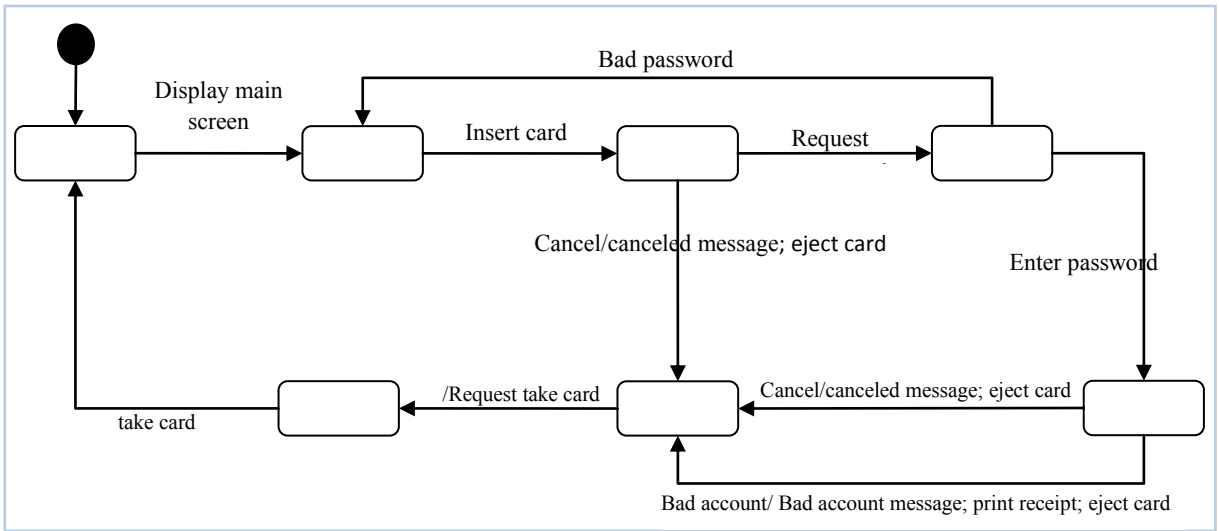


Figure 3.1: the flatten FSM of ATM

Whittle and Schumann described the FSM that represents an ATM machine processes, Figure 3.1 illustrates the generated FSM after merging sequence diagrams, the event *cancel* might be generalised as Whittle and Schumann introduced it [30], a *cancel* event has occurred many times in different states and all these cancel events take place when card inside ATM machine. Partitioning statechart into different super-states to introduce hierarchy depending on whether card inside machine while transaction processing or card inside machine to activate it to be ready for processing.

Figure 3.2 shows the generated statechart corresponding to the FSM of ATM transactions after generalization and introduction of hierarchy. However, introducing hierarchy in this example depends on manual partitioning of an FSM into composite states. Introducing machine learning concepts is a crucial to ensure partitioning done automatically and to make introducing hierarchy more effective and general.

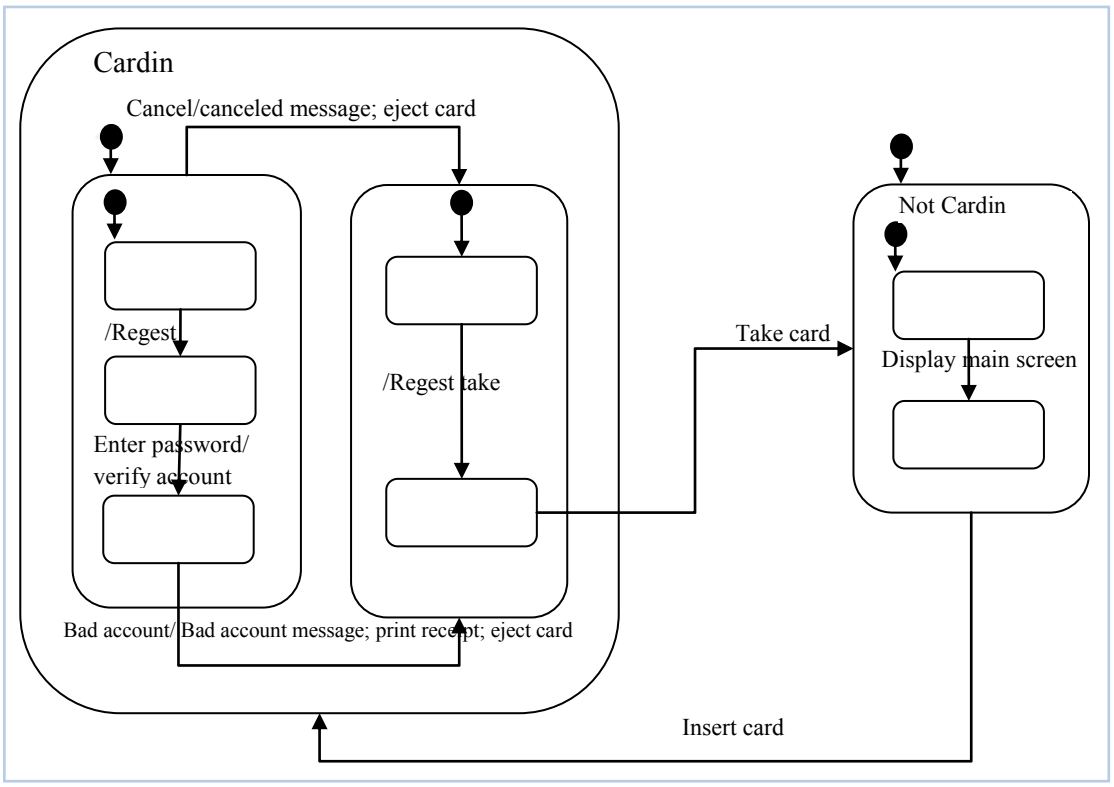


Figure 3.2: statechart of ATM

One of the successful methods used to extract hierarchical statecharts from FSMs is *SCHAEM* which depends on the concept of *decomposition*, shown by Ashish [24], where the *decomposition* approach works on dividing complex FSMs into simpler sub-state machines; these simpler FSMs are introduced to be components for AND-states to introduce concurrency. In addition, *SCHAEM* method introduces hierarchy as well by formation OR-state depending on clustering approach.

Ashish Kumar [24] introduced hierarchy to the flatten FSM first, and then the decomposition method is used to implemented simpler FSMs to satisfy concurrency. Moreover, the statechart extraction method will be applied after finishing the decomposition phase; this method produces an equivalent statecharts from an FSM with consideration for hierarchy and concurrency. Some metrics are used to evaluate these equivalent statecharts compared with

FSMs. Figure 3.3 shows the statechart that produced after decomposing of the FSM shown in Figure 2.2 as Ashish [24] described it;

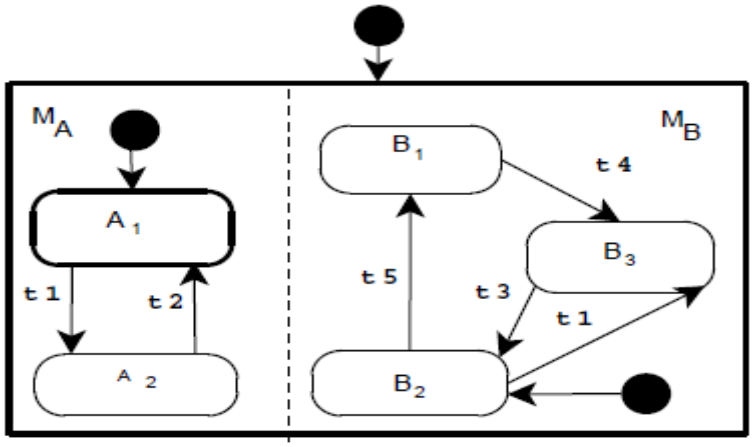


Figure 3.3: The generated statechart using mapping [23]

Figure 2.2 shows an FSM with has six states and fifteen transitions, the decomposition method works as follows:
Firstly, dividing the set of states into two partitioning as constraints suggested by Ashish [23, 24]:

A= {{S1, S2, S3}, {S4, S5, S6}}
B= {{S1, S6}, {S2, S5}, {S3, S4}}

The above sets have been generated according to transitions occurring into small groups. For instance A1= {S1, S2, S3} are sharing the same outgoing transition *t1*, so that when *t1* occurs in any state in A1 system switches to states in A2= {S4, S5, S6}. A1 and A2 will be the first component of AND-state.

The second partitioning has three sets B1= {S1, S6}, B2= {S2, S5}, and B3= {S3, S4}. Transitions (*t1*, *t2*) are used to construct AND component named as $\mathbf{M_A}$; the remaining transitions are (*t3*, *t4*, *t5*) causes the partition B to be divided into three groups of states. For instance, *t5* is the outgoing transition for states {S2, S5} and *t4* is the outgoing transition for states {S1, S6}, and so on [7].

Systa *et al* [31] describe introducing different ways to detect the possible forming of composite state. These ways depend on the statechart diagram matrix as below:

Table 3: The statechart diagram matrix

	H	I	J	K
A	x	a		z
B	x	a		z
C		a	b	z
D		a	b	

Table 3 illustrates that A, B, C, D, H, I, J, and K are states; transitions are shown as x, a, b, and z. Each transition in any cell illustrates that leaving the state in the same row and entering the state in the same column such as x transitions leaving states (A-B) and entering state H. The purpose of using this table is to identify all possible composite states in any flat state machine.

Figure 3.4 shows the generated statechart after forming all these super states:

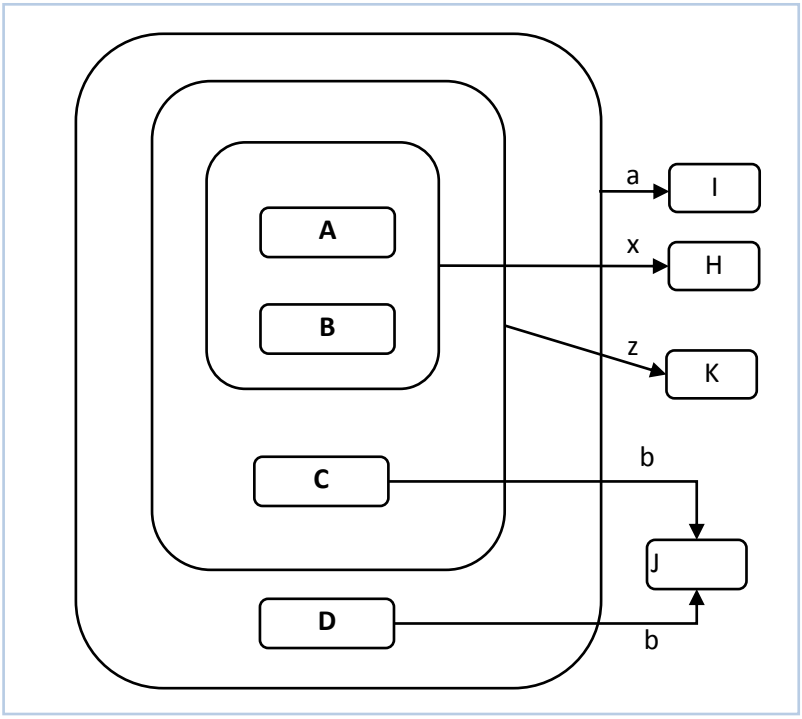


Figure 3.4: statechart using mapping [31]

The following algorithm was introduced in [31] to obtain the previous statechart after forming three composite states:-

Input: *A statechart diagram G.*
Output: *A semantically equivalent statechart diagram with composite states.*
Method:

repeat
Find the greatest (ties are broken arbitrarily) non-contradicting set S of states such that all states in S have a similarly labeled leaving transition entering the same state;
Construct a new superstate containing the states of S;
Replace the similarly labeled leaving transitions by a single transition leaving from a superstate contour;
until *no set S of at least two states can be found;*

Figure 3.5: introducing statechart with composite state [31]

The previous algorithm is aimed to have maximum sub-states that include a composite state without any semantic to choice simple states to be candidate to composite states. This is very helpful to identify maximum composite states that might be generated from FSMs.

Khriss *et al* [22] describe a way of generating a statechart automatically from a collaboration diagram, this done in different processes. These processes are implemented in order to obtain system behaviour specifications. The following list shows the whole activities and what happens in each activity.

1- Requirement acquisition: use case is used to describe scenarios.
2- Each use case is represented as collaboration diagram.
3- Generating statechart diagram from collaboration diagram and this transformation summarises in five step as follows [22]:
    a- Create a statechart diagram for each class represented as the objects in a collaboration diagram.
    b- All variables that are not attributes of the objects of collaboration diagram are introduced as state variables.
    c- For the objects, transitions are created depending on the messages are sent from and to the objects.
    d- Collect all state diagrams and transitions to connect them by states in the correct sequence.
    e- Label all states in each state diagram and validate them.

3.2 Case Study

Studying the possible ways of constructing statecharts from FSMs is an important factor in this project, the main challenges that might occur is defining which states have to merge together to construct composite states. Introducing hierarchy and concurrency is always done by merging simple states into super states (composite states) in order to reduce the number of transitions; however merging transitions should keep system behaviour unchanged. In this studying some facts will be clear about merging states have constraints should be taken in consideration in such transformation. Suppose there is an FSM with four states and five transitions as below:

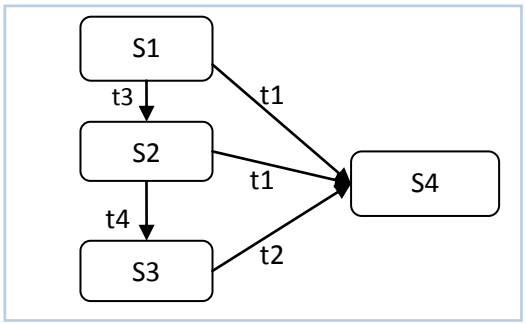


Figure 3.6: simple FSM

Table 4 illustrates state- transition table represents the previous FSM:

Table 4: The State-Transition Table of Original FSM

	S1	S2	S3
T1	S4	S4	-
T2	-	-	S4
T3	S2	-	-
T4	-	S3	-

The first possible merging is depends on grouping simple states to form an OR-composite states in order to similarity transitions. Figure 3.7 illustrates that transitions labeled with *t1* leave states *S1* and *S2* reaching the same target state *S4*, then states *S1* and *S2* have been merged together. The system behavior in the generated statechart is similar to the original FSM and the number of transition is reduced by one.

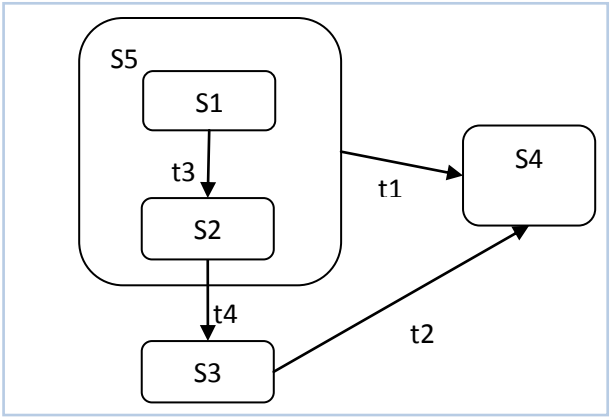


Figure 3.7: OR-state (clustering approach)

Table 5 shows that the previous way of generalizing simple states into composite states:

Table 5: the State-Transition table of the first option of generalization

	S1	S2	S3
T1	S4	S4	-
T2	-	-	S4
T3	S2	-	-
T4	-	S3	-

As we have mentioned before the main advantage of a statechart over an FSM is a small number of transitions in statecharts. To achieve this, a new way of merging simple states to form composite states will be shown below:

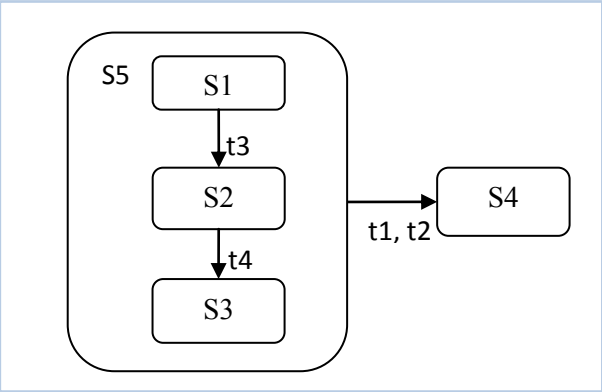


Figure 3.8: second way of generalization

Figure 3.8 shows that states *S1*, *S2*, and *S3* are combined together as composite state *S5*. This way merging assumes when transitions *t1* and *t2* are taken, a system switches from states *S1*, *S2*, *and S3* to state *S4*. The number of transition is decreased by two, however this merging does not successes in maintaining the behaviour of a system as the following state-transition table summarizes (see Table 6).

Table 6: The State- Transition of Second option of Grouping states

	S1	S2	S3
T1	S4 √	S4 √	S4 X
T2	S4 X	S4 X	S4 √
T3	S2 √	-	-
T4	-	S3 √	-

Table 4 summarizes the above ways of generalizing states into composite states and the improvement that is obtained after each generalization.

Table 7: The improvement of merging states option

	No. of transition	No. of State	No. of OR state	Improvement
Fig.3.6	4	4	0	-
Fig.3.7	3	4	1	+1
Fig.3.8	2	4	1	-2

According to the previous study, some points have concluded as follow:
Merging more simple states into a composite state without any concern for similarity of events will reduce the number of transitions as our aiming required. As a result, system behaviour will be changed contrary to what should be preserved it.

As we described above, the clustering has a vial role in reducing the number of transitions to simplified FSMs with hierarchy approach. The reduced number of transitions has a relationship with the number of simple states that were clustered together as an OR-composite state. The number of simple states in each OR-composite state increases depending on the similarity of outgoing transitions to the same target state. To construct this composite state by embedding simple states, some information related to transitions are needed to be known such as source and target states. Finding the commonality among transitions that satisfy constraints is the main factor for the clustering to be effective.

The approach that might be used in this project is introducing AND-states and OR-states as composite states, depending on the transition information (such as the source and target state for each transition). Knowing transitions information might help in defining similarity transitions, reducing the number of events. Suppose there are two states (*A* and *B*) and both of them are linked to state *C* with different transitions labelled with *t1* (see Figure 3.10) and these states are combined together as an OR-composite state: the number of transition is reduced by one as shown in Figure 3.9.

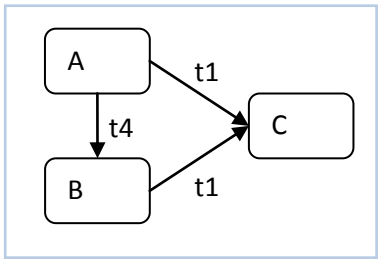


Figure 3.10: FSM before merging states

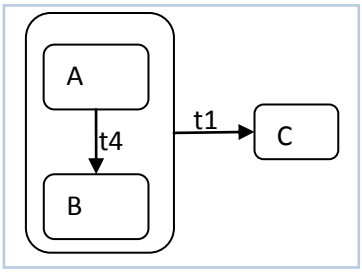


Figure 3.9: OR-state constructing

To show the efficiency of the technique described above, suppose the following FSM (Figure 3.11) with multiple states and transitions:

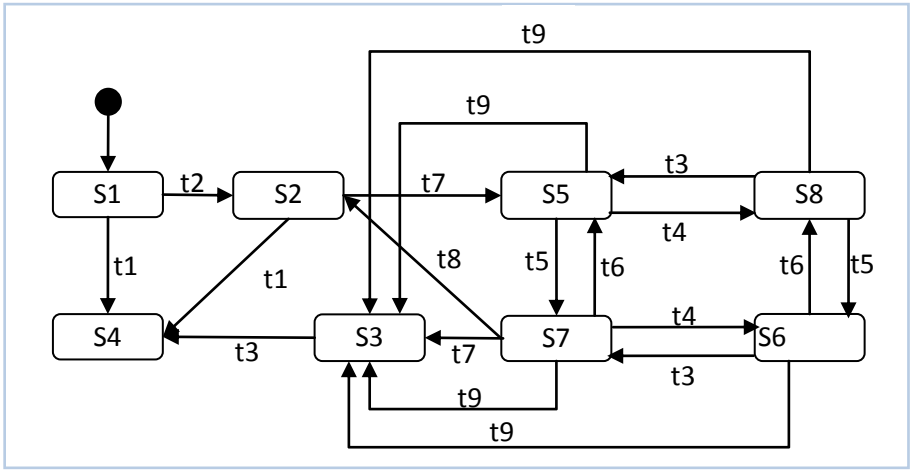


Figure 3.11: complex FSM

In Figure 3.11, the flat state machine has eight states and twenty transitions; some of these transitions have the same target states. These states might be combined together first using OR-states to be the new source state for transitions with the same target state.
Figure 3.12 shows the same model after introducing hierarchy; here the number of transitions is reduced by seven. However, sometimes introducing such composite states is impossible and this technique will not be effective in those cases.

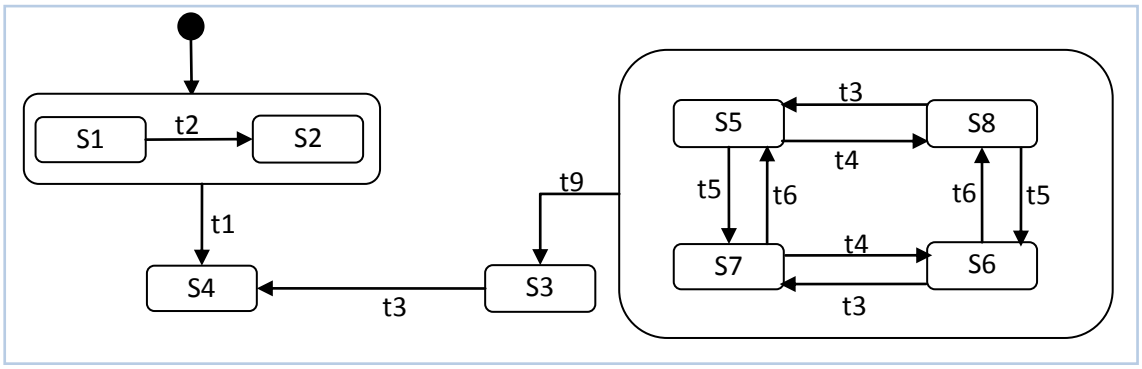


Figure 3.12: statechart with hierarchy

Figure 3.13 illustrates the generated statechart after introducing concurrency.The statechart shows in Figure 3.13 has eight transitions compared to the original FSM which has twenty before introducing hierarchy and concurrency. Figure 3.13 shows how the mapping method is used to introduce AND-components.

