Constructive Formal Conversion of Moore Machine to Deterministic Finite Automata

Shagufta Riaz* and Nazir Ahmad Zafar**

*Faculty of Information Technology, University of Central Punjab
Plot No. 31-A, Main Gulberg, Lahore, PAKISTAN
Tele: +92-42-5755314-7; Fax: +92-42-5710881

**Department of Computer and Information Sciences
Pakistan Institute of Engineering and Applied Sciences
Nilore, Islamabad, PAKISTAN
Tele: +92-51-2207381-2; Fax: +92-51-2208070

Abstract—The complexity of a system in terms of its state space and functional aspects enforces to choose most prolific and rigorous approaches to sustain the development of the software systems. Finite automaton is a formal and natural medium to describe the static and dynamic behavior of a complex system. Z notation is an ideal specification language having inherent capability to describe state space of a system and then defining operations over it. This paper deals with the integration of finite automata and Z which can enhance the modeling power for a complex system. Although Moore machine cannot be used for language recognition but its conversion into deterministic finite automata makes it possible to pretend as a language recognizer. Here a formal conversion of Moore machine to deterministic finite automata based on a string as well as a language is proposed using Z. Formal specification of the linkage between Z and automata is analyzed and validated using Z/EVES tool.

Keywords—Deterministic finite automata, Moore machine, Integration of approaches, Z, Validation

I. INTRODUCTION

The development process of any system begins with the notion of gathering requirements and design specification. And this is a preliminary phase where the existence of even slightly vague concepts and design issues create ripple affect till the end of the software system development lifecycle. So, there is a need to recognize and clarify or reduce all ambiguities at each stage of the software development. And one of the major challenges faced by high confidence systems is that reliable high performance or scalable systems require correct and timely integration of specifications and designs at all levels of detail. This leads to introduce high assurance techniques or tools. The formal methods are mathematical modelling techniques which can be applied for specification of computer software and hardware systems. The formal methods can benefit at any stage of the software development lifecycle. These techniques also provide basis for requirements capturing and analysis as well as upgrade the level of integrity of the system. Further, a robust architecture and the provable design can be proposed using these techniques. The significance of formal methods can also be viewed as: there are a number of projects where formal methods have been used as an integral part of their development process [1]. Because of the rigorous proof of techniques, formal methods are playing a vital role in software engineering education and industry [2].

The specification written using formal techniques yields a significant advantage over natural and notational languages. This is because the resultant specification is unambiguous, can be checked mechanically, and is open to proof [3]. The formal specifications used in software documentation and debugging by presenting real-life scenario show the worthy impact over informal specification techniques.

Z notation [4] is a formal specification language. It is a model oriented approach, which is based on set theory and first order predicate logic. It is used for specifying the abstract data types and sequential programs. Z notation can also be used to define state space of a system and then defining operations over it.

Automata theory is a branch of the theory of control systems based on the study of mathematical models. The theory of automata was introduced in the middle of 20th century in
connection with finite automata. The finite automata, with finite memory, are mathematical models of nervous systems and electronic computers which can process the discrete information. A system may be considered to be modelled as an abstract state and a sequence of operations. Automata are very helpful in capturing the behaviour of a system.

The design of a complex system, not only requires the techniques for capturing behavior of the system but also its functionality. Functions over any of the systems can be decomposed in terms of operations and constraints, and hence Z notation is an ideal application of capturing functionalities of a system. Control over a system can be viewed in terms of visual flows in between the system’s functions. Automata theory is very powerful in modeling control over the system’s behavior. Consequently, it requires an integration of automata and Z to increase modeling power for a complex system, which is one of the objectives of this research.

A finite automaton with certain output is referred as Moore machine. Moore machine similar to finite automata processes discrete information. And in addition to it produces certain output if required. Although Moore machine does not define any language but it can be converted into deterministic finite automata to pretend as a language accepter. In this paper, we present a conversion mechanism using Z notation. To achieve the objective of integration of automata and Z notation, we develop a formal semantics of transformation for a subset of automata to Z focusing on conversion of Moore machine to deterministic finite automata. Z/Eves tool is used to verify and analyze the syntactic and semantic analysis of the Z specification code.