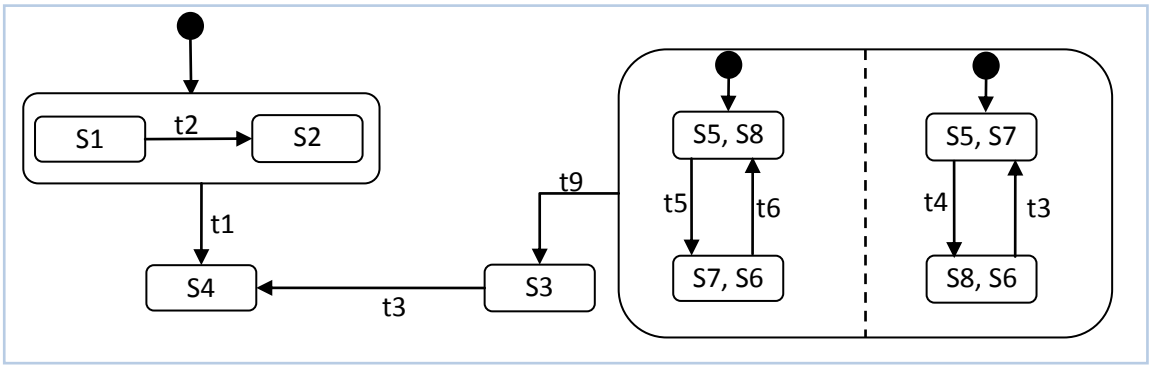


Figure 3.13: a hierarchical statechart with concurrency

Chapter 4: Requirements and Analysis

4.1 Requirements

The previous section showed related work and different techniques in handling FSM to obtain a hierarchical statechart by introducing hierarchy and concurrency. Requirements are represented in different phases as construction steps, with consideration to the objectives discussed above:

1. Import state machine in XMI format, as ArgoUML has the ability to export this format for any FSM.
2. Work through the imported FSM to convert it to a statechart, which is required in order to identify states and transitions from the XMI state machine, using JDOM to access states' information (such as ID and name). Transition information should be defined as well, such as the source and target states for each transition.
3. Produce a DOT file describing the information gathered in the previous stage to view the loaded FSM as graphics.
4. Implement an OR-composite state that represents the hierarchy using clustering approach, after completing finding possible OR- composite states.
5. Producing another DOT file to show how clustering approach has been implemented.
6. Implement an AND-composite state that represents the concept of concurrency.
7. Produce the final statechart that corresponds to the flat state machine, keeping the system's behaviour.

4.2 Analysis

The crucial part of this project is to analyse how each step of constructing statechart from FSM will be done. Additionally, some issues related to previous methods have been taken in consideration to avoid them in this project. Deciding which methods are useful to generate OR, AND composite states.

Firstly, exporting FSM from ArgoUML tool as XMI format will be done manually; in our research, FSMs with different levels of complexity will be drawn and exported from ArgoUML to use them to see whether our developed system able to construct statechart from them.

Secondly, loading XMI files which represent FSM to the system requires identifying states and transitions accurately; also ensure the loaded FSM is valid regarding to reachability approach. Knowing how XMI represent metadata in different levels to gather information about states and transition easily.

Thirdly, producing DOT file automatically might be done by looking to the syntax of this language and how hierarchy of states can be described. GraphViz has been selected to be a tool of converting an FSM to a picture both to visualize the outcome of conversion and to defining it.

Fourthly, the clustering approach described in section 2.3 above is the effective method of constructing OR-composite; hence this approach will be used regarding to efficiency and correctness of grouping similar transitions together. Finally, the decomposition approach described in sections 3.1 and 3.2 is very effective in obtaining possible AND-state constructing; this method might be used in this project to introduce concurrency correctly.

Chapter 5: Design

In this chapter, the design of the proposed system will be described in details. The following diagram illustrates design structure of the developed converter:

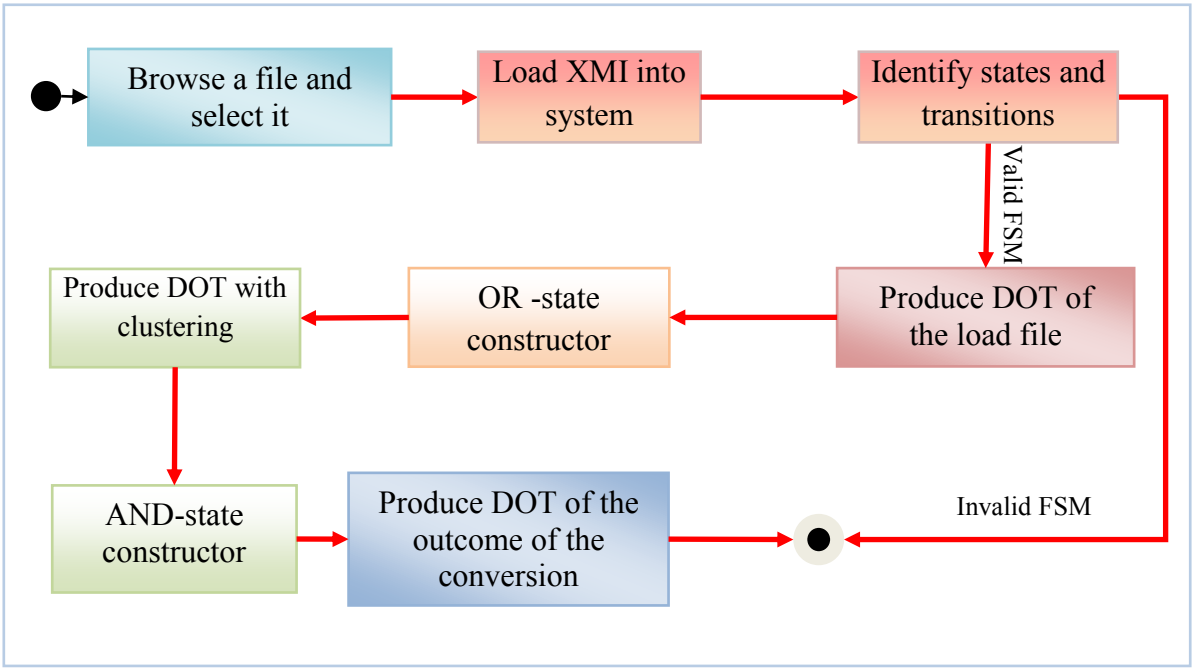


Figure 5.1: flowchart of converter

The following list summarizes the whole process as shown in Figure 5.1:

- ***Browse file and select it:*** Browse an XMI file that is exported using ArgoUML; the selected file is used to be the program input.
- ***Loading FSM into the conversion system:*** in this phase the system will identify states and transitions by gathering information related to states and transitions by accessing XMI element using JDOM. The system checks whether the loaded FSM is valid or not; the proposed system will reject any invalid FSM in order to reachability and correctness considerations.
- ***Produce a DOT file***: this will be done for each valid FSM in this phase; this allows us to view the load FSM graphically (for debugging).
- ***Constructing OR-states:*** in this step, all possible OR-states in the loaded FSM will be detected and the possible recursive clustering determined; after that constructing them will be in this phase.
- ***Producing a DOT file representing the outcome of clustering:*** After that, producing another DOT file to reflect recursive clustering.
- ***Constructing AND-states:*** after constructing OR-states, this phase concerns with trying to construct AND-state to the generated hierarchical FSM.
- ***Producing a DOT file representing concurrency:*** *in this step* producing a DOT file to view concurrency.

5.1 The overall structure of XMI file

Loading an XMI file into the conversion using JDOM is required knowing the structure of FSMs as XMI format, where the purpose of this is to help us in defining all states and transitions. The following diagram shows the hierarchy of each FSM as XMI.

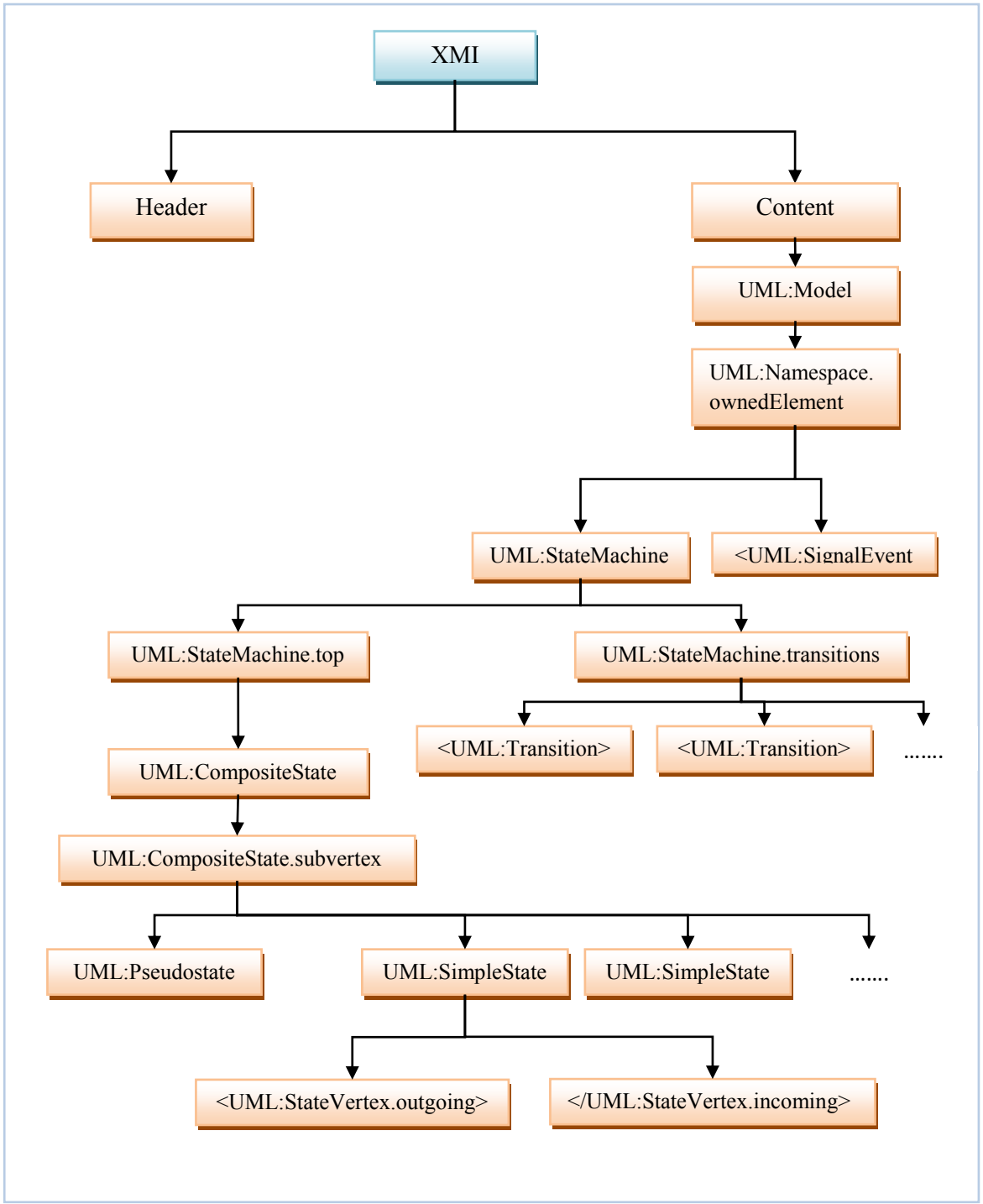


Figure 5.2: the structure of FSM as XMI format

5.2 Loading States

Figure 5.3 shows an XML schema (XSD) diagram of each simple state element in a XMI file where attributes and nested elements that are related to a simple state is. The loaded FSM consists of many simple states; the 'xmi.id' attribute is a unique value that is used to identify them, 'name' is another attribute in simple state element which is needed to be a second property that represents each state. Others attribute such as 'xmi.idref' and 'isSpecifiction' are not necessary in our designing. There are two inner XMI elements that represent outgoing and incoming transitions to each state such this information useful to find possible OR-states.

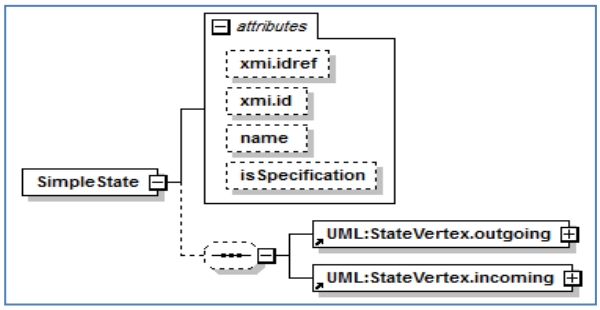


Figure 5.3: XSD diagram states attributes and elements

5.3 Loading Transitions

Loading transitions in the proposed system requires studying the transition element in an XMI file to extract useful information. Transition element has similar attributes to a simple state element such as xmi.id which is also unique to represent each transition (see Figure 5.4). The inner elements of transitions element are an important in gathering related information such as source, target, and event label of each transition. Knowing the source and target of each transition necessary for identification of OR-states, where each of them is a simple state as the OR-state construction is described later in this chapter.

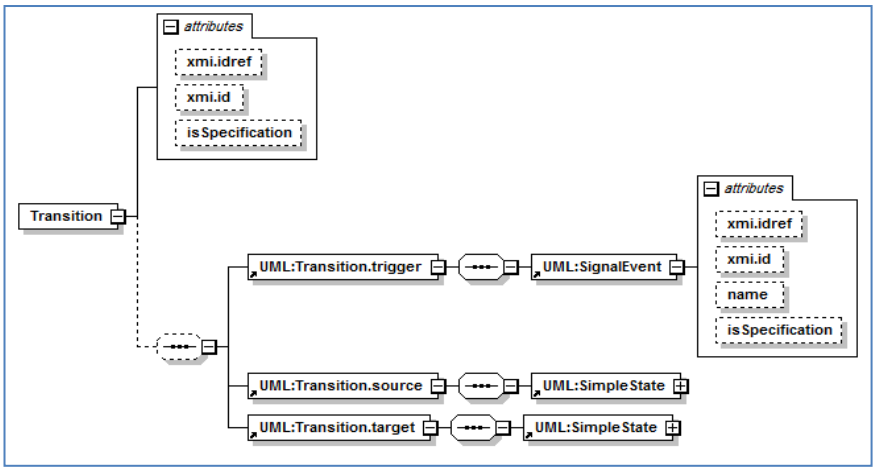


Figure 5.4: Transition attributes and elements represented using XSD diagram

5.4 Design choices

5.4.1 Browse and select file

It has been agreed that designing browsing and selecting an XMI file will be done using a File chooser to select only XMI file format.

5.4.2 Load XMI file

After choosing an XMI file, information related to states and transitions is required to extract them using JDOM. In addition checking validity of the loaded file has designed to be done in this phase. Figure 5.5 shows design selected to extract states and transitions.

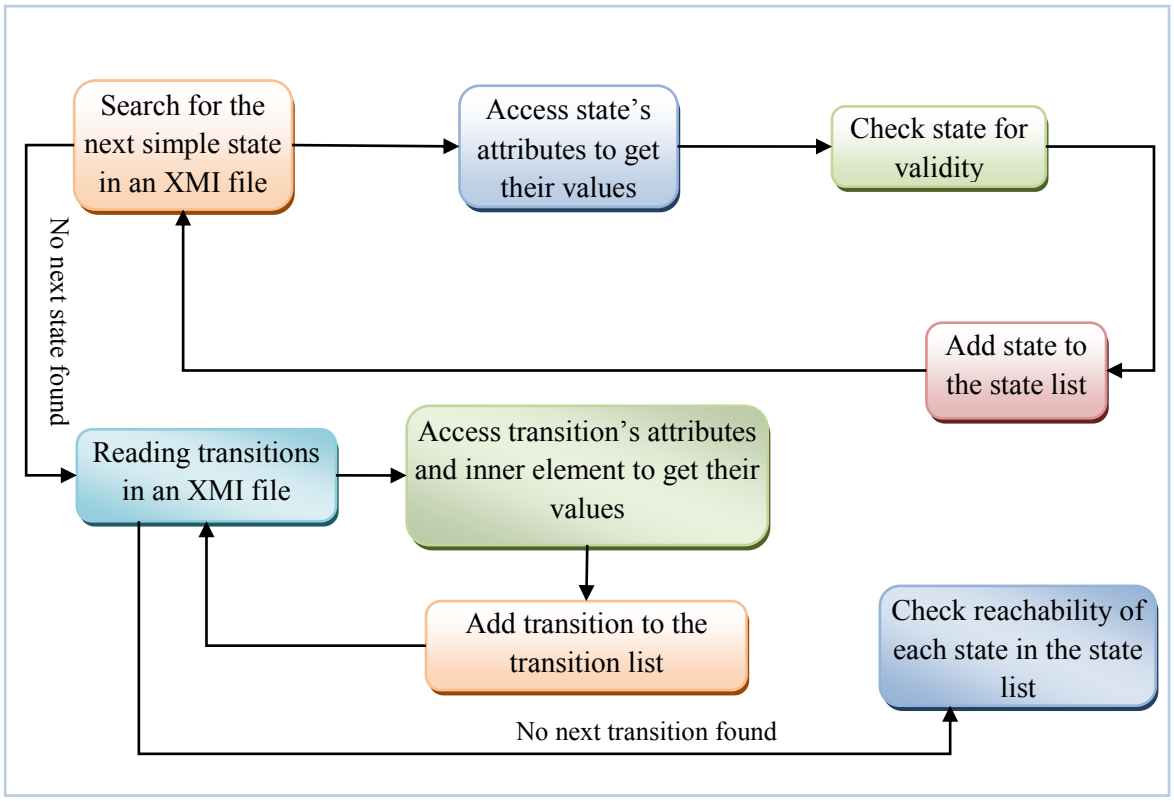


Figure 5.5: state machine for extracting states and transitions

Figure 5.5 summarises the whole process of extract states and transitions from a loaded FSM, where JDOM play a vital role to extract them. The load FSM in XMI format should be valid, checking reachability of each state is required to have all information about states and transition; for this reason, in our design the checking of reachability is done after finishing reading all states and transitions. Checking of correctness includes that each state should not have more than one outgoing transition with the same label and each state must has a name and a unique ID.

5.4.3 Produce DOT file for the loaded FSM

After a successful load of an FSM, the proposed system will be able to produce a dot file where states and transitions are described as below:

```
digraph finite_state_machine
{
  node [shape = circle];
      S4 -> S2[ label = t5 ];
      S3 -> S4[ label = t7 ];
      S1 -> S4[ label = t7 ];
      S1 -> S3[ label = t3 ];
      S2 -> S5[ label = t7 ];
      S5 -> S4[ label = t6 ];
      initial -> S2[ label = t2 ];
      initial -> S1[ label = t1 ];
      S5 -> S2[ label = t9 ];
}
```

Figure 5.6: DOT file of FSM after loading

Generating DOT file is designed depends on transitions information, where the required information is acquired from transition objects. Each transition objects has a source state, a destination state, and label. Source and destination states are represented as a circle in the DOT notation. To ensure that the generated DOT file is valid in syntax, Graphviz tool able to read a valid dot file and produce a graph represents it. Also, Graphviz has ability to show warnings if there is any error in syntax related to the entered DOT file. Figure 5.7 illustrates how Graphviz reads a DOT file and views graphically [10, 11].

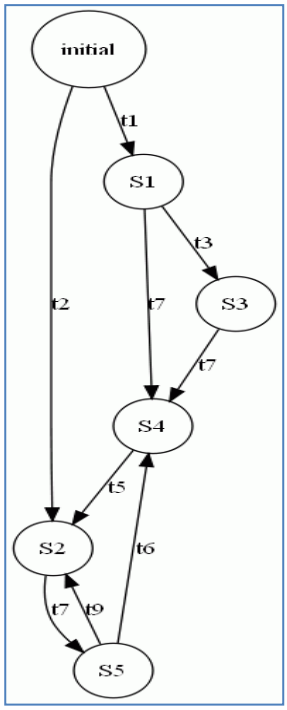


Figure 5.7: visualising DOT file using GraphViz

5.4.4 OR-composite state constructor

There are many techniques for forming OR-composite states after loading a FSM; some of them are complex and other simple. In this section, two vital techniques will be explained below:

5.4.4.1 First Technique: Transition dependency

In this technique, transitions are the starting point to find possible OR-state as follows:

- **Make a set (T) for all transition events**
  - **For each transition event t_i, in T**
    - *Count number of transitions (n) with label t_j in FSM*
    - **If n > 1, there are some transitions with same label**
      - ☒ *Create list T_j*
      - ☒ *add all transitions in a FSM with same label t_j reaching target states into T_j*
      - ☒ *Src_of_transitions_to_the_same_target state (T_j)*
      - ☒ *If number of source > 1,*
        - * *Create OR instance*
        - * *Add all returned sources as component of OR-state*
    - **Else, continues with t_{j+1}**

Src_of_transitions_to_the_same_target state (T), begin

- For a target state s, let a set of maximum sources

  $Src(s,T)=\{source(t) \mid t_i \in T \wedge target(t)=s\}$

- Return list of sources

To show how the previous algorithm works, consider the FSM shown in Figure 5.8:

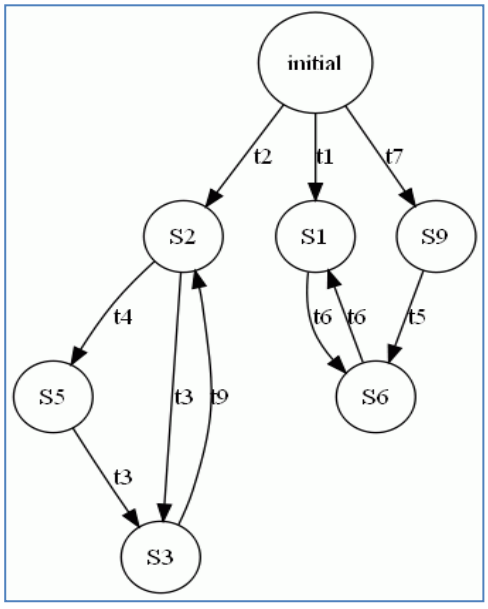


Figure 5.8: an example of FSM

There are seven states and ten transitions. The method of generating OR-composite states starts by gathering all events found in the loaded FSM, and keeping them in special list. The following set is made for all transitions events occurred in the loaded FSM:

$$T' = \{t1, t2, t3, t4, t5, t6, t7, t9\}$$

After collecting all events, each event in the previous set will be counted regarding to the whole Transition (*T'*) in a FSM as below (see Table 8):

$$T = \{t1, t2, t3, t3, t4, t5, t6, t6, t7, t8\}$$

Table 8: the occurrence number of each label in the whole transitions

Event in T'	*Number of occurring in T*
t1	1
t2	1
t3	2
t4	1
t5	1
t6	2
t7	1
t9	1

The following set of transitions do not satisfy the constraint of occurring more than once, hence the method will ignore all these events:

$$T' = \{t1, t2, t4, t5, t7, t8\}$$

The method will create two lists for t3 and t6; these lists are used to store all transition instances related to each as below:

$$T_{event\_t3} = \{S2 \xrightarrow{t3} S3,\ S5 \xrightarrow{t3} S3\}$$

$$T_{event\_t6} = \{S1 \xrightarrow{t6} S6,\ S6 \xrightarrow{t6} S1\}$$

Each instance in both lists is different such that the first element in Tevent_t3 has id, source state, and target state that is different to the second element in the same list. Each list is passed to get_sources method to see whether there is any possibility to construct an OR-composite state. Get_sources() method received a list of instances' transitions such as the T_{event_t3} list; both elements in the T_{event_t3} list has the same target set, then all sources of those instances will be components of OR-state. The second list T_{event_t6} has different target states, hence no possibility to form an OR-state.

This technique depends on finding maximum components of OR-state and similar target states for those components.