The major objective of this paper is to (i) achieve the integration of the automata and formal methods to enhance the power of deterministic finite automata by using the inherent characteristic of Moore machine and (ii) provide a syntactic and semantic well organized specification using Z notation to apply the formal techniques for the system development.

In section 2, related work is discussed. This section also contains some of the applications of deterministic finite automata and Moore machine. In section 3, an introduction to Formal methods is given. In section 4, finite automata and Moore machines are discussed. Integration of automata and Z notation is given in section 5. Finally, concluding remarks are given in section 6.

II. RELATED WORK

There exists a lot of research work on integration of approaches [5], [6], [7], [8], [9], [10], [11] but still there is a need for identifying and proving the relationships among the formal and informal techniques. The work of J.S. Dong [12], [13] is closely related to ours in which they deal with the integration of Object-Z and Timed Automata. These papers present an effective combination of the two techniques with novel composition and communication mechanisms. In [14], [5], C.T. Chou and E.A. Boiten proposed a constructive formalization of some important concepts of automata using Nuprl. In [15], by R. Bussow and W. Grieskamp, the integration of formal methods exemplifies the objective of integration of approaches. The author established a combination of Z with state charts in which operations were defined over schemas in linking with state charts. In [16], [17], the authors provide a relationship between Z notation and Petri Nets. In another study, the authors presented an integration of formal methods to apply the advantages of a more formal approach to the development of automotive software in an industrial setting [18]. Brendan Mahony, Jin Song Dong introduced an integration of Object-Z and Timed CSP, known as TCOZ. This integration was particularly suited for specifying complex systems whose components have their own thread of control [19].

Finite automata have various applications in computer science. Compiler construction, software engineering and security, interactive games, natural-language processing, artificial intelligence, bimolecular science and DNA computing [20] are some of its applications areas. Pattern matching in bioinformatics, modeling control behavior, modeling of finite state systems and defining a regular set of finite words are some of other important applications of finite automata. Evolution of finite automata has resulted in several modern applications, e.g. spelling checkers and advisers, multilanguage dictionaries, thesauri, minimal perfect hashing and text compression [21], optimization of logic based programs and specification and verification of protocols [22]. The human computer interaction is another interesting research area where abstract machines can be used.

There are various applications of Moore machine. One of the applications of it is modeling digital electronic systems which are designed as clocked sequential systems (CSS). Microprocessors, digital clocks, mobile phones, cordless telephones, electronic calculators are some of the examples of CSS. Clocked sequential system is a restricted form of Moore machine where the state changes only when the global clock signal changes.

III. INTRODUCTION TO FORMAL METHODS

Formal methods are approaches, based on the use of mathematical techniques and notations, for describing and analyzing properties of software systems [23]. That is, descriptions of a system are written using notations which are mathematical expressions rather than informal notations. These mathematical notations are based on discrete mathematics and algebraic concepts, such as logic, set theory and graph theory. There are several ways in which formal methods may be classified. One frequently-made distinction is between model oriented and property oriented methods [23]. Model oriented methods are used to construct a model of a system’s behavior [4]. State transition diagrams, for example, are used to model the behavior of a system as a set of states and transitions between them. Property oriented methods are used to describe software in terms of a set of properties, or constraints, that must be satisfied.

Formal methods are used to improve the quality of complex computer software by means of documenting system specifications in a precise and structured manner. Z is one of the most popular specification languages in formal method.
The Z notation [24] is a model oriented approach, which is based on set theory and first order predicate logic [25]. It is also used for specifying the behavior of abstract data types and sequential programs. Z notation [26], [27] is used in our work for specification because it describes a state space of a system and a set of operations that may be performed on it.

IV. FINITE AUTOMATA AND MOORE MACHINE

Finite automata are abstract models of machines, based on mathematical techniques and notations which can be represented using diagrams. These models can receive an input and perform computation to generate a certain output by going through a sequence of transformations. The acceptance of the input depends upon the criterion of the finite automata that is called its transition function. The transition function determines the next state depending on the current state and input alphabet. Finite automaton consists of a finite number of states, one of which is designated as the initial state, and some or none of which are final states. The transition function takes a state and an input alphabet and produces a state as an output. Deterministic and nondeterministic finite automata are two most frequently used types of finite automata. Deterministic finite automaton is a simple finite automaton and takes care of exactly one transition against each possible input alphabet.