5.4.4.2 Second technique: States dependencies

In this technique, states are the root to find possible OR-state as follows:

- **Collect all simple states (S) found in FSM**
  - **For each state s_i, in S**
    - *Collect all incoming transitions reaching to s_i into IT Set*
    - **If IT size > 1, which mean there are many transitions are coming to s_i**
      - ☒ *Get_sources_state(*IT*)*
      - ☒ *If number of source > 1,*
        - * *Create OR instance*
        - * *Add all returned sources as component of OR-state*
    - **Else, continues with s_{j+1}**

Get_sources_state(IT), begin
- For each t_i in IT
  - * Add t_i to source list
  - * If the label of t_i same as t_{j+1}
    - Add t_{j+1} to source list
- Return list of sources

The second technique works as effectively as the first technique in finding all possible OR-composite states. Here are the set of states shown in Figure 5.8:

$$S = \{initial, S1, S2, S3, S5, S6, S9\}$$

After collecting all states inside FSM, 'initial' state will be selected first from the set of states to find all transitions coming to the 'initial state', the size of IT set will be zero; so, the algorithm selects another state. Suppose S3 is selected to find all incoming transitions, the set of incoming transitions will be shown as below:

$$T_{event\_t3} = \{S2 \xrightarrow{t3} S3,\ S5 \xrightarrow{t3} S3\}$$

The resultant set consists of two incoming transitions, this list will be the basis to find OR state possibility, which is similar to the first technique.

5.4.4.3 Finding recursive clustering and overlapping

The important step in designing the conversion is to find possible recursive clustering and overlapping. Suppose we have more than one possible OR-state that might be generated in any FSM. The following FSM demonstrates how recursive clustering and overlapping is detected:

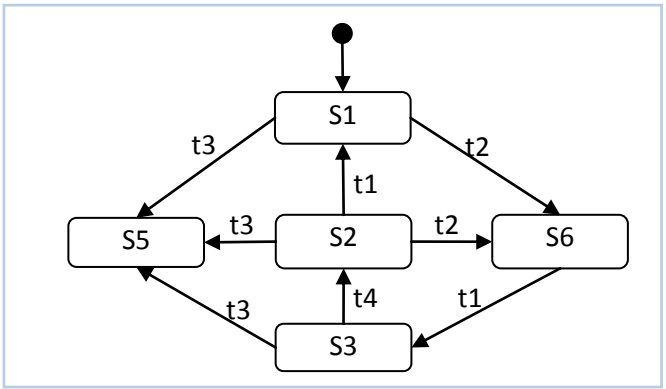


Figure 5.9: FSM

The first OR-state has three components as below:

OR-State1= {S1, S2, S3}

The second OR-state is as follows:

OR-State2= {S1, S2}

The components of OR-state2 occur in the first OR-state1, and then the second set of states will be nested in the first set as below:

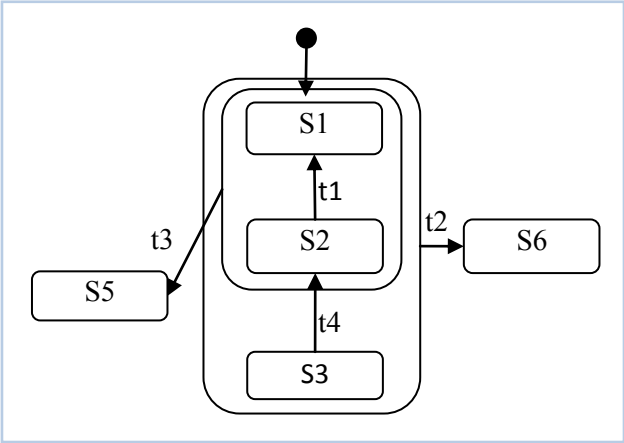


Figure 5.10: nested clustering

Suppose the second OR-state2 has been changed to the following:

OR-State2= {S2, S5}

State S2 will be overlapped. According to the previous examples and the following theory, this shows how the recursive clustering and overlapping are discovered.

- Collect all possible OR-states that occurred in the Clustering set (CS)
- Calculate the size for each OR-state in the CS
- For each OR-state
  - Select the largest size for or-state or_0
    - ✓ Construct or_0
  - For each remaining OR-state in the CS
    - Select or_i state form the CS
    - Compare or_i component with the component or_0
    - If all components of or_i also occurs in the or_0, which means containment:
      - ➢ or_i is nested in or_0
    - if intersection but or_i is not contained in or_0, then
      - ➢ or_i is overlapped with or_0
    - else this or_i is not part of or_0
    - add or_0 to the processed OR-state list (or_{done}) and delete or_0 from CS

5.4.5 Data structure

There is different data structure that might be used to store information gathered from a XMI file. The challenge is to select an effective data structure among multiple data structures in java language. The selected data structure should be in format <key, value> data structure; there are multi data types that support the concept of key- value pair such as HashMap, TreeMap, and HashTable.

TreeMap will be used as data structure in our system. It allows storing data in order keys to allow us to iterate through them easily; rapid retrieval of data, especially when FSM become larger, is also possible.

5.4.6 Program design

5.4.6.1 Activity diagram

The following activity diagram describes the whole processing of converting FSM into state chart with preserving the behaviour of system.

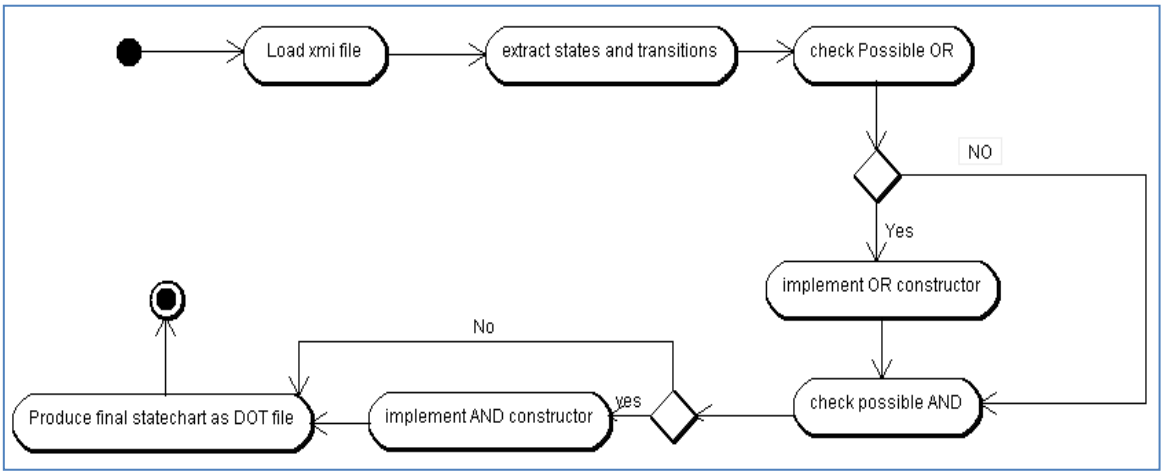


Figure 5.11: activity diagram of the general processes in the conversion system

Class diagram

Figure 5.12 illustrates the overall structure of classes in the developed system, where the important class will be described below. Other classes will be explained in the appendix section:

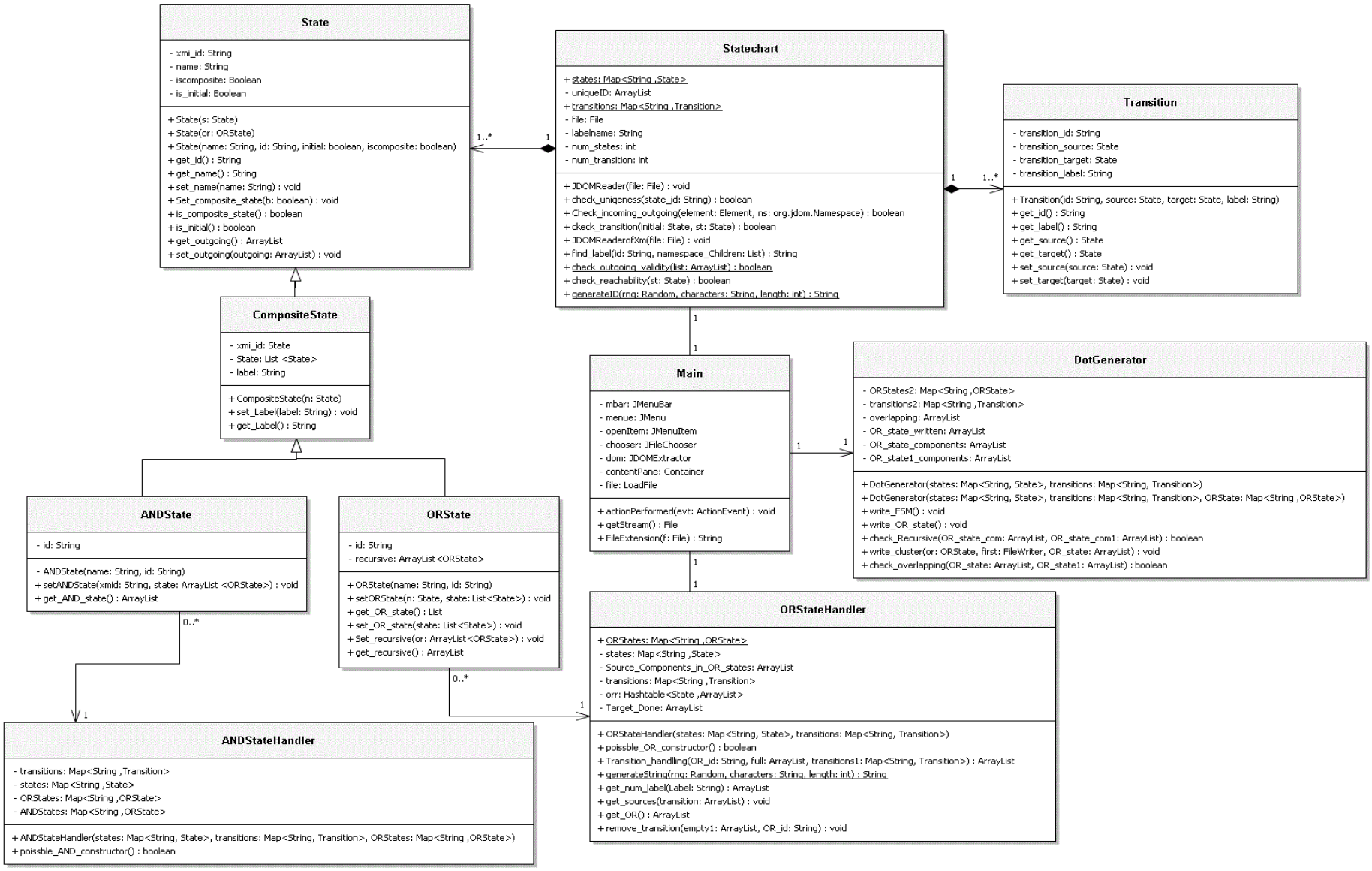


Figure 5.12: Class diagram of the conversion system

5.4.6.1.1 Main Class

Main class is the crucial class in this system in order to create the interface that allows users to browse to select an XMI file and convert it to a statechart. An instance of this class manages all other functionality of the converter such as generating DOT files are invoked from it. There is a method called *actionPerformed()* to cope with a series of events such as browsing the file and open stream to read the content; also loading the file using JDOM to identify all states and transitions. Producing a DOT file for the load FSM is also performed in the *actionPerformed()* method. In this class, designing the interface is done by defining a menu that allows the user to carry out the conversion easily. *FileExentsion (File f)* method is called to identify the extension of selected file, because of there are two format handlers for each kind FSM format; first XMI handler to cope with XMI file to identify states and transitions, second _X format handler which is different from XMI format to extract states and transitions from it.

5.4.6.1.2 Satechart class

Satechart class is the crucial class in our design that is responsible for loading FSM in XMI format after reading it. The main method in this class is *JDOMReader(File file)* which is used to access all XMI elements to gather information about states and transition using JDOM. *JDOMReaderxm (File file)* method is called to read FSM in _X format, this format represented as (S1 S2 t1) which means 'S1' is a source state, 'S2' is a target state, and 't1' is transition's label. *Check_Reachability()* is a method that is used to check whether the current state is reachable or not to achieve our aim of loading valid FSM only. Another important method *check_outgoing_validity()* is used to check outgoing transitions for each state to avoid multiple transitions leaving the state with the same label. *Check_uniqness ()* method for validating that transitions and states Ids are uniqueness. Each state in an FSM must have incoming and outgoing transitions, *Check_incoming_outgoing()* method is called to check this validation. *Check_transitions*() method is used to ensure information about source and target sources related to each transition is provided and not empty value. Loading FSMs in _X format does not provide IDs for states and transitions, *generate_ID()* function is used to provide a unique ID for each states and transition loaded from an FSM.

5.4.6.1.3 ORStateHandler class

ORStateHandler class is the class responsible for finding possible OR-states in the loaded FSM and constructing them. The *possible_OR_constructor()* method determines whether the loaded FSM has any possibility to have an OR-state constructed depending on the clustering approach introduced in Chapter 5. In addition, *get_sources(transition)* method is an important function in collecting all sources' states related to each transition before sending the returned list *Transition_handling()* method to handle them to decide which transitions are required to remove them. Finally, *remove_transition()* function is called to remove a unnecessary transitions after clustering states depends on the list returned by the *transition_handling()* method.

5.4.6.1.4 DOTGenerator class

DOTGenerator class is created to generate DOT files after different stages of the conversion processes. This class is initialised by states, transitions, and OR-states. *write_FSM()* method is used to write a DOT file for the load FSM. *write_OR_state()* is a crucial method to construct clustering and nested clustering. The nesting clustering is designed to be more flexible without any constraints to the number of nesting. *Check_recursive(ArrayList list, ArrayList list2)*

method is used to check to find possible nested states between two OR_states components, where the first parameter is larger than the second parameter. Each OR-state has an attributes named ‘recursive’ to store all nested state to each OR-states, doing this depends on ordering OR-states in order to their components size. *Check_overlapping(ArrayList list, ArrayList list2)* method is invoked to detect any overlapping between states.

Chapter 6: Implementation and Testing

This chapter explains how the design shown in chapter 5 is implemented. In addition, testing documentation to show how the developed system copes with different kinds of a FSM.

6.1 Implementation:-

The implementation process has been divided into several tasks; the purpose of doing this is to make the development process more systematic. Tasks are shown below:

6.1.1 Implementation the user interface of the application

The application interface has three main menus. 'File' menu has two options, first the 'open' option to open an XMI file is shown in Figure 6.1 and to print the content of it, second the 'Load XMI file' option to load an XMI file using JDOM, then the system prints the names of the loaded states and transitions with presenting the total number of states and transitions after completing loading them. Figure 6.3 illustrates the 'Convert' menu has two options, the first option is to generate DOT file of the loaded FSM; the second option is to convert the loaded FSM into a statechart and generate a DOT file representing composite states.

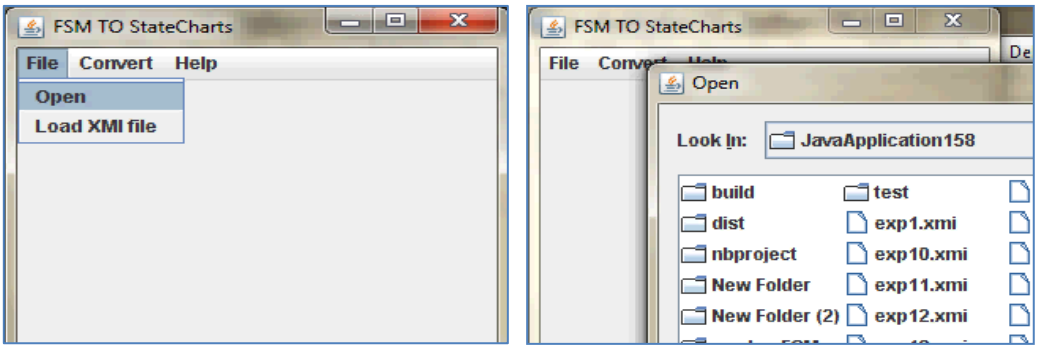


Figure 6.1: File menue and open option

6.1.2 Loading FSM into the system

The second implementation task is to load FSM into the system using JDOM described in chapter 5. Figure 6.2 shows the interface for loading a file, also the output in Figure 6.2 illustrates states and transitions information that are loaded. After loading them, the total number of states and transitions that are loaded are shown in the output to demonstrate that all FSM components were loaded successfully. The program checks the validity of loaded FSM and ensures the loaded FSM achieves reachability while printing the reason of rejecting an invalid FSM. In addition, checking the uniqueness of the state and transition IDs is performed while loading states and transitions.

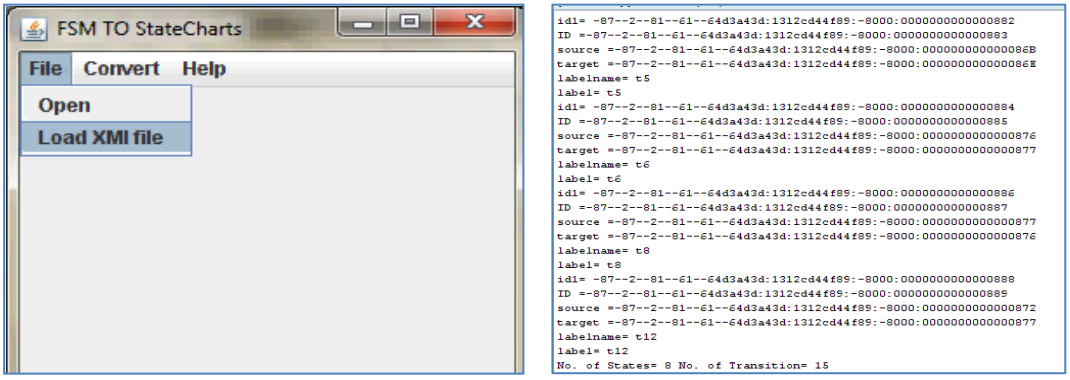


Figure 6.2: loading XMI file option and the output during loading it

6.1.3 Produce a DOT file representing the loaded FSM

Writing a DOT file is implemented by creating an instance of FileWriter in the Java language. For writing a DOT file for a FSM a "*node*" shape is selected to be a shape of each state in an FSM.

```
FileWriter FSMwriter = new FileWriter(new File("FSM.dot"));
FSMwriter.write("digraph finite_state_machine {");
FSMwriter.write("node [shape = circle];");
```

Writing each transition with source states, target states, and its label can be done as follows:

```
FSMwriter.write(source.get_name()+" -> "+target.get_name()+"[ label = "+transitions.get_label()+" ];");
```

Figure 6.4 shows how the DOT file content is written after loading an FSM, this is done by clicking on 'produce DOT file' item from the '*Convert*' menu to illustrate the loaded FSM graphically. Figure 6.5 show the FSM graphically after entering a DOT file to GraphViz tool.

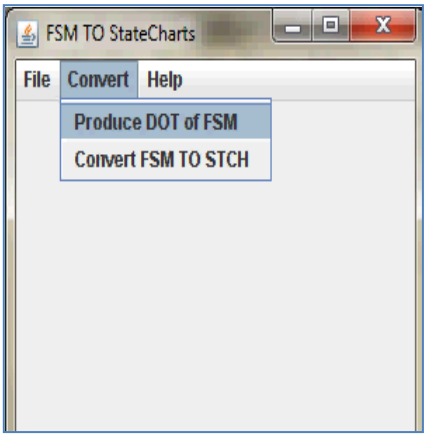


Figure 6.3: produce a DOT file option

```
digraph finite_state_machine {
node [shape = circle];
S0 -> S1[ label = t3 ];
S2 -> S4[ label = t1 ];
S1 -> S4[ label = t1 ];
S2 -> S3[ label = t2 ];
S6 -> S7[ label = t4 ];
S5 -> S7[ label = t4 ];
S7 -> S4[ label = t1 ];
S6 -> S4[ label = t1 ];
S5 -> S4[ label = t1 ];
S4 -> S6[ label = t12 ];
S6 -> S5[ label = t8 ];
S1 -> S3[ label = t2 ];
S5 -> S6[ label = t6 ];
S1 -> S2[ label = t5 ];
S3 -> S4[ label = t1 ];
}
```

Figure 6.4: DOT content of each FSM

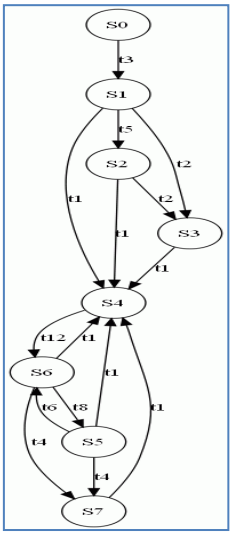


Figure 6.5: GraphViz output of each DOT file

6.1.4 Extracting statechart from a FSM and producing a DOT file to illustrate the generated statechart

The option of convert an FSM into a statechart includes two steps. First, find possible composite states and construct them. Second, generate another DOT file to visualise how clustering and recursive clustering.

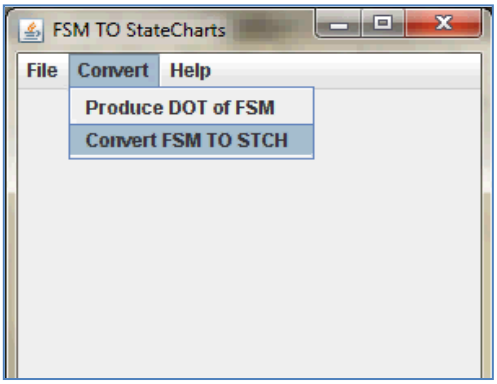


Figure 6.6: convert option

```
No. of States= 11 No. of Transition= 8
No. of OR States= 3
```

Figure 6.7: the actual output after generating a statechart

Figure 6.8 shows how the DOT language copes with each composite state in written in the DOT language as a cluster, also how the recursive clustering is implemented by checking the opportunity to add recursive cluster among composite states (composite state nested inside other composite state). Figure 6.9 illustrates an example of how GraphViz tool view a recursive cluster inside other cluster

```
digraph OR_FSM {
 compound=true;node [shape = Box];
 subgraph clusterrzbca{
 subgraph clusteraxssg{
 S6-> S5[ label = t8 ];
 S5-> S6[ label = t6 ];
 S6;
 S5;}S1-> S2[ label = t5 ];S7;S6;S5;S2;S3;S1;}
 subgraph clusterbwdvu{
 S2;
 S1;}S0-> S1[ label = t3 ];
 S6 -> S7[label = t4 ][ltail= clusteraxssg];
 S4-> S6[ label = t12 ];
 S2 -> S4[label = t1 ][ltail= clusterrzbca];
 S2 -> S3[label = t2 ][ltail= clusterbwdvu];
}
```

Figure 6.8: OR-state representing as clusters and a nested clustering

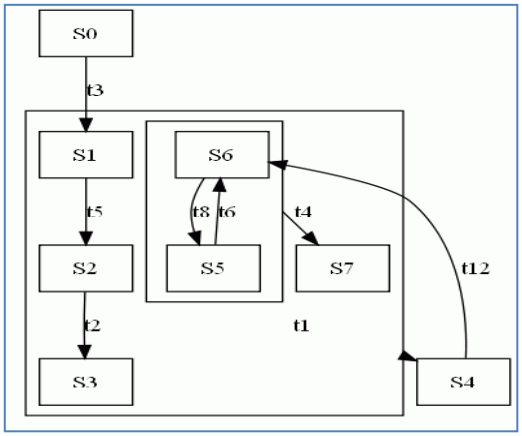


Figure 6.9: GraphViz tool output for a nested clustering

6.2 Testing

Testing is an important phase in life cycle of developing the conversion system. Testing is carried out during coding to ensure new features added to the system work properly and do not conflict with existing features [26].

6.2.1 Functional Testing

The Category-Partition method described by Ostrand and Balcer [27] is used to generate possible test cases to test functionality to detect any errors in the conversion system. For constructing composite states, there are multiple parameters with different categories that can be derived using the category-partition method. The following list summarizes the input and output parameters that are derived from the conversion tool.

- ***Files that are loaded***
  The validity of FSMs is tested against different rules that must be satisfies in each valid FSM such as states reachability.
- ***Transitions entering target states with same label.***
  Each state in a FSM is a source for outgoing and a target for incoming transitions. This parameter concern with incoming transitions entering target states with same label.
- ***OR-states generated using the conversion system.***
  The OR-states that are generated using the system depend on the number of target states with at least two incoming transitions with same label.
- ***Pattern of clustering.***
  A pattern of clustering is described as the final form that will be shown by DOT language. The following list summarizes the possible pattern that might be obtained:

For S such that $S \xrightarrow{l} S_i$, we would like to form a cluster

$C(l, S_i) = \{S \mid S \xrightarrow{l} S_i\}$

- Clusters $C(l_a,S_i)$,$C(l_b,S_j)$ are **independent** if $C(l_a,S_i) \cap C(l_b,S_j) = \varnothing$
- They are **nested** if $C(l_a,S_i) \subseteq C(l_b,S_j)$
- They **intersect (overlap)** if $C(l_a,S_i) \cap C(l_b,S_j) \neq \varnothing$

For the first parameter described above, Files might have an error such the file is empty, and the format of the loaded FSM file is not supported by the conversation system; from above we can get the key parameter which is validity (size, reachability, Compatibility of formats and correctness).

For the second parameter, the number of transitions that are entering target states is an important category. The number of OR states that should be obtained from a converter is the only category for the third parameter. For pattern of clustering, there are multiple categories related to this parameter which is independent clusters, nested clusters, and overlapping clusters.

Partitioning each category mentioned above into partitions as below:

- The validity of loaded file
  - Valid
  - Not valid, this includes two possible reasons are taken in consideration as below:
    1- XMI might be wrong.
    2- XMI is correct but some states are unreachable.

- The number of transitions with same label entering states (NTES):
  - Zero
  - One
  - Two
  - Many

- The number of possible OR-states might be gained using the converter (PORC):
  - None if [NTES < two]
  - One if [NTES > one]
  - Two if [NTES > one]
  - Many if [NTES > many]

- The pattern of clustering (POC) :
  - None if [PORC = none]
  - Independent if [PORC > one]
  - Nesting if [PORC > one]
  - Nested and independent if [PORC > two]
  - Overlapping
  - Independent and overlapping if [PORC > two]

The following table summarizes the possible test cases according to test specification described above with constraints related to conditions.

Table 9: possible test cases

Validity of the loaded FSM	Number of transitions with same label entering states	The number of the possible OR-states	Pattern of clustering
Not valid	-	-	-
valid	**Zero**	None	None
valid	**One**	None	None
valid	**Two**	one	Independent
valid	**Many**	one	Independent
valid	**Many**	Two	Independent
valid	**Many**	Two	Overlapping
valid	**Many**	Two	Nested
valid	**Many**	Many	Independent and nested
valid	**Many**	Many	Independent
valid	**Many**	Many	Overlapped
valid	**Many**	Many	Nested
valid	**Many**	Many	Independent and overlapping
valid	**Many**	Many	Nested and overlapping

6.2.2 Possible test cases

The following test cases are selected according to the Category-Partition method to test functionality:

6.2.2.1 First Test Case

This test case shows that the loaded FSM might be not valid, due to the reachability constraint or unacceptable format. The FSM in Figure 6.10 illustrates that state 'S2' is not reachable from its initial state. The converter will reject such FSMs with explaining the reason of refusing.

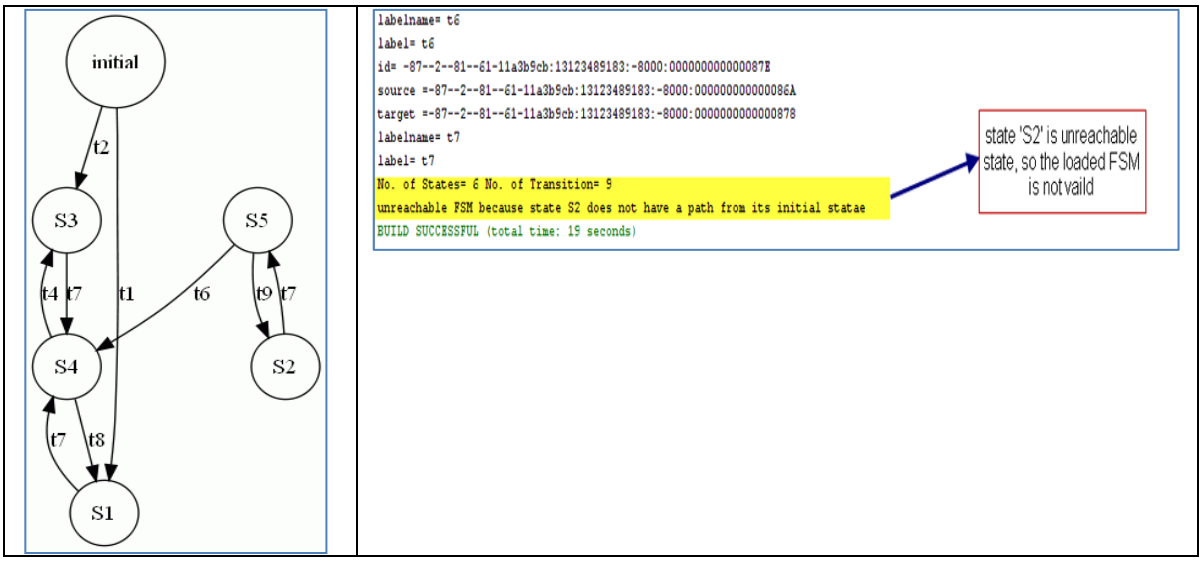


Figure 6.10: an invalid FSM and the actual output by the conversion system

Figure 6.11 shows another invalid FSM where states (S2, S5) are unreachable form its initial states, the developed system terminate loading an invalid FSM. State 'S5' is not printed in the actual output because the system is terminated when a first unreachable state is detected.

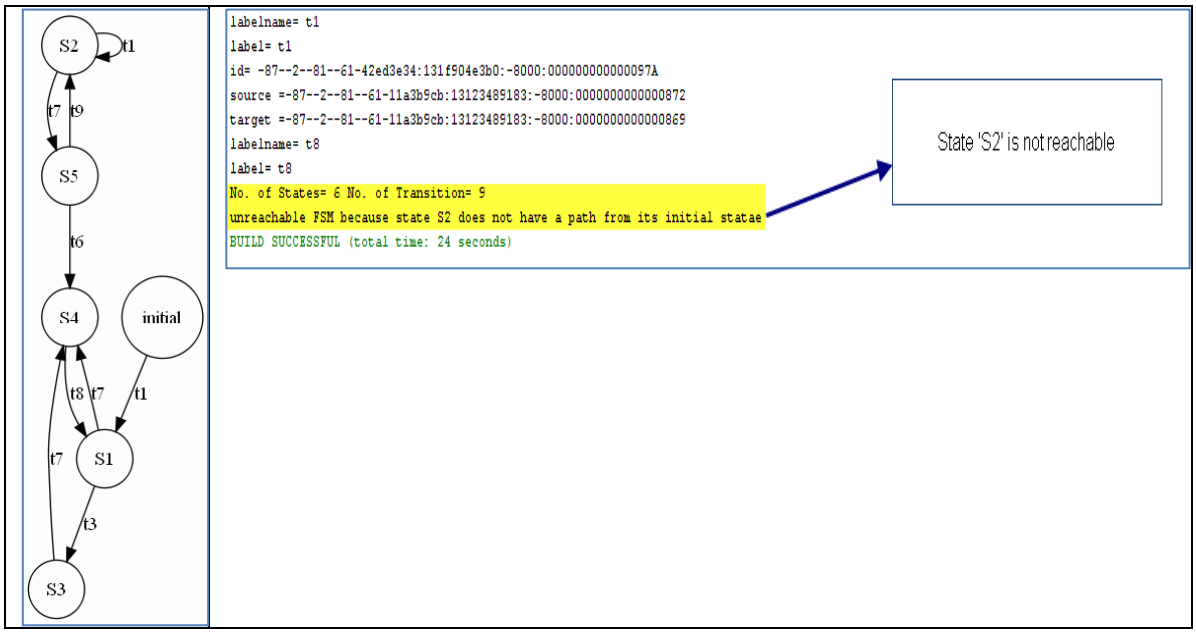


Figure 6.11: an invalid FSM for unreachable states reason

Figure 6.12 illustrates another example of an invalid FSM where state 'S2' does not have an incoming transition and state 'S5' does not have an outgoing transition. This testing expresses how the developed system detects an invalid FSM depending on different rules must be found in FSMs such as each state must has an incoming and outgoing transitions.

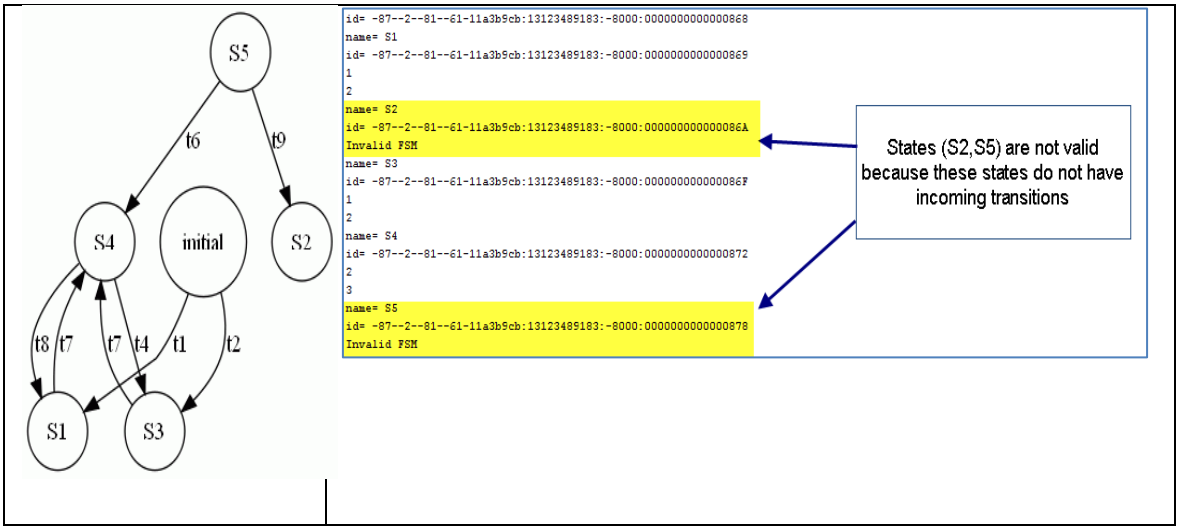


Figure 6.12: an invalid FSM for unsatisfied rules of valid FSMs

6.2.2.2 Second Test Case

An FSM might be too simple for any OR-states to be constructed. This means no two transitions at least with same labels entering the same target state. Figure 6.13 shows an FSM which has six states and nine transitions.

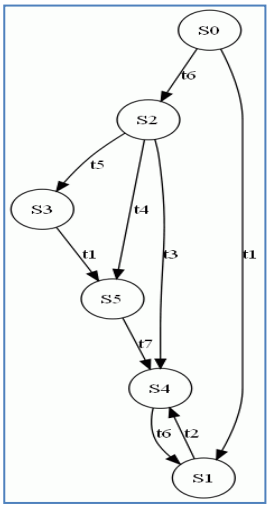


Figure 6.13: an FSM without transitions with same label entering same states

The conversion system demonstrates the previous FSM does not have any possible to construct OR-state from it, where the FSM shown in Figure 6.13 does not have at least two transitions with same label entering the same target state. The generated statechart is the same as the entered FSM without any composite states generated. Figure 6.14 illustrates the actual output obtained by the converter for FSM shown in Figure 6.13.

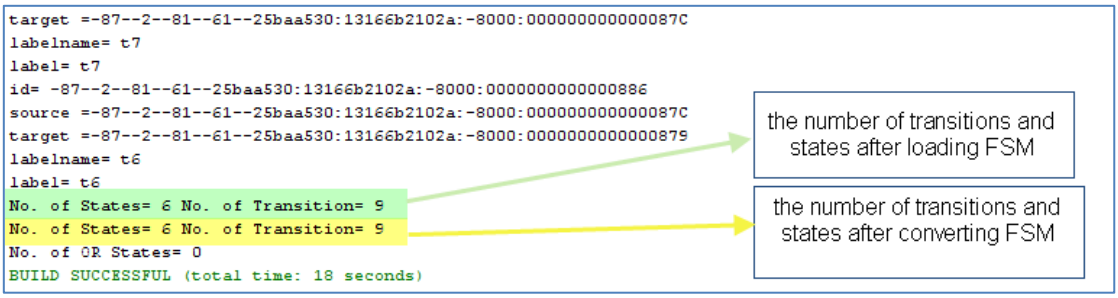


Figure 6.14: the actual output of the second test case

6.2.2.3 Third Test Case

This test case demonstrates that the generated statecharts might contain a single OR-state; the clustering approach is used to group states together to be components of an OR-state.

The input is an XMI file represents an FSM with 6 states and 9 transitions shown in Figure 6.15, where two transitions labelled with 't7' enter state 'S4', sources of those transitions are merged together to form an OR-state as the following statechart shows. After clustering those sources, transitions leaving sources (components of an OR-state) should be handled to reduce the number of transitions. It is reduced by one according to our expectation.

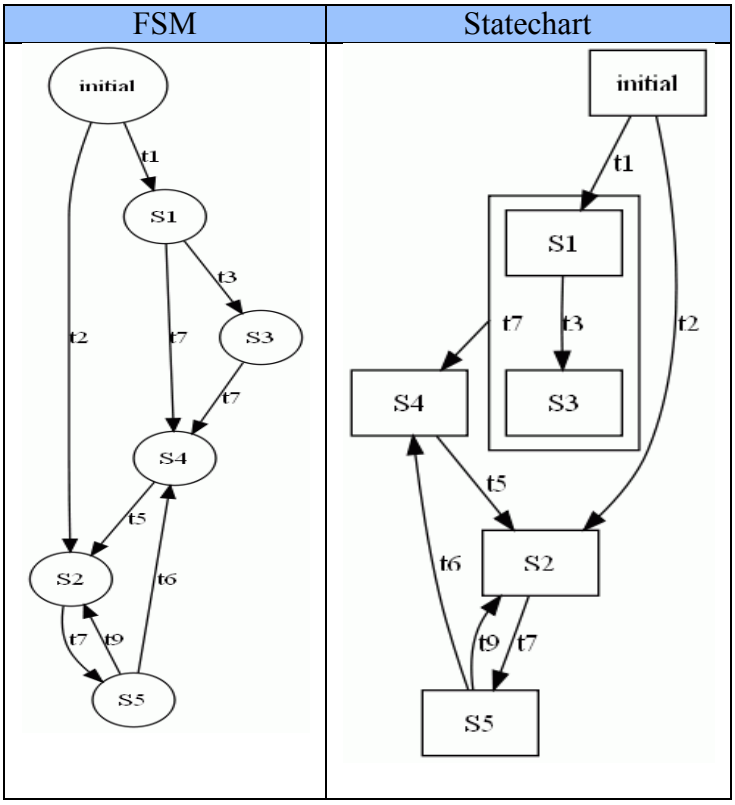


Figure 6.15: an FSM and the generated statechart of the third test case

Figure 6.16 illustrates the actual output of the converter after converting the entered FSM into the corresponding statechart, there is one possible OR-state detected in the loaded FSM. The generated statechart demonstrates that 'S1' and 'S3' are clustered together.

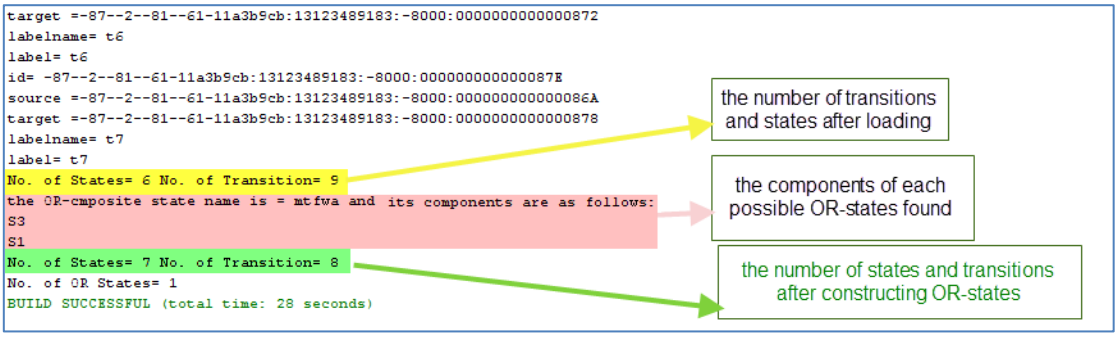


Figure 6.16: the actual output of the third test case

6.2.2.4 Fourth Test Case

The clustering approach that was implemented in our system supports the ability to have multiple independent composite states. In this case, there are two independent composite states.

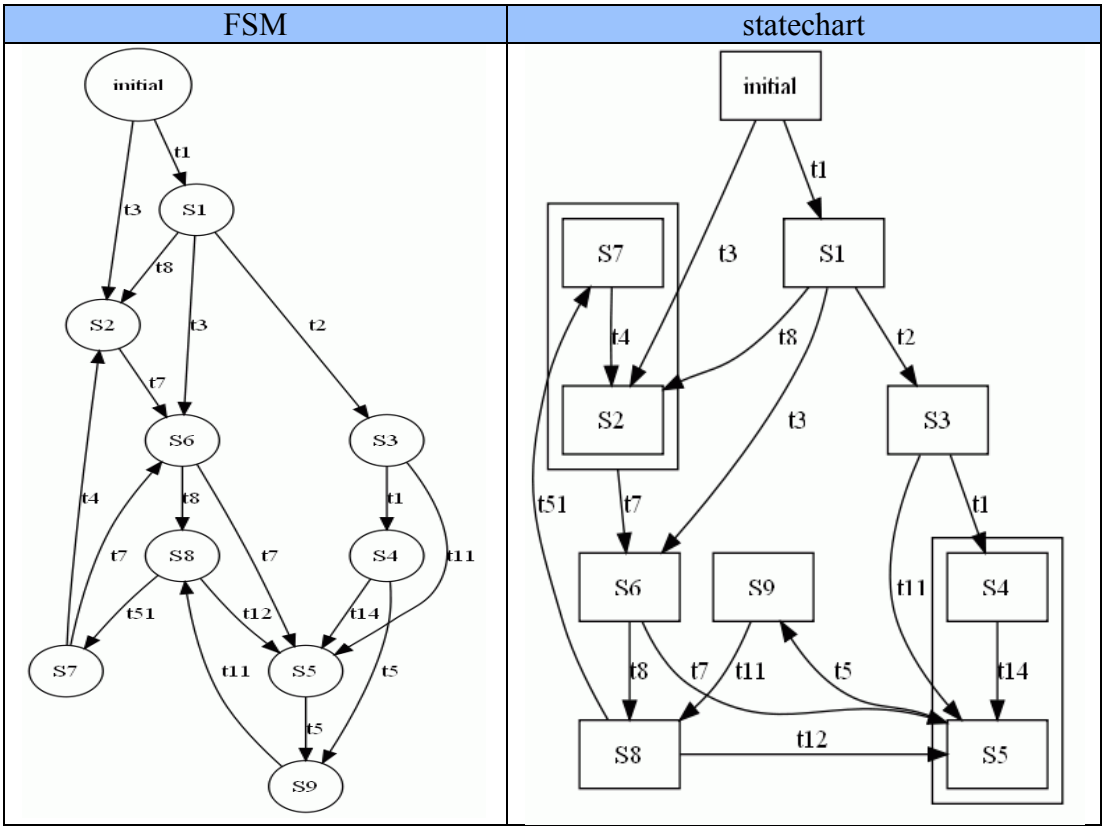


Figure 6.17: an FSM and the generated statechart of the fourth test case

An FSM illustrated in Figure 6.17 has many transitions that leave states to the same target state, such as three incoming transitions ‘t7’, ‘t7’, and ‘t3’ enter state ‘S6’. According to the clustering approach, two of entering transitions have the same label ‘t7’; source states of transitions are ‘S7’ and ‘S2’, these are grouped together. In addition, there is another OR-state is detect by the conversion system where its components are ‘S4’ and ‘S5’, these components are sources of the incoming transitions labelled with ‘t5’ reaching state ‘S9’. Figure 6.18 shows the actual output, where OR-states and their components are shown below:

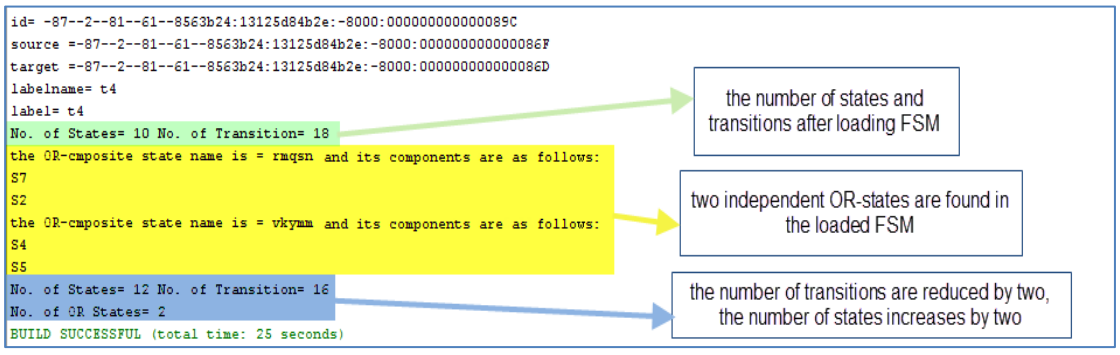


Figure 6.18: the actual output of the fourth test case

6.2.2.5 Fifth Test Case

The previous test case is shown that the number of components of each OR-state is two. In this case, five components are inside the clustered state to ensure that our system does not have any constraint on the number of OR-state components. The clustering method described in chapter 5 works without a limitation on the number of sub-states of OR-states it can construct.

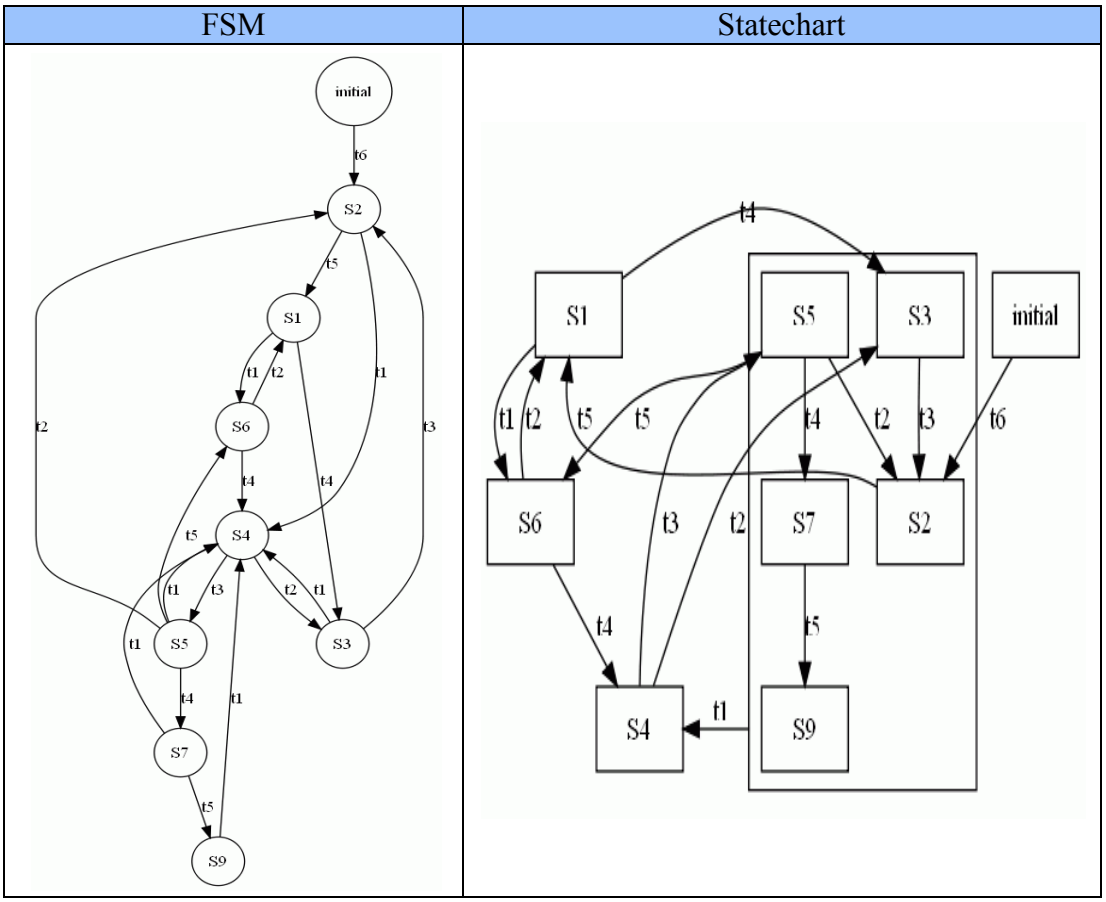


Figure 6.19: an FSM and the generated statechart of the fifth test case

The previous FSM in Figure 6.19 illustrates that label with 't1' has occurred six times on transitions and five of them have the same target state 'S4'. All sources of these transitions (S9, S3, S5, S7, and S2) are clustered together. Figure 6.20 shows that the number of transitions is reduced by 4, where the number of components in composite states has an effect on reducing the number of transitions.

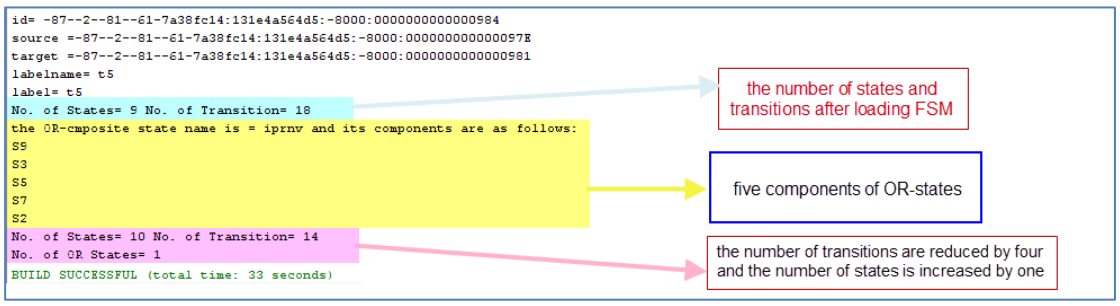


Figure 6.20: the actual output of fifth test case

6.2.2.6 Sixth Test Cases

The following test case demonstrates that multiple independent composite states are maintained by the conversion system using the clustering method. The input is a FSM which has twelve states and twenty-one transitions shown below, for example state 'S9' has three incoming transitions labelled with 't8', 't8', 't11'. Sources of transitions labelled with t8 are components of an OR-state.

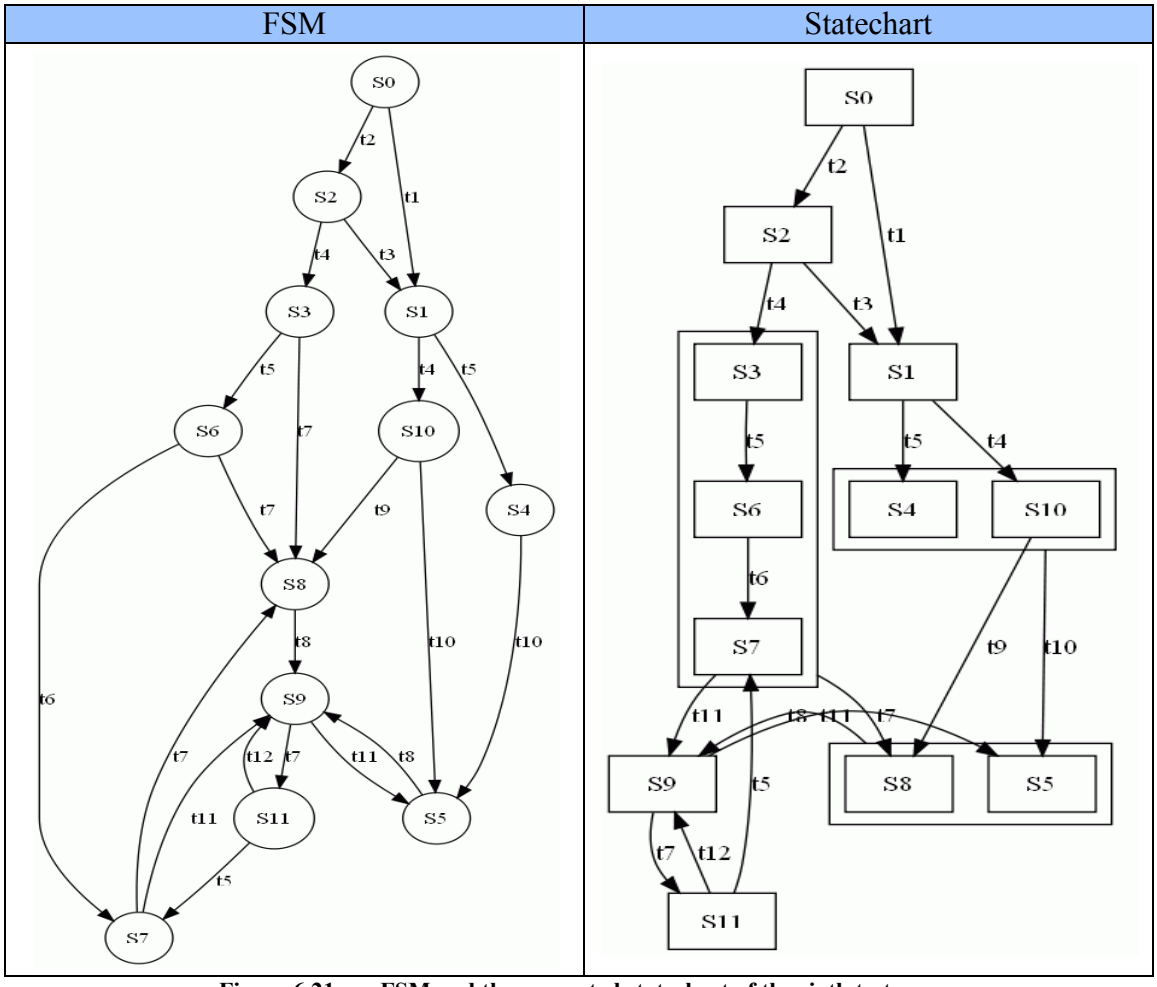


Figure 6.21: an FSM and the generated statechart of the sixth test case

Figure 6.22 shows the actual outcome after converting the previous FSM shown in Figure 6.21 into the corresponding statechart. Each component of OR-states occurs once, so all these OR-states are independent.

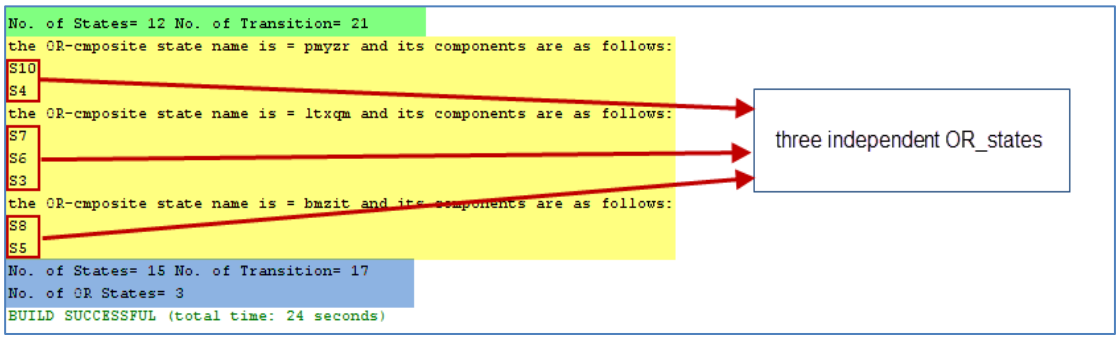


Figure 6.22: the actual output of sixth test case.

6.2.2.7 Seventh test Case

This test case concerns with the statechart with overlapping states, produced by the method described in chapter 5 to tackle with this case. The generated statechart shown in Figure 6.24 has a state shared between two clusters. The input FSM has 7 states and 12 transitions; states (S3, S4) have going transitions labelled with 't4' entering state 'S7', those states are components of composite state. However, two transitions with labels 't3' leaving states (S3, S2) and entering state 'S6', states (S3, S2) are components of another composite state. State 'S3' occurs in both components of different composite states, so this state will be shared between them.

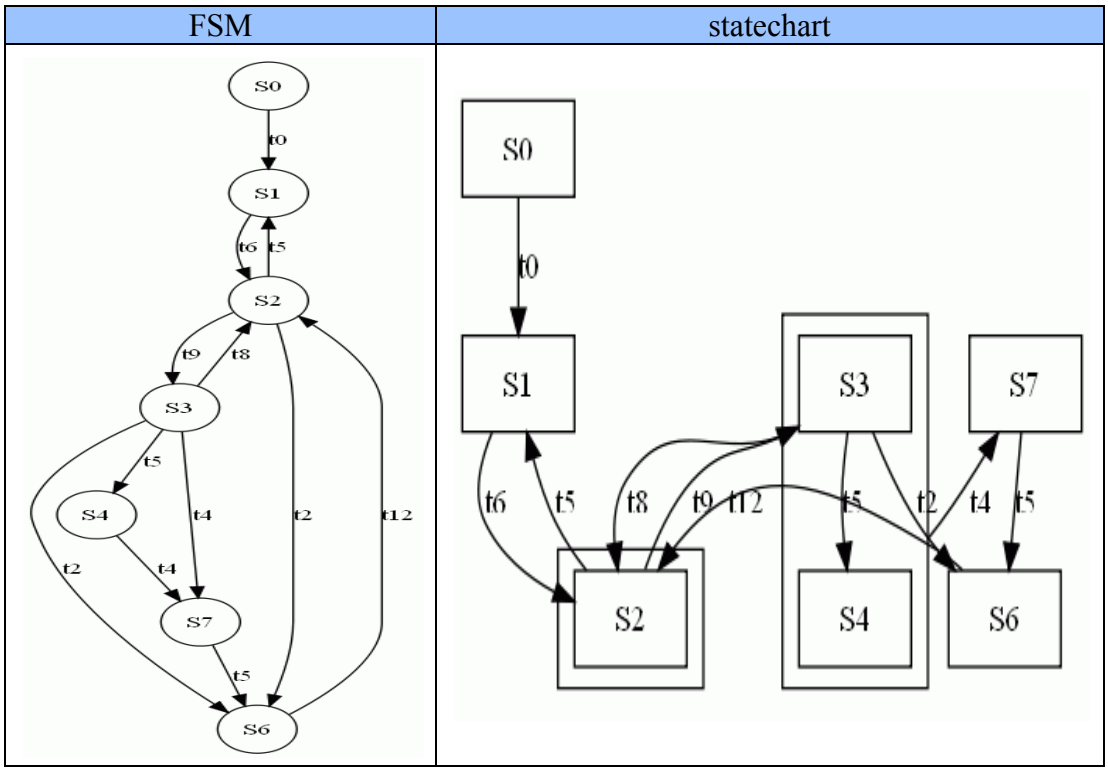


Figure 6.23: an FSM and the generated statechart of the seventh test case

Figure 6.24 illustrates the actual outcome obtained by the converter where state 'S3' is shared between two clusters. In this test case example without allowing the overlapping method, the number of transitions is reduced by one unlike in this case where the number is reduced by two. The lack of supporting overlapping between states by the DOT language has led the generated statechart to be appeared without sharing state 'S3'.

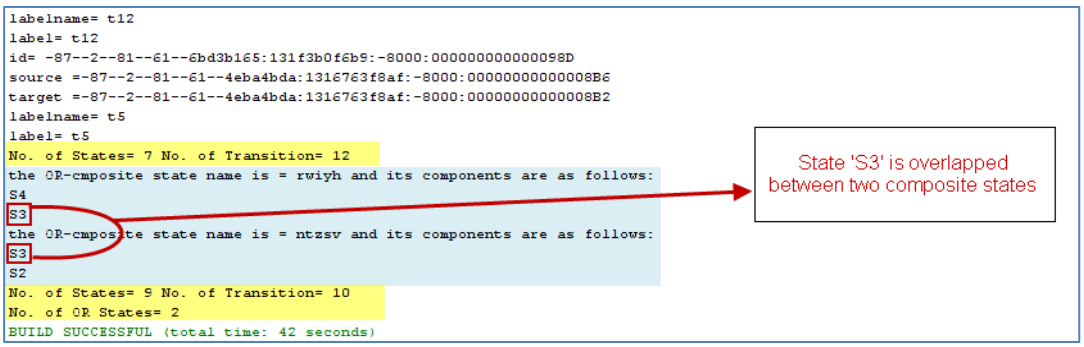


Figure 6.24: the actual output of the seventh test case

6.2.2.8 Eighth Test Case

This test case demonstrates that two nested OR-states might occur and our conversion system deals with this situation by detecting nested OR-states, this is done by implementing finding recursive clustering and overlapping method described in chapter 5 for all possible OR-states, where the input is the FSM with thirteen states and twenty-six transitions shown in Figure 6.25, where our expectation is generating statechart with three OR-states and two of them are nested in the largest one.

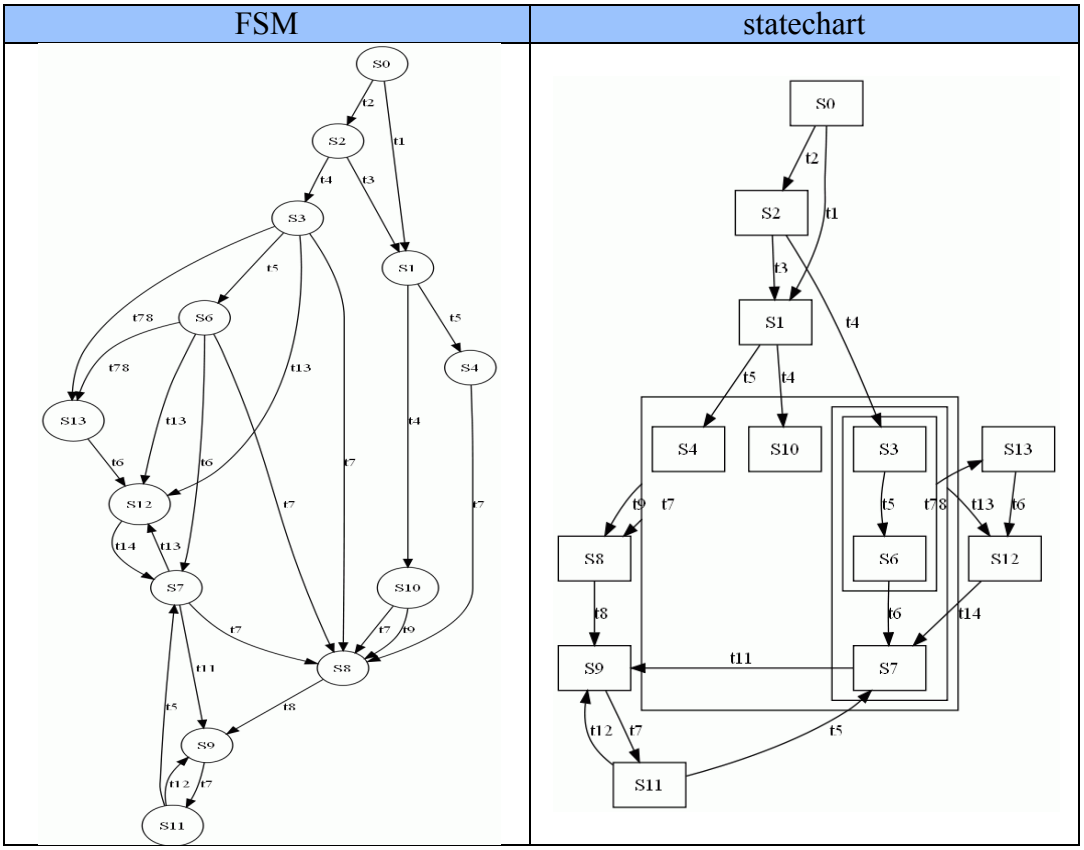


Figure 6.25: an FSM and the generated statechart of the eighth test case

The generated statechart illustrates that there are three composite states and two of them is nested in the largest OR-state, where first or-state is consist of two simple states (S3, S6), which is nested inside another larger OR-state which consists of three states (S3, S6, and S7). Both of these OR-states are nested in the largest OR-states which its components are five simple states (S4, S10, S3, S6, and S7). The reduction in number of transitions is clear in this test case where the number decreases from 26 to 19.

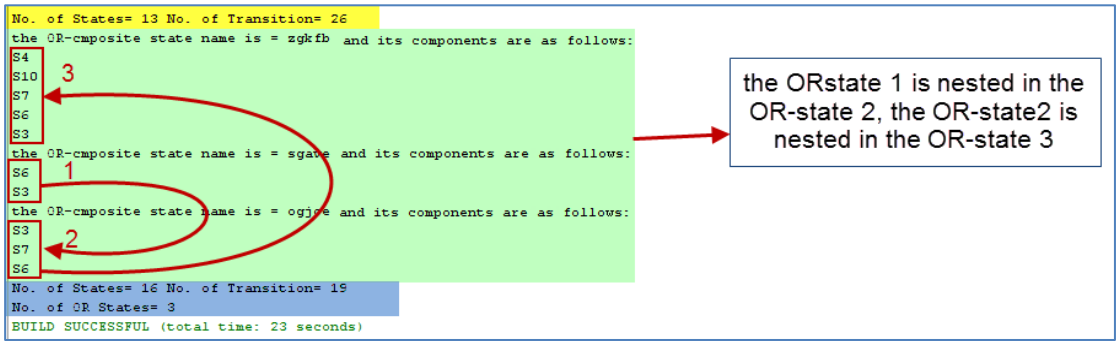


Figure 6.26: the actual output of the eighth test case

6.2.2.9 Ninth Test Case

This case is an important case in measuring the effectiveness of our system in cope with complex statecharts where independent and nested OR-states are found.

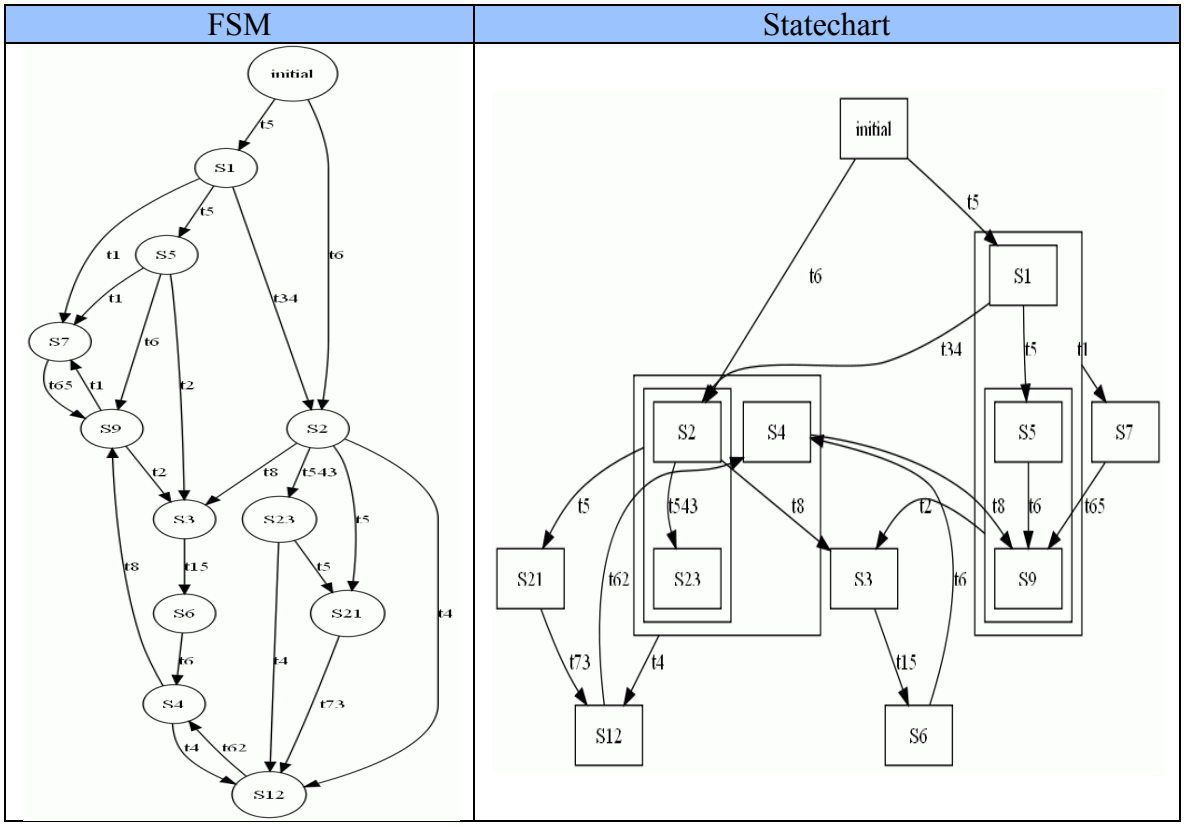


Figure 6.27: an FSM and the generated statechart of the ninth test case

Figure 6.28 shows the outcome of converting the previous FSM into a statechart, there are four OR-states detected by the conversion system; there are similar components of OR-states that occur many times in different OR-states. Those components lead to nested states. The reduction of the number of transitions is shown clearly when the nested OR-states occur.

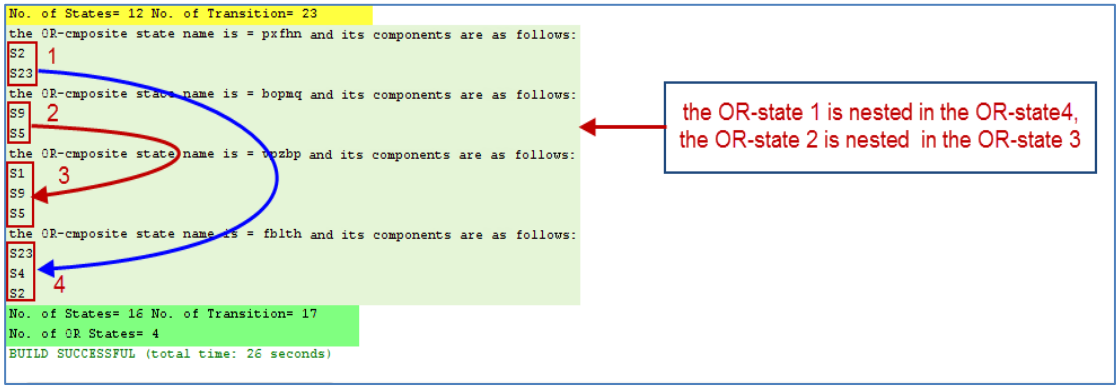


Figure 6.28: the actual output of the ninth test case

6.2.2.10 Tenth Test Case

Testing the possibility to have multiple independent and nested OR-states in the generated statechart is a crucial case to prove how the function of detecting nested states works successfully without any constraint on the number of nested states. In addition, this test case demonstrates how the converter able to identify nested composite states without any concerns about the complexity of the entered FSM.

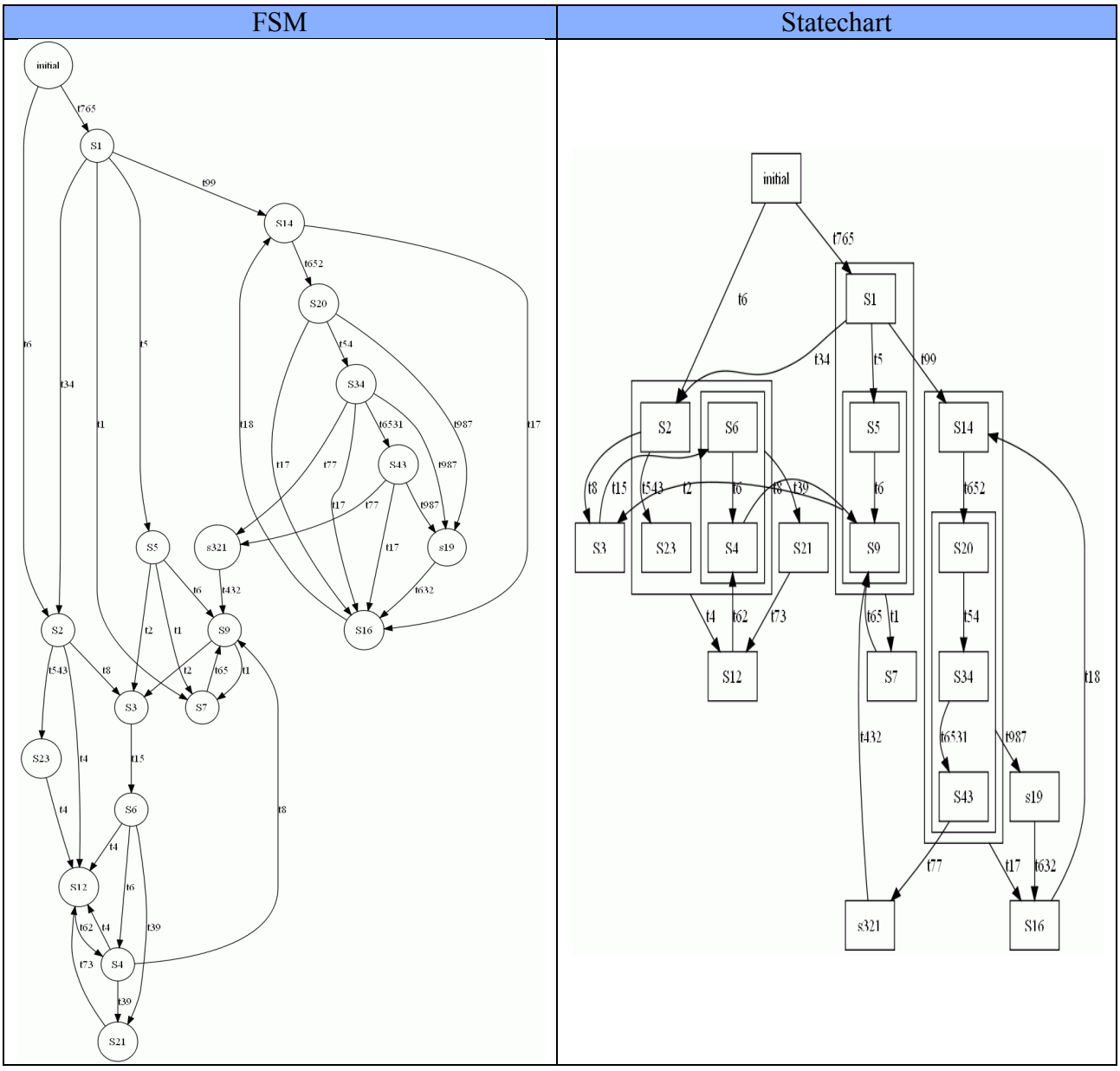


Figure 6.29: an FSM and the generated statechart of the tenth test case

Figure 6.30 illustrates the outcome obtained after loading the previous FSM in the converter, where seven OR-states are detected and the OR-state handler works on detected all nested states. This test case demonstrates how clustering and nested states have an effect in reducing the number of transitions from 40 to 27.

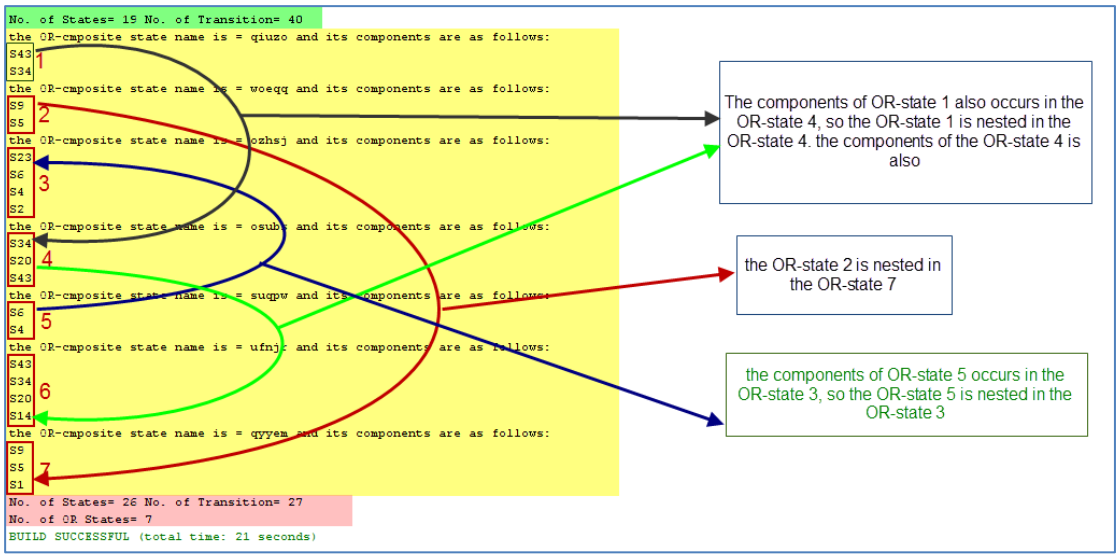


Figure 6. 30: the actual output of the tenth test case

6.2.2.11 Eleventh Test Case

This test case expresses that the generated statechart might contains overlapped states and independent composite states, the purpose of this testing is to check whether overlapping state found in different composite states will not affect another composite state. The input FSM shown in Figure 6.31 has ten states and seventeen transitions; states (S3, S4) have going transitions labelled with 't4' entering state 'S7', those states are components of composite state. The following generated statechart has shared state 'S3' between two clusters where this overlapping does not affect another composite state with components (S10, S9).

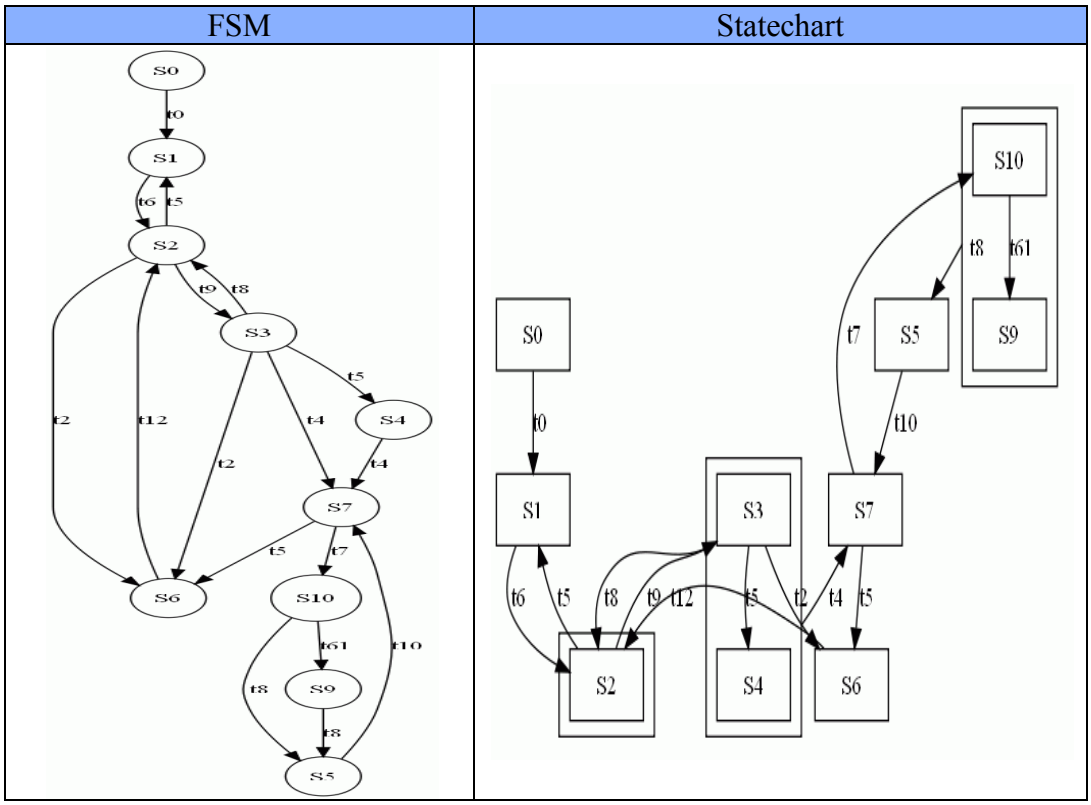


Figure 6.31: an FSM and the generated statechart of the eleventh test case

Figure 6.32 shows that state 'S3' occurs in both components of different composite states, so this state will be overlapped between them. Two states (S10, S9) are merged together because of those state has outgoing transitions labelled with 't8', where this composite state is independent without any relation with other composite states.

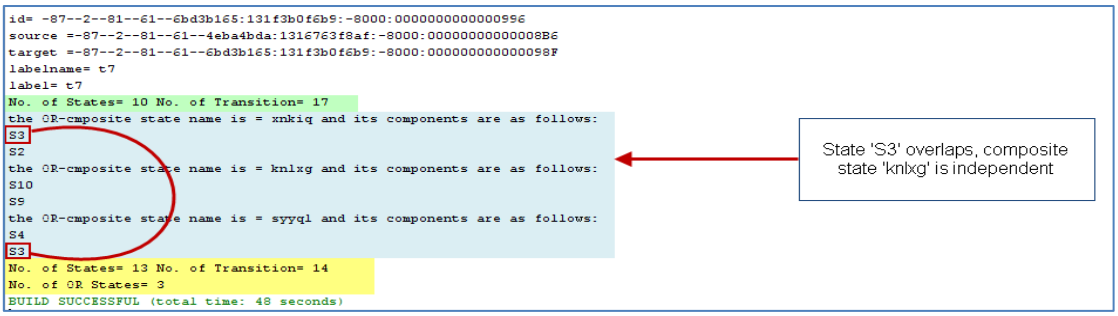


Figure 6.32: the actual output of the eleventh test case

6.2.2.12 Twelfth Test Case

The generated statechart might contain nested clusters and overlapped state between composite states. This test case is a vital test whether the developed system is able to generate statechart without any conflict between nested clusters and overlapped states in different composite states.

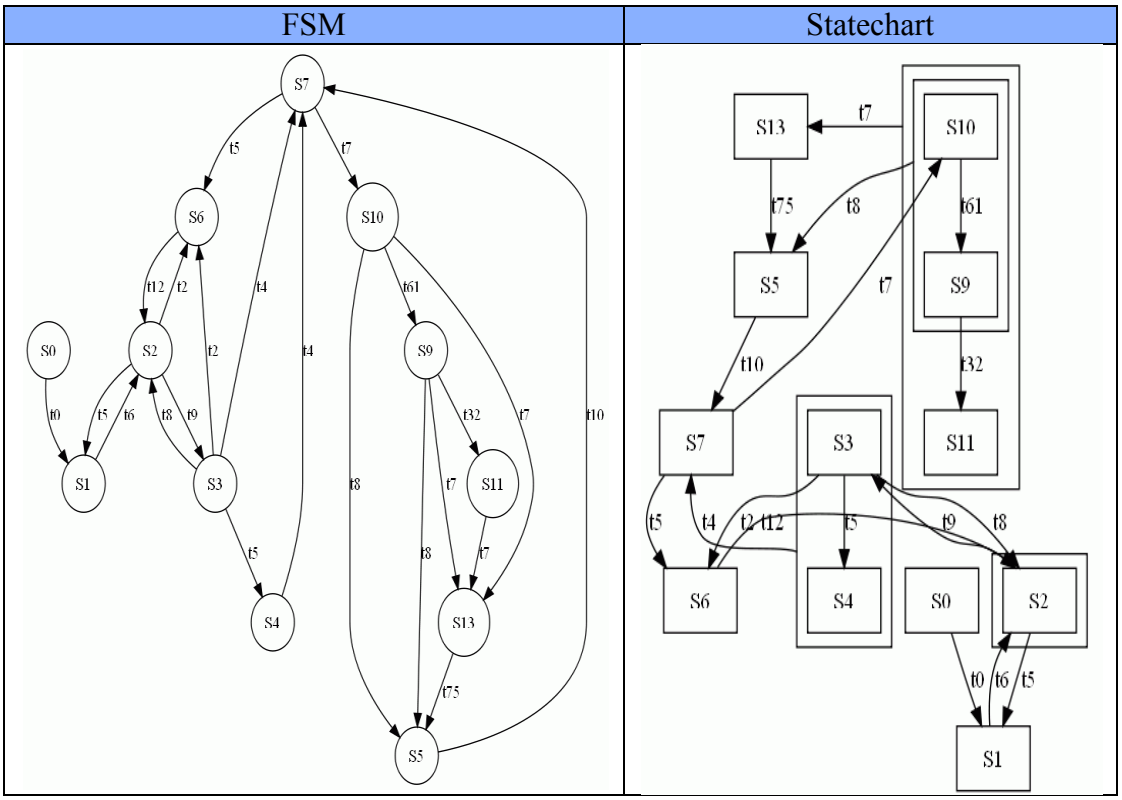


Figure 6.33: a FSM and the generated statechart of the twelfth test case

The outcome of the conversion system shown in Figure 6.34 express nested states is detected correctly. State 'S3' overlaps between two composite states, where the reduction of transitions is five. Without introducing the overlapping concept, the number of transitions will be reduced by four.

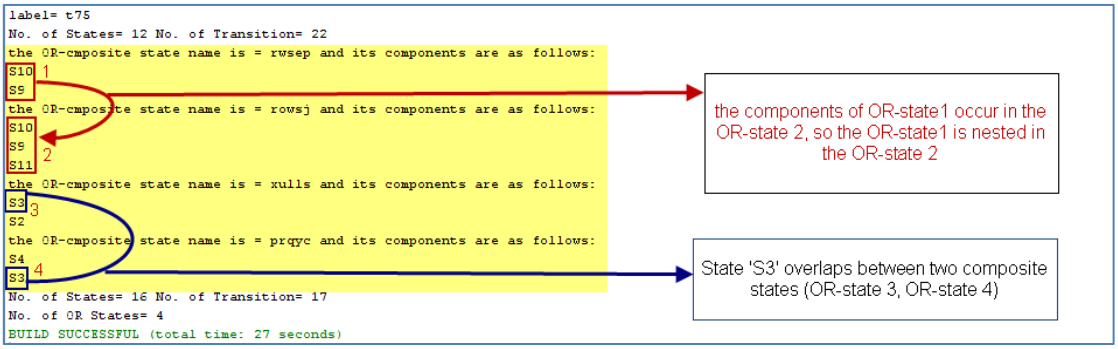


Figure 6.34: the actual output of the twelfth test case

6.2.3 JUnit Testing

JUnit is a framework for testing in java. It is used to ensure that different code segments work correctly. Figure 6.35 shows the output of JUnit testing that is used in developing the developed system.

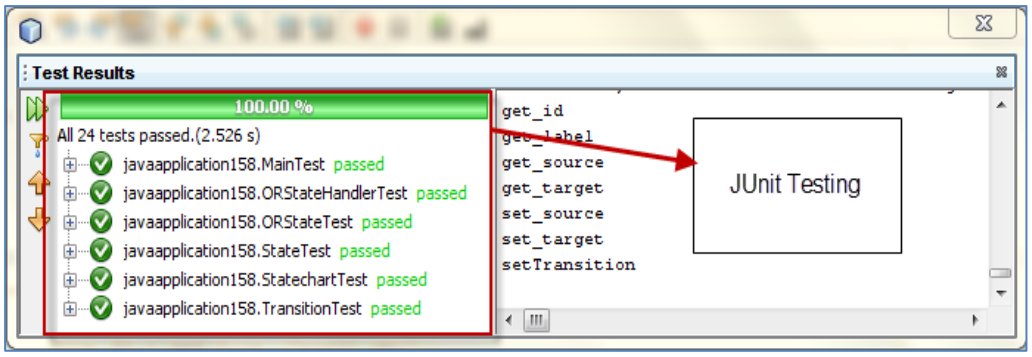


Figure 6.35: JUnit Testing

Generating JUnit testing is divided into different test cases depends on functions that are developed. The following list summarizes the functions that are tested.

- For loading a FSM into the conversion system, testing against the states and transitions that are expected to loaded. This is done by testing number of states and transitions, also testing loading FSM by comparing the expected states and transitions with actual states and transitions in the FSM to be loaded. Figure 6.36 illustrates a segment of test unit that is used to check the function of loading states and transitions.

Figure 6.36: Figure 6. 11: JUnit testing example of loading FSM

An assert facility supported by JUnit testing is used to test actual outputs versus expected results.

- For testing the generated composite states, defining expected composite states with their components. In addition, testing generating composite states includes two steps

of testing. First, testing the possibility of generating composite states which is a Boolean assertion testing, this testing is shown in Figure 6.37.

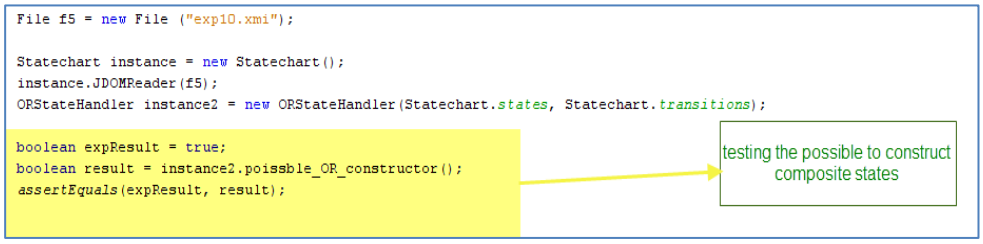


Figure 6.37: assertion testing of possible constructing composite states

Secondly, the testing of correct merging of the components of each composite state has been done using equal assertion between expected components of composite states with actual results; the following description summarizes the process of testing the generated composite state components with expected sub-states that are merged to form composite states.

- Defines the expected components of each composite state as below:

```
ArrayList composite_state_component1=new ArrayList(Arrays.asList("S2","S1","S7","S6","S5","S3"));
ArrayList composite_state_component2=new ArrayList(Arrays.asList("S2","S1"));
ArrayList composite_state_component3=new ArrayList(Arrays.asList("S6","S5"));
```

.
.

- Collect all generated composites states
- For each OR-state, collect it components in List
- Assert equality of the composite state components (simple states clustered) versus the correspond array list defined above, and so on with others

Figure 6.38 shows how JUnit testing is used to test components of the generated composite states.

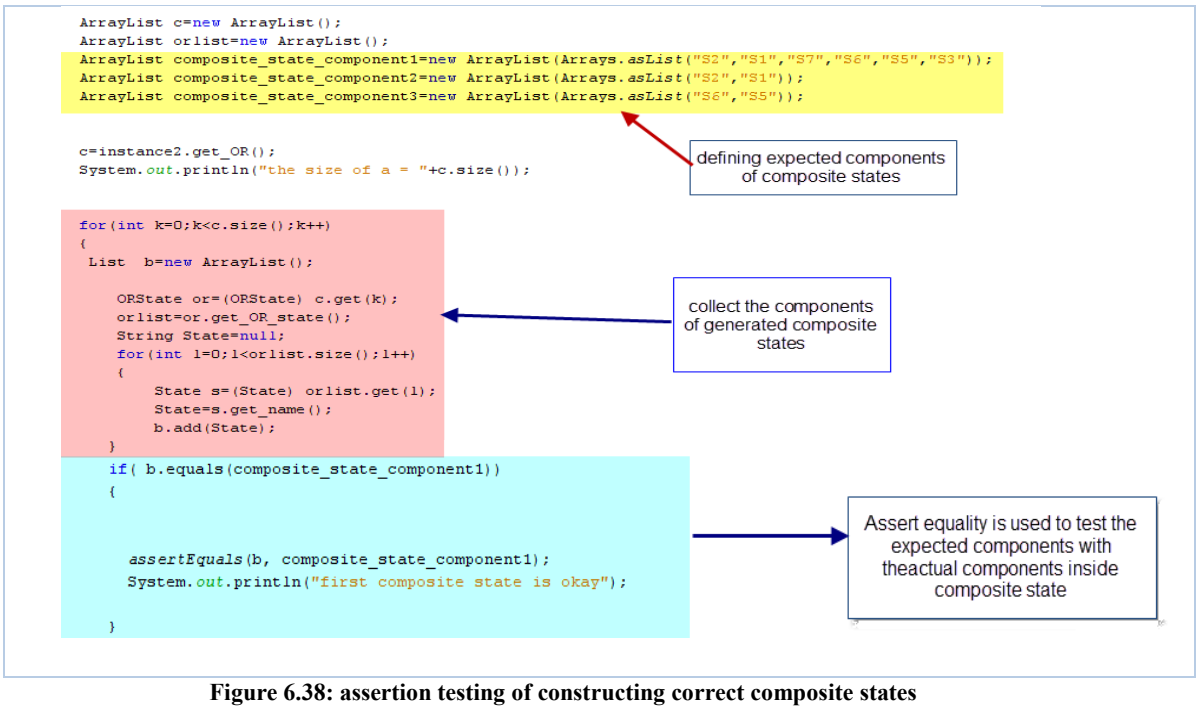


Figure 6.38: assertion testing of constructing correct composite states

Chapter 7: Results and discussion

In this section, the results that are acquired during developing the converter system will be described. In addition, the effect of introducing hierarchy into FSMs is to reduce the number of transition, where observations about introducing hierarchy to FSMs will be discussed in details. This chapter includes project goals achievements, implementation challenges, limitations, and future works.

7.1 Goals achieved

It can be said that the project achieved its goals in general as mentioned in chapter successfully. The developed system satisfies the aim that it is built for; converting FSM into statecharts is done by introducing hierarchy and concurrency. Studying different approaches of introducing them was a crucial aim by showing previous and related works. Loading FSM into the system includes checking its validity to ensure the loaded FSM is checking is achieved reachability and correctness, rejecting any invalid FSMs has been done with describing the reason of reject them. Finding possible composite states in the loaded FSM to construct statechart is accomplished in general. The generated system is used clustering approach to introduce hierarchy to FSMs.

7.2 The effect of composite states in reducing transition numbers

The experiment has done using 727 FSMs to study and measure how the developed system is able to construct composite states. No possible to present all those result here, some of them is selected to discuss the effect of introducing composite state and nested states on reducing the number of transitions.

These random DFA (Deterministic Finite Automata) are generated using a framework named as (STAMINA), an algorithm to generate DFA described by Bogdanov, K et al [32].The STAMINA framework depends on the deduction of a set of 100 random where the size of each is about 50 states. The generating of a random DFA, grammar inference competitions are implemented to generate transitions structures. A uniform probability that a pair of states are connected using a transition to generate random state machines. Each state must be reachable from the initial state in the generated DFA. Designing an algorithm to generate DFA depends on the following parameters where the aim is to ensure that FSMs generated reflect the structural of FSMs in real software:

- *Alphabet size*
  The number of states and transitions does not have any relationship with an alphabet size.
- *Relationship between depth and states*
  The depth has a relationship with the number of states, the following expression represent this relation:
  $$Depth = (0:36 * states) + 1.3$$
- *Degree distributions*
  For each state, the number of in/out degree would be between 1-2 in-going and out-going transitions. Scoring is used to interpret states to classify them into different type of states such as ‘root’ or other.

Table 10 summarizes our experiments to find out the effect of converting FSMs to statecharts by introducing hierarchy. The table shows the average number of transitions before and after converting FSMs into hierarchical statecharts. The difference between the averages number of

transition is counted, we notice that the high number in the average of reducing the number of transitions after generating statecharts is 10; the likelihood to construct composite states from random FSMs that are generated using a framework 'STAMINA' is limited, and this because random FSMs that are generated represent real systems, where the generated FSMs does not shown to have more than three numbers of transitions with same labels entering the same target state. The number of samples in each group is different such as FSMs with 20 states has 76 samples and FSMs with states 25 states has 10 samples. If the number of samples is larger, the possibilities to have more varied composite states are increased.

Table 10: information summarized using random FSMs

Numbers of states	Number of random FSMs	Total number of transitions	Average number of transitions	Average number of transitions after converting	The difference between the average number of transitions before and after converting	Average number of states	Average number of state after converting	The different between the average of states before and after converting
20	76	3472	46	42	-4	19	23	4
25	100	6053	61	56	-5	24	28	4
40	10	1086	109	106	-3	39	44	5
45	100	12219	122	116	-6	44	50	6
60	10	1753	175	168	-7	59	67	8
65	100	19866	182	175	-7	64	70	6
80	10	2370	237	229	-8	80	87	7
85	100	24572	246	238	-8	84	91	7
100	10	3046	305	295	-10	99	108	9
105	100	30789	308	298	-10	104	111	7
120	10	3659	366	358	-8	120	127	7
125	100	37112	371	363	-8	124	132	8

Figure 7.1 shows that the number of transitions is reduced depending on the number of samples of FSMs in each group of FSM. These numbers are averages where some of FSMs have shown a clear reduction in transitions depends on number of transitions with same labels entering the same target. However, a clear increasing of the number of states is shown, and this due to constructing new composite states. The total number of states includes simple states and composite states.

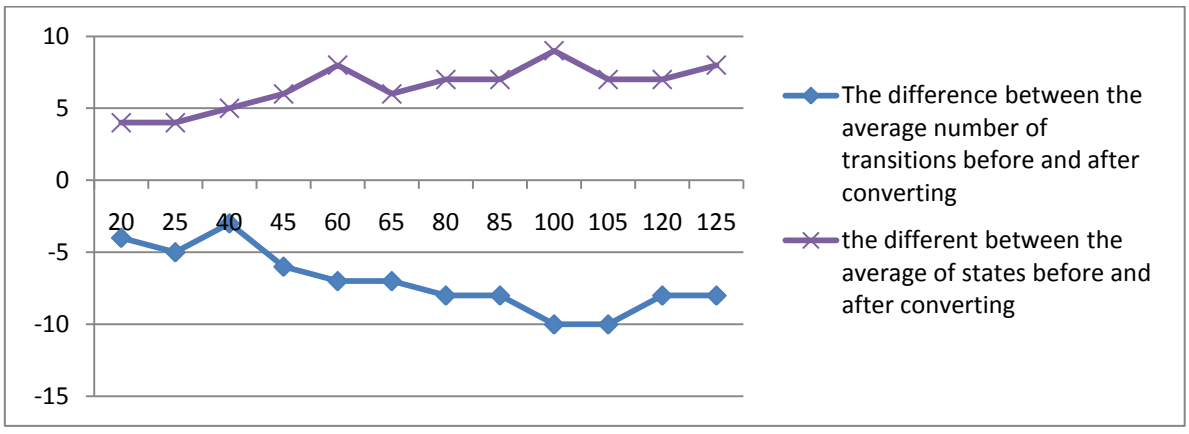


Figure 7.1: the difference of number of transition and states after generating statecharts

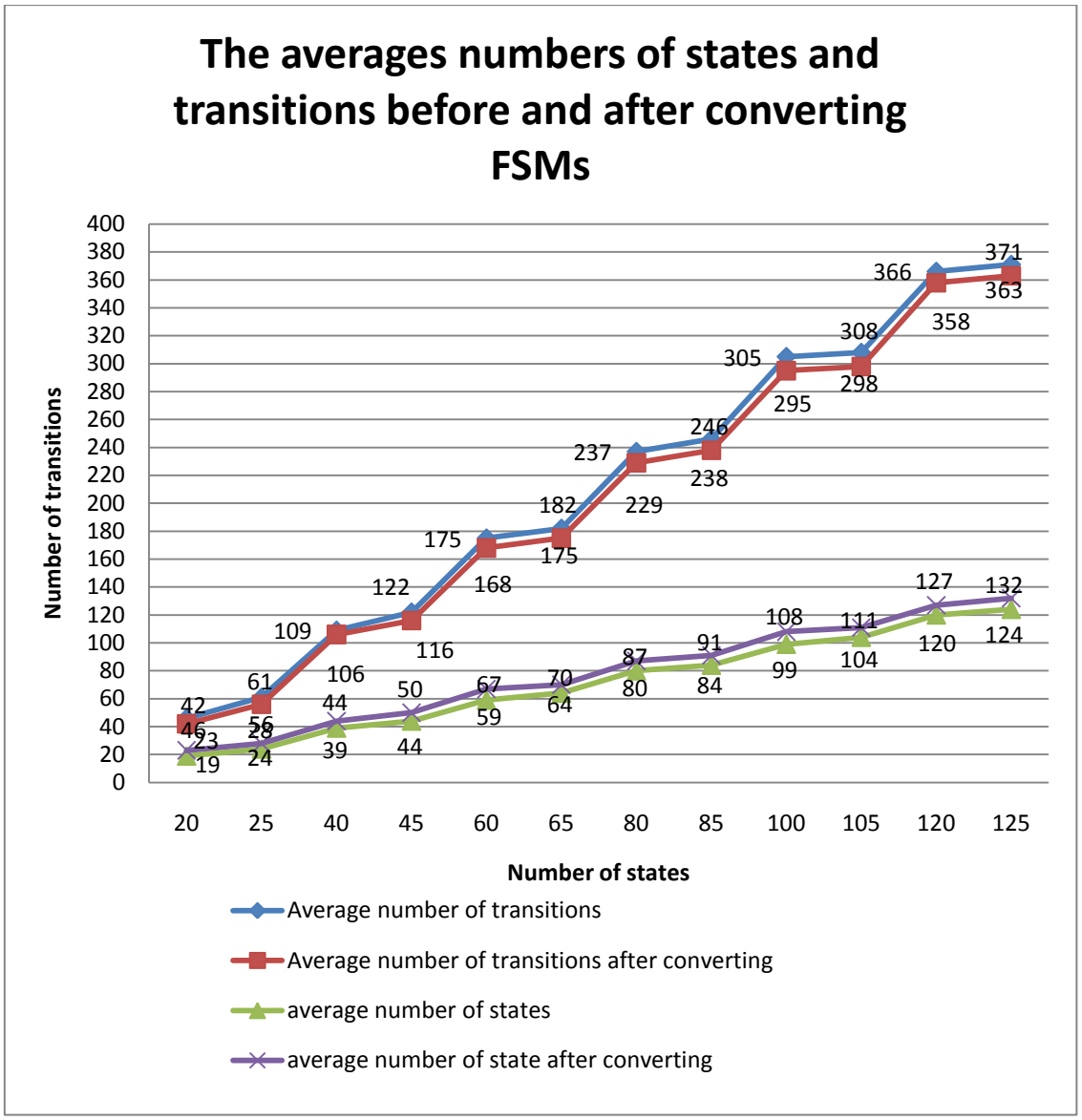


Figure 7.2: The averages numbers of states and transitions before and after converting FSMs

Figure 7.2 illustrates the variation between the average numbers of transitions after constructing OR-state. However, the random FSMs that are generated automatically using 'STAMINA' support the single level of hierarchy (depth) of composite states. Because of no way to present more than one level of depth by introducing recursive (nested) composite states in those random FSMs, the reduction of number of transitions is not obvious as expected. The impact of nested level on the understandably in the generated statecharts compared with those statecharts that are generated described in chapter 6 such as the tenth test cases where the number of transitions is reduced by thirteen.

Table shows different FSMs, where some of them are generated manually to demonstrate that introducing nested states has an effect on reducing the complexity of the generated statechart. The following table illustrates some selected FSMs to see the reduction of transition and the relationship between reduced transitions and the number of composite states:

Table 11: different FSMs generated manually and automatically using STAMINA

	No of transitions	No of transitions after OR-composite state	No of States	No of states after OR-composite state	The reduction of transition numbers	No of composite state
Fsm1	9	8	6	7	1	1
Fsm2	11	10	7	8	1	1
Fsm3	7	5	4	6	2	2
Fsm4	12	10	8	10	2	2
FSM5	15	8	8	11	7	3
Fsm6	18	14	10	13	4	3
Fsm7	24	17	12	16	7	4
Fsm8	23	17	12	16	6	4
Fsm9	33	24	16	21	9	5
Fsm10	32	22	16	22	10	6
Fsm11	41	29	19	25	12	6
Fsm12	40	27	19	25	13	6
Fsm13	59	49	20	27	10	7
FSM14	80	71	25	33	9	8
FSM15	150	141	43	52	9	9
FSM16	161	148	43	53	13	10
FSM17	148	135	43	56	13	13
FSM 18	122	110	43	54	12	11
FSM19	373	353	123	142	20	19
FSM20	435	419	123	139	16	16

Increasing the number of composite states and their components that might be obtained from the generated statecharts has an effect on reducing the number of transitions. The reduction of transitions ranges between one and two for each composite state that is generated when FSMs is generated using 'STAMINA', this occurs because of the hypothesis that are supposed in 'STAMINA' framework of linking states using transition by 1 or 2 incoming and outgoing edges [32].

The nested composite states have an apparent effect on reducing the number of transitions. For instance, FSM-5 that are generated manually has been constructing three OR-states from it using clustering approach and two of them are nested inside the largest size of OR-state; the FSM-5 is a crucial example to illustrate the effect of nested composite states on the number of transitions, where the FSM -5 has reduced the number of transitions by seven. FSM-12 is an example of the effect of nested states and overlapping between them, where the number of transitions is reduced by 13. However, the random generating of FSMs using 'STAMINA' is

shown that nested clustering has a low probability to occur unlike overlapping between clusters that occurs clearly. For instance, FSM-13 has a state 'V2' overlaps in different clusters, and only one nested cluster occurs.

7.3 Limitations

The limitation of the DOT language in coping with overlapping between nodes has led to lack of visualising overlapping between clusters components, GraphViz developers work to solve this issue. There is another utility in GraphViz tool which is 'neato'; this utility deals with drawing graphs with nodes overlapping in some cases such as graph shown in Figure 7.3.

Figure 7.3 illustrates how 'neato' presenting overlapping between clusters, however 'neato' feature does not work successfully in many cases such as if there is more than two nodes shared in different clusters.

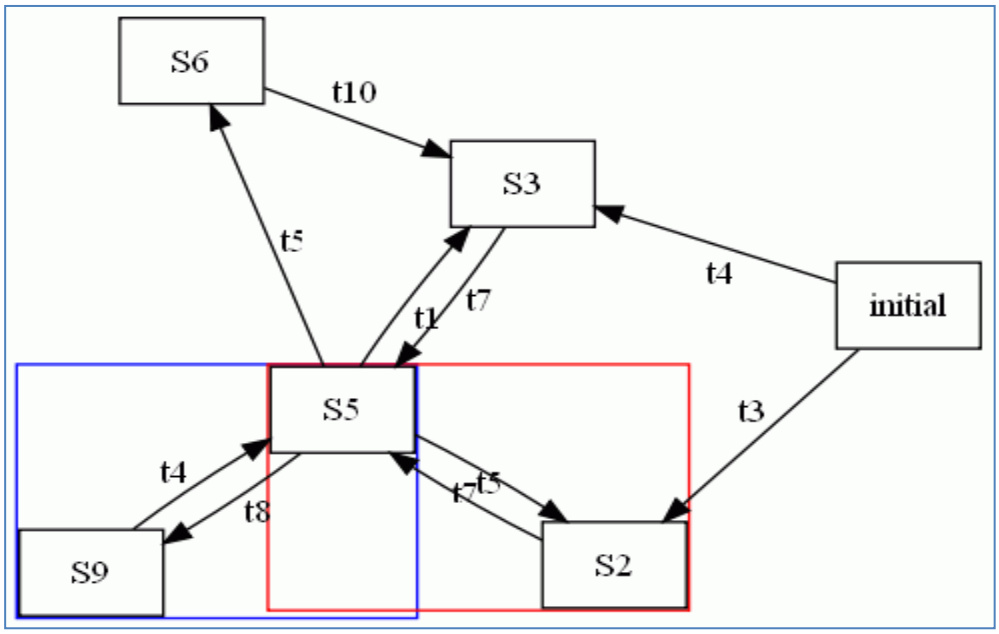


Figure 7.3: overlapping state using 'neato'

7.4 Future Works and Further Improvements

The tool works successfully on converting FSMs into statecharts, however this system might be extended to accept different formats of input of FSMs rather than using XMI format. Finding an effective method to generate AND-states to hierarchical FSMs to make the generated statecharts support concurrency using some techniques mentioned in chapter 3.

This developed tool might be used in different fields such as using the generated statecharts to generate database schema using converting statecharts into class diagrams. It might be used to generate code automatically from the generated class diagram. Measuring and testing the generated code using original FSM.

Chapter 8: Conclusions

To summarise, this project aimed to convert FSMs into statecharts automatically by introducing hierarchy and concurrency to the loaded FSMs. This has motivated the research study to identify different concepts related to statecharts in order to identify elements that are needed to taken in consideration to accomplish the conversion processes. In this works a tool is designed to convert FSMs into statecharts where the behaviour is preserved during converting them.

Numbers of related works of reverse engineering between statechart and other UML models such us generating statecharts from sequence diagrams are considered in this work to indentify techniques and theories that helped us to achieve it. In addition, the purpose of investigating similar conversion between different models into statecharts is summarized in two aims: investigating exciting techniques and theories to introduce hierarchy and concurrency and avoid limitations occurred in different conversion. By achieving those aims, beneficial methods are derived from variety theories described in chapter 3. Case study has been done depending on different techniques are extracted from related researches to help us in introducing hierarchy, also decision making about the way of merging simple states to be components of composites state has been accomplished using metrics in chapter 3.

Synthesizing a statechart from a FSM was performed using different techniques such as clustering approach described in chapter 5 to introduce hierarchy. The aim of constructing statecharts as described in chapter 1 is reducing the complexity found in FSMs. Introducing nested states and overlapping has the considerable impact on reducing the number of transitions in FSMs to achieve the aim that we work for. However, adding hierarchy on random FSMs that are generated using 'STAMINA' framework does not observe an obvious reduction of the number of transitions, and this because of the number of transitions with same labels are ranged between 1 -2 enter target states.

Metrics are used to evaluate the generated statecharts against FSMs such as the number of transitions that are reduced after converting FSMs into statecharts, and to measure and testing the expected number of transitions and states versus the observed numbers of states and transitions after converting FSMs into statecharts. Moreover, measuring the expected composite states with the actual composite states in the generated statechrats is one effective usage of metrics in evaluating our works.

Number of test cases was used in order to assess and examine the ability of converting different FSMs into statecharts using the developed system. The results of those test cases demonstrate that the tool is designed well; measuring these results has been done by comparing the acquired outcomes with expected results and fulfilling the aims that we have developed the tool to achieve them.

References

[1] Chu, H., Li, Q., Hu, S. and Chen, P. (2006) An approach for reversely generating hierarchical UML statechart diagrams. *Fuzzy Systems and Knowledge Discovery, Proceedings,* 4223**,** 434-437.

[2] Cruz-Lemus, J., Genero, M., Manso, M. and Piattini, M. (2005) Evaluating the effect of composite states on the understandability of UML statechart diagrams. *Model Driven Engineering Languages and Systems, Proceedings,* 3713**,** 113-125.

[3] Cruz-Lemus, J., Genero, M., Piattini, M. and Toval, A. (2005) An empirical study of the nesting level of composite states within UML statechart diagrams. *Perspectives in Conceptual Modeling,* 3770**,** 12-22.

[4] Cruz-Lemus, J. A., Maes, A., Genero, M., Poels, G. and Piattini, M. (2010) The impact of structural complexity on the understandability of UML statechart diagrams. *Information Sciences,* 180**,** 2209-2220.

[5] Latella, D., Majzik, I., AND Massink, M. (1999) Towards a formal operational semantics of UML statechart diagrams. *In 3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems.* Boston, Kluwer Academic Publishers.

[6] Douglass, B. Powel (2001) State machines and Statecharts. *Embedded System Conference.* San Francisco, USA.

[7] Douglass, B. Powel (2007) *Real time UML workshop for embedded systems,* Burlington, Mass., Elsevier.

[8] Mikk, E., Lakhnech, Y., Petersohn, C. And Siegel, M. (1997) On formal semantics of Statecharts as supported by STATEMATE. In *2nd BCSFACS Northern Formal MethodsWorkshop*, Springer-Verlag.

[9] Eshuis, R. (2009) Reconciling statechart semantics. *Science of Computer Programming,* 74**,** 65-99.

[10] Gansner, E. R. (2009) Drawing graphs with *Graphviz*. *Graphviz Drawing Library Manual.*

[11] Gansner, E. R., Koutsofios, E. and North, S. (2009) Drawing graphs with dot. *dot User's Manual.*

[12] Genero, M., Miranda, D. and Piattini, M. (2003) Defining metrics for UML statechart diagrams in a methodological way. *Conceptual Modeling for Novel Application Domains, Proceedings,* 2814**,** 118-128.

[13] Graphviz.Org. *Graphviz - Graph Visualization Software* [Online]. Available: http://www.graphviz.org/ [Accessed June 11 2011].

[14] Grässle, P., Baumann, H., Baumann, P., Smith, C. and Chakrabarti, P. (2005) *UML 2.0 in action a project based tutorial,* Birmingham, U.K., Packt Pub.

[15] Grose, T. J., Doney, G. C. and Brodsky, S. A. (2002) Mastering XMI Java programming with XMI, XML, and UML. New York, John Wiley & Sons.

[16] Harel, D. (1986) Statecharts: A visual formalism for complex systems. *Science of Computer Programming Program.8*, 3, 231–274.

[17] Harel, D. and Kahana, C. (1992) *On statecharts with overlapping.* Rehovot, Israel, Dep. of Applied Mathematics and Computer Science, Weizmann Inst. of Science.

[18] Harel, D. and Naamad, A. (1996) *The STATEMATE semantics of statecharts.* Rehovot, Israel, Weizmann Institute of Science, Dept. of Applied Mathematics and Computer Science

[19] Horrocks, I. (1999) *Constructing the user interface with statecharts,* Harlow, England, Addison-Wesley.

[20] JDOM (2011) JDOM - Wikipedia, the free encyclopedia. *Wikipedia, the free encyclopedia.* Available: http://en.wikipedia.org/wiki/JDOM [Accessed April 18 2011].

[21] JDOM.ORG. *JDOM* [Online]. Available: http://www.jdom.org/ [Accessed April 22 2011].

[22] Khriss, I., Elkoutbi, M. and Keller, K. (1999) Automating the synthesis of UML StateChart diagrams from multiple collaboration diagrams. *Unified Modeling Language: Uml'98: Beyond the Notation,* 1618**,** 132-147.

[23] Kumar, A. (2008) *A Novel Technique to Extract Statechart Representation of FSMs*. *Department of Computer Science and Engineering.* West Bengal, India, Indian Institute of Technology Kharagpur.

[24] Kumar, A. (2009) SCHAEM: A Method to Extract Statechart Representation of FSMs. *2009 Ieee International Advance Computing Conference, Vols 1-3***,** 1556-1561.

[25] Genero, M., Piattini-Velthuis, M., Cruz-Lemus, J.AND Reynoso, L. (2004) Metrics for uml models. *The European Journal for the Informatics Professional,* V**,** 43–48.

[26] Massol, V. and Husted, T. (2004) *JUnit in action* Greenwich, Conn., Manning.

[27] Ostrand, T. J. and Balcer, M. J. (1988) The category-partition method for specifying and generating functional tests. *Communications of the ACM,* 31**,** 676–686.

[28] Robbins, J. and Redmiles, D. (2000) Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Information and Software Technology,* 42**,** 79-89.

[29] Rumbaugh, J., Jacobson, I. and Booch, G. (2005) The unified modeling language reference manual 2nd ed. Boston, Addison-Wesley.

[30] Whittle, J. and Schumann, J. (2000) Generating Statechart Designs From Scenarios. *In International Conference On Software Engineering (ICSE 2000).*

[31] Systa, T., Koskimies, K. and Makinen, E. (2002) Automated compression of state machines using UML statechart diagram notation. *Information and Software Technology,* 44**,** 565-578.

[32] Walkinshaw, N., Bogdanov, K., Damas, C., Lambeau, B. and Dupont, P., (2010) A framework for the competitive evaluation of model inference techniques, *MIIT 2010 - Proceedings of the 1st International Workshop on Model Inference In Testing, Held in Conjunction with ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2010 , pp. 1-9.

[33] Wellner, P. (1989) Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. *CHI '89 Conference Proceedings.* New York.

[34] WIKIPEDIA (2011). DOT language - Wikipedia, the free encyclopedia. Available: http://en.wikipedia.org/wiki/DOT_language [Accessed April 23].

[35] Wu, X., Cox, B.D., and Esterline, A.C. (2000) Representing and Interpreting Multiagent Plans with Statecharts. Maui,Hawaii, World Automation Conference.

Appendix A: completed describing classes described in the class diagram:

- ***State class:***
  - **Attributes:**
    - String xmi_id: this property is used to present the unique id of each state loaded from a FSM.
    - String name: each state exported from ArgoUML has a name; this attribute is used to represent the name of each state found in the loaded FSM.
    - ArrayList outgoing_transitions: each state has many outgoing transitions; this information is used for validating the loaded FSM.
    - ArrayList incoming_transitions: incoming transitions are shown in any valid state, using this information to check these incoming transitions for a target state to find possible OR-states.
    - Boolean is_initial: this attribute will be false for all simple states except the initial state in any FSM.
  - **Methods:**
    - Get_id: returns a string representing the ID of state instance.
    - Get_name: return a string is used to obtain the name of each state to write it for generating DOT files.
    - Set_outgoing_transitions(ArrayList outgoing): this method is called to set outgoing transitions of each state while loading any FSM.
    - Set_incoming_transitions(ArrayList): is used to set incoming transitions related to each loaded state.
    - Get_outdoing_transitions(): returns an ArrayList of outgoing transitions related to each state.
    - Get_incoming_transitions: returns an ArrayList of incoming transitions related to each state.
    - Set_composite_state(Boolean b): used to set the status of the generated composite (ORstate, ANDstate) to be composite state kind of state where the default of each state is a simple state.
    - Is_composite_state: returns a boolean value, where writing the clustering as DOT format is required to distinguish composite states from simple states.

- ***Transition class:***
  - **Attributes:**
    - String transition_id: presents the unique id of each transition loaded from a FSM.
    - State transition_source: used to represent the source state of each a transition instance and to construct components of composite states.

- State transition_target: used to present the target of transitions, where this attributes is valuable for knowing incoming transitions to each target to construct composite states.
- String transition_label: presents the event of transitions such as ‘t1’, ‘t2’, and so on.

- **Methods:**
  - Get_id(): returns a string representing the ID of transition instances.
  - Get_label(): returns a string describing the label of each transition.
  - Get_source(): returns the source of transition.
  - Get_target(): returns the target of each transition object.
  - Set_source(State source): sets a new source for any transition. This method will be used during the transition handling when a new composite state is generated that is required to change sources of transitions that are related to the simple state (composite components) to the generated composite state.
  - SetTransition(State source,State target, String label): this method is invoked to create an instance of transition reading each transition from an XMI file.

- ***ORState Class:***

This class inherits from Composite Class all attributes and methods defined in it. The following attributes and methods are defined in the ORState class:

- **Attributes:**
  - String id: presents the unique ID of the constructed ORstate.
  - ArrayList <ORState> recursive: this is an important property to assign a nested ORstate to another ORstate instance.

- **Methods:**

  - setORState(String xmid,ArrayList <State> state): used to set a series of the components of the ORstate instance, where those components is an ArrayList of simple states.
  - get_OR_state(): returns an ArrayList of the components of ORstate.
  - Set_recursive(ArrayList <ORState> or): used to set a nested ORstate to another instance of ORstate.
  - get_recursive(): returns an ArrayList of ORstate instances nested in the instance of ORstate.

- ***CompositeState Class***:

  This class inherits from State Class all attributes and methods defined in it. The following attributes and methods are defined in the CompositeState class:

- **Attributes:**
  - String label: used to present the label of each composite state.
  - ArrayList <State> State: represents the component of ORstate instance, where an ORstate instance inherits all attributes and methods defined in this class.
  - ArrayList <ORState> ORState: this property is defined to present the component of ANDstate instances.
- **Methods:**
  - set_Label(String label): sets the generated composite states (ORstate , ANDstate) with label.
  - get_Label(): returns a label related to each composite state.

- ***ANDState Class:***

  This class inherits from Composite Class all attributes and methods defined in it. The following attributes and methods are defined in the ANDstate class:

- **Attributes:**
  - String id: presents the unique ID of the constructed ANDstate.
- **Methods:**
  - setANDState(String xmid,ArrayList <ORState> state): used to set a series of the components of ANDstate instance which is an ArraryList of ORstates.
  - get_AND_state(): returns an ArrayList of the components of ANDstate.

Appendix B: FSMs Formats

FSM.XMI Format

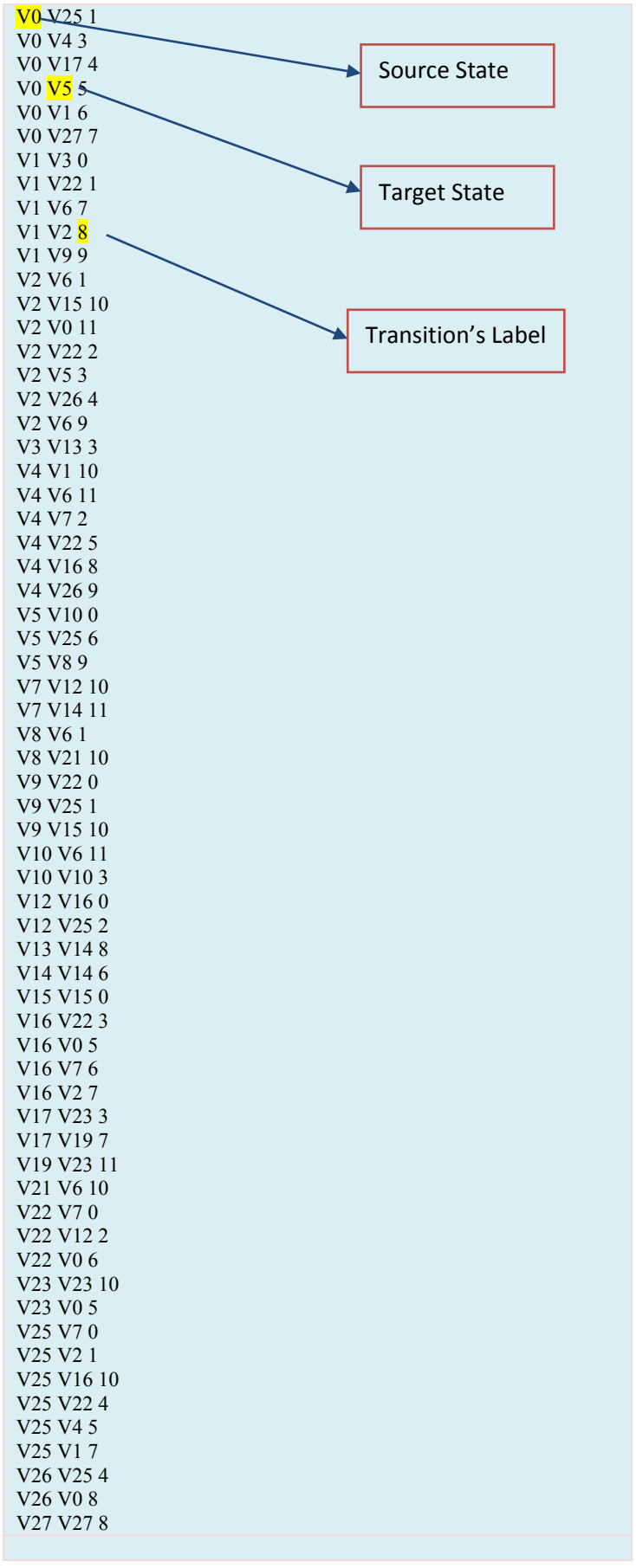
```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Wed Jul 13 12:41:07 BST
2011'>
  <XMI.header>    <XMI.documentation>
      <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
      <XMI.exporterVersion>0.32.1(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan 2010) $
</XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000865'
      name = 'untitledModel' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:StateMachine xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000866'
          isSpecification = 'false'>
          <UML:StateMachine.top>
            <UML:CompositeState xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000867'
              name = 'top' isSpecification = 'false' isConcurrent = 'false'>
              <UML:CompositeState.subvertex>
                <UML:Pseudostate xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000868'
                  name = 'initial' isSpecification = 'false' kind = 'initial'>
                  <UML:StateVertex.outgoing>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086B'/>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086C'/>
                  </UML:StateVertex.outgoing>
                </UML:Pseudostate>
                <UML:SimpleState xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000869'
                  name = 'S1' isSpecification = 'false'>
                  <UML:StateVertex.outgoing>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000870'/>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000873'/>
                  </UML:StateVertex.outgoing>
                  <UML:StateVertex.incoming>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086B'/>
                  </UML:StateVertex.incoming>
                </UML:SimpleState>
                <UML:SimpleState xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086A'
                  name = 'S2' isSpecification = 'false'>
                  <UML:StateVertex.outgoing>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087E'/>
                  </UML:StateVertex.outgoing>
                  <UML:StateVertex.incoming>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086C'/>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000876'/>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000879'/>
                  </UML:StateVertex.incoming>
                </UML:SimpleState>
                <UML:SimpleState xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086F'
                  name = 'S3' isSpecification = 'false'>
                  <UML:StateVertex.outgoing>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000875'/>
                  </UML:StateVertex.outgoing>
                  <UML:StateVertex.incoming>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000870'/>
                  </UML:StateVertex.incoming>
                </UML:SimpleState>
                <UML:SimpleState xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000872'
                  name = 'S4' isSpecification = 'false'>
                  <UML:StateVertex.outgoing>
                    <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000876'/>
                  </UML:StateVertex.outgoing>
```

```
            <UML:StateVertex.incoming>
              <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000873'/>
              <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000875'/>
              <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087A'/>
            </UML:StateVertex.incoming>
          </UML:SimpleState>
          <UML:SimpleState xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000878'
            name = 'S5' isSpecification = 'false'>
            <UML:StateVertex.outgoing>
              <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000879'/>
              <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087A'/>
            </UML:StateVertex.outgoing>
            <UML:StateVertex.incoming>
              <UML:Transition xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087E'/>
            </UML:StateVertex.incoming>
          </UML:SimpleState>
        </UML:CompositeState.subvertex>
      </UML:CompositeState>
    </UML:StateMachine.top>
    <UML:StateMachine.transitions>
      <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086B'
        isSpecification = 'false'>
        <UML:Transition.trigger>
          <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086D'/>
        </UML:Transition.trigger>
        <UML:Transition.source>
          <UML:Pseudostate xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000868'/>
        </UML:Transition.source>
        <UML:Transition.target>
          <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000869'/>
        </UML:Transition.target>
      </UML:Transition>
      <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086C'
        isSpecification = 'false'>
        <UML:Transition.trigger>
          <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086E'/>
        </UML:Transition.trigger>
        <UML:Transition.source>
          <UML:Pseudostate xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000868'/>
        </UML:Transition.source>
        <UML:Transition.target>
          <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086A'/>
        </UML:Transition.target>
      </UML:Transition>
      <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000870'
        isSpecification = 'false'>
        <UML:Transition.trigger>
          <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000871'/>
        </UML:Transition.trigger>
        <UML:Transition.source>
          <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000869'/>
        </UML:Transition.source>
        <UML:Transition.target>
          <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086F'/>
        </UML:Transition.target>
      </UML:Transition>
      <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000873'
        isSpecification = 'false'>
        <UML:Transition.trigger>
          <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000874'/>
        </UML:Transition.trigger>
        <UML:Transition.source>
          <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000869'/>
        </UML:Transition.source>
        <UML:Transition.target>
          <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000872'/>
        </UML:Transition.target>
```

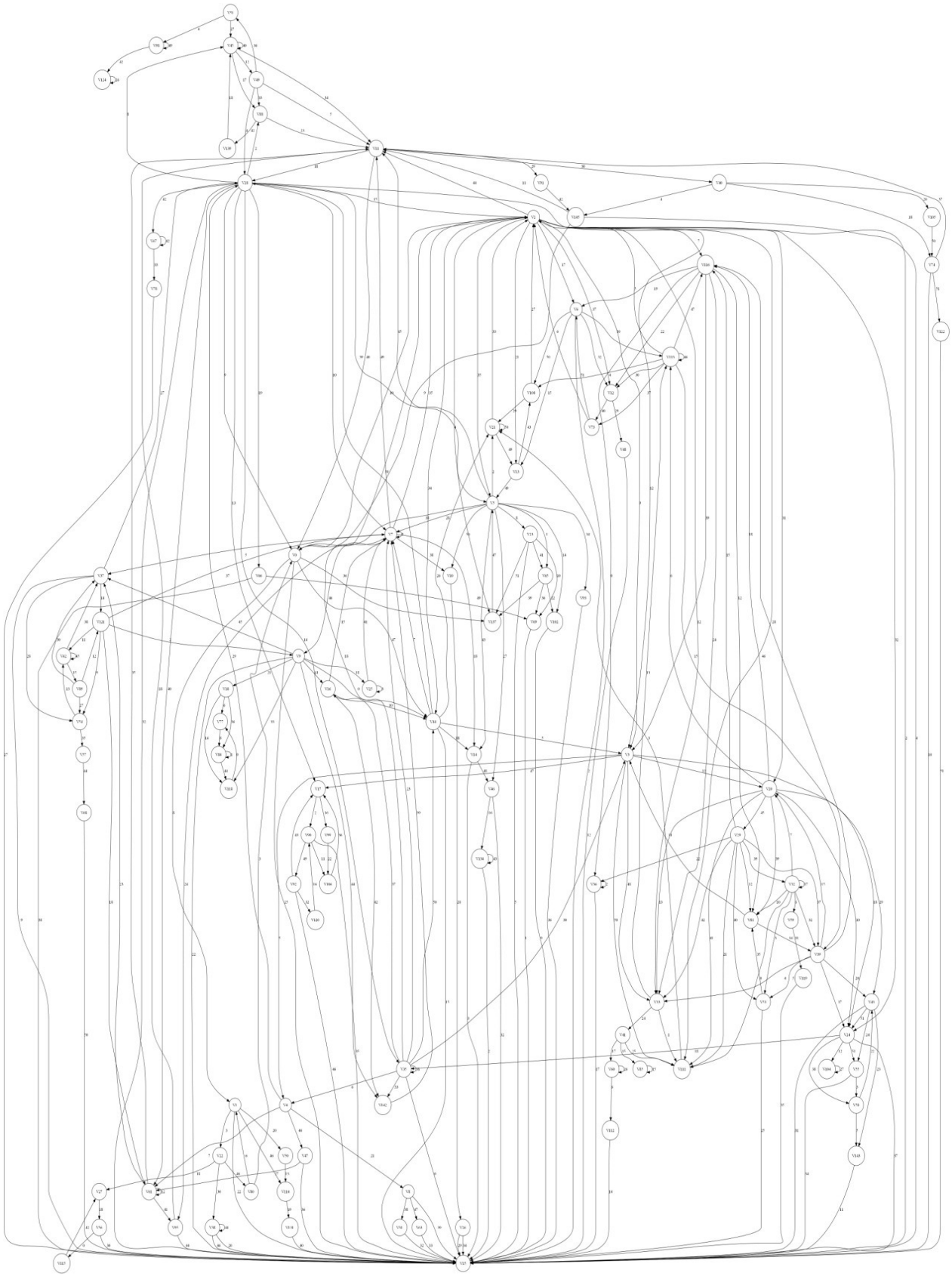
```
        </UML:Transition>
        <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000875'
          isSpecification = 'false'>
          <UML:Transition.trigger>
            <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000874'/>
          </UML:Transition.trigger>
          <UML:Transition.source>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086F'/>
          </UML:Transition.source>
          <UML:Transition.target>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000872'/>
          </UML:Transition.target>
        </UML:Transition>
        <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000876'
          isSpecification = 'false'>
          <UML:Transition.trigger>
            <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000877'/>
          </UML:Transition.trigger>
          <UML:Transition.source>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000872'/>
          </UML:Transition.source>
          <UML:Transition.target>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086A'/>
          </UML:Transition.target>
        </UML:Transition>
        <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000879'
          isSpecification = 'false'>
          <UML:Transition.trigger>
            <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087D'/>
          </UML:Transition.trigger>
          <UML:Transition.source>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000878'/>
          </UML:Transition.source>
          <UML:Transition.target>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086A'/>
          </UML:Transition.target>
        </UML:Transition>
        <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087A'
          isSpecification = 'false'>
          <UML:Transition.trigger>
            <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087B'/>
          </UML:Transition.trigger>
          <UML:Transition.source>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000878'/>
          </UML:Transition.source>
          <UML:Transition.target>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000872'/>
          </UML:Transition.target>
        </UML:Transition>
        <UML:Transition xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087E'
          isSpecification = 'false'>
          <UML:Transition.trigger>
            <UML:SignalEvent xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000874'/>
          </UML:Transition.trigger>
          <UML:Transition.source>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086A'/>
          </UML:Transition.source>
          <UML:Transition.target>
            <UML:SimpleState xmi.idref = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000878'/>
          </UML:Transition.target>
        </UML:Transition>
      </UML:StateMachine.transitions>
    </UML:StateMachine>
    <UML:SignalEvent xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086D'
      name = 't1' isSpecification = 'false'/>
    <UML:SignalEvent xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000086E'
      name = 't2' isSpecification = 'false'/>
```

```
      <UML:SignalEvent xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000871'
        name = 't3' isSpecification = 'false'/>
      <UML:SignalEvent xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000874'
        name = 't7' isSpecification = 'false'/>
      <UML:SignalEvent xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:0000000000000877'
        name = 't5' isSpecification = 'false'/>
      <UML:SignalEvent xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087B'
        name = 't6' isSpecification = 'false'/>
      <UML:SignalEvent xmi.id = '-87--2--81--61-11a3b9cb:13123489183:-8000:000000000000087D'
        name = 't9' isSpecification = 'false'/>
    </UML:Namespace.ownedElement>
  </UML:Model>
 </XMI.content>
</XMI>
```

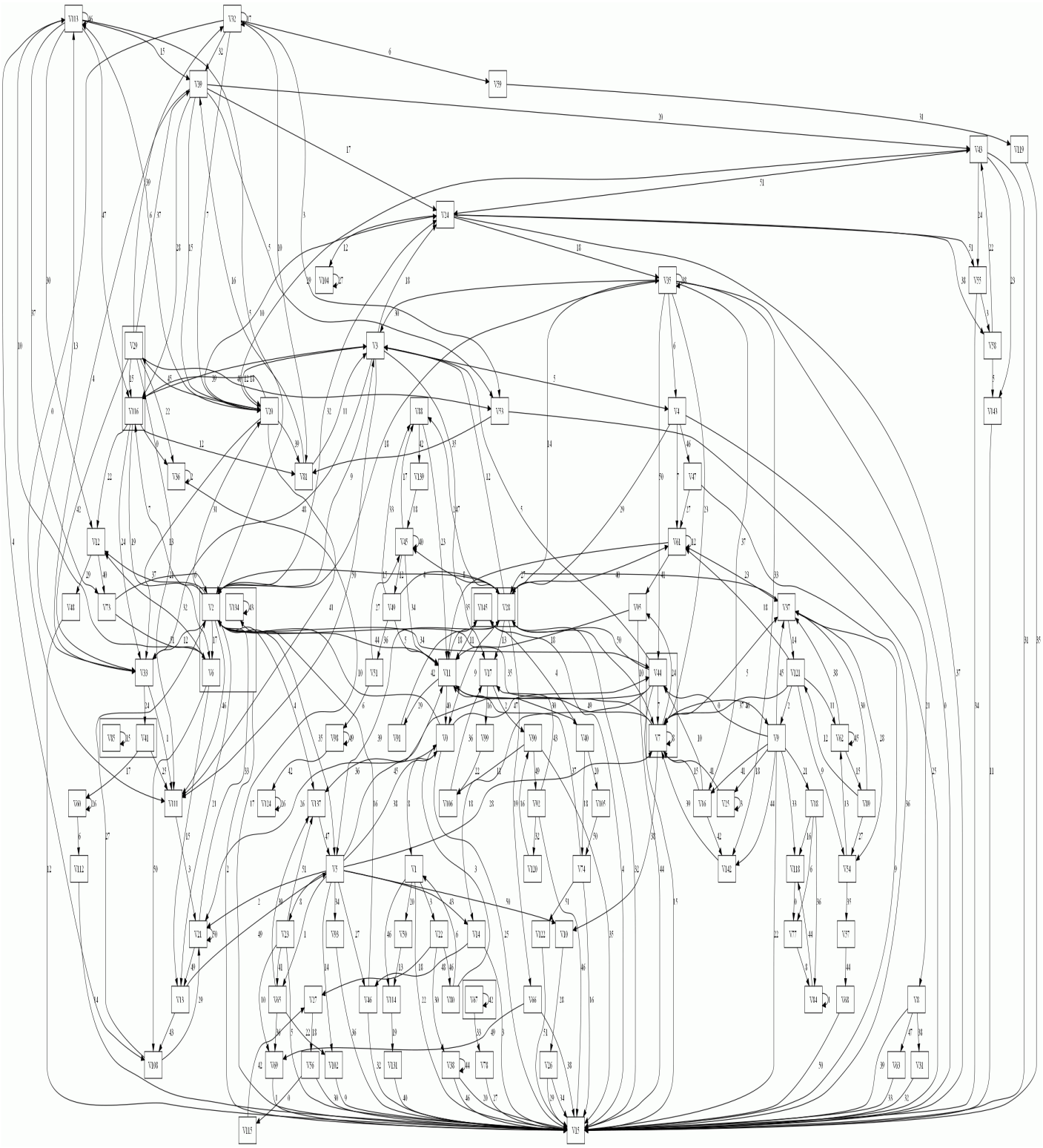
FSM.x_ Format


V0 V25 1
V0 V4 3
V0 V17 4
V0 V5 5
V0 V1 6
V0 V27 7
V1 V3 0
V1 V22 1
V1 V6 7
V1 V2 8
V1 V9 9
V2 V6 1
V2 V15 10
V2 V0 11
V2 V22 2
V2 V5 3
V2 V26 4
V2 V6 9
V3 V13 3
V4 V1 10
V4 V6 11
V4 V7 2
V4 V22 5
V4 V16 8
V4 V26 9
V5 V10 0
V5 V25 6
V5 V8 9
V7 V12 10
V7 V14 11
V8 V6 1
V8 V21 10
V9 V22 0
V9 V25 1
V9 V15 10
V10 V6 11
V10 V10 3
V12 V16 0
V12 V25 2
V13 V14 8
V14 V14 6
V15 V15 0
V16 V22 3
V16 V0 5
V16 V7 6
V16 V2 7
V17 V23 3
V17 V19 7
V19 V23 11
V21 V6 10
V22 V7 0
V22 V12 2
V22 V0 6
V23 V23 10
V23 V0 5
V25 V7 0
V25 V2 1
V25 V16 10
V25 V22 4
V25 V4 5
V25 V1 7
V26 V25 4
V26 V0 8
V27 V27 8
Source State
Target State
Transition's Label

Appendix C: Complex FSM Generated using the tool and visualizing it using GraphViz



Appendix D: the generated statechart using the tool and visualizing it using GraphViz



Project Plan

Tasks	20 May – 30 May	31 May – 19 Jun	20 Jun – 27 Jun	28 June – 13 Jul	14 Jul – 15 Aug	16 Aug – 29 Aug	30 Aug - 31 Aug	01-Sep
Design								
Design (in details)								
Implementation								
load xmi from argoUML								
Produce DOT								
OR-state constructor								
AND-state constructor								
complete statecharts constructor								
decision-making process for states								
Testing								
Testing OR-state function								
Testing AND-state function								
Testing and evaluation complexities								
complete testing								
Report								
writing design part in report								
writing OR-state part in report								
writing AND-state Part in report								
complete writing report								
Submit MSC Report								

The modified interface of the Tool