The transition function of deterministic finite automaton undergoes some changes to create a Moore machine. Deterministic finite automaton can select or reject a certain sequence of input alphabets depending on the transition function. But omission of the accepting restriction and an additional feature of printing an output character against each state, converts deterministic finite automaton into a Moore machine. The output of Moore machine is determined by the current state alone and does not depend directly on the input alphabet. Moore machine does not accept any language because every possible input string creates an output string and there is no such thing as a final state. The processing is terminated when the last input letter is read and the last output character is printed [28]. Unlike a deterministic finite automaton, Moore machine gets complexity introducing the criterion for output selection against each state of the machine. However, a Moore machine can be converted into a deterministic finite automaton after a certain modification. For this purpose, only the transition function will be changed and a final state will be identified after running a string or a collection of strings on the Moore machine. The state where the string is ended in the Moore machine is considered as a final state in the deterministic finite automaton. The formal conversion of Moore to deterministic finite automata is given in the subsequent section.

V. CONSTRUCTION OF FORMAL CONVERSION

In this section, a formal conversion is given from Moore machine to deterministic finite automata using Z notation. The definitions used, in this paper, are based on some well known books on Automata and Computation Theory [10], [22]. Following is a sequence of activities to construct DFA from Moore machine.

- Designing Deterministic Finite Automata (DFA)
- String Decider
- Language Accepter
- Designing Moore Machine
- Printing Output
- Conversion from Moore Machine to DFA.

A. Design of Deterministic Finite Automata

We start with the definition of deterministic finite automata. A deterministic finite automaton is a five tuple \((Q, \Sigma, \delta, q_0, F)\), where

1. \(Q\) is a finite non-empty set of states
2. \(\Sigma\) is a finite set of alphabets
3. \(\delta: Q \times \Sigma \rightarrow Q\) is the transition function which takes a state \(q\) and an alphabet \(a\) as inputs and produces the same state or a new state as output.
4. \(q_0\) is the initial state
5. \(F\) is a finite set of final states.

The above 5-tuple DFA = \((Q, \Sigma, \delta, q_0, F)\) will be a deterministic finite automata if for each state \(s_1\) and for every alphabet \(a\), there is a unique state \(s_2\) such that \(\delta(s_1, a) = s_2\). To formalize the DFA in Z notation, \(Q\) and \(\Sigma\) are represented as \(S\) and \(I\) respectively.

\([S, I]\)

In modeling using sets in Z notation, we do not impose any restriction upon the number of elements and a high level of abstraction is supposed. Further, we do not insist upon any effective procedure for deciding whether an arbitrary element is a member of the given collection or not. As a consequent, our \(S\) and \(I\) are sets over which we cannot define some operations, e.g., cardinality to know the number of elements in these sets.

To describe a set of states, for a particular finite automaton, a new variable \(states\) is introduced. Since a given state \(s\) is of type \(S\), therefore \(states\) variable must be a type of power set of \(S\). Similarly, to describe a set of alphabets, for the same finite automaton, a variable \(inputs\) is used. Since a given alphabet \(a\) is of type \(I\) therefore \(inputs\) must be of type of power set of \(I\). As we know that \(\delta\) relation is a function because for each input \((s_1, a)\), where \(s_1\) is a state and \(a\) is an alphabet, there must be a unique output \(s_2\) of type \(S\) which is image of \((s_1, a)\) under the transition function \(\delta\). Hence we can declare \(\delta\) as \(transFunc: S \times I \rightarrow S\). It is obvious that the initial state \(startState\) is of type \(S\). Now our last one construct \(F\) set of final states, is required to be defined. The set of final states for our DFA, under construction, is represented by \(finalStates\) and is of type of power set of \(S\).

For a moment, we have used formal mathematical language of Z notation which is used to describe various objects. It is to be mentioned here that the same language can be used to define the relationships in between these objects.
This relationship will be used in terms of constraints after composing these objects. The schema structure is used here for composition of these objects because it is very powerful at abstract level of specification and it helps in describing a good specification approach. All of the above components of DFA are encapsulated and put in the schema named as DFA.

**DFA**

states: P S
inputs: P I
startState: S
finalStates: P S
transFunc: S x I -> S

startState \in states
finalStates \subseteq states
\forall s1: S; a: I \ | s1 \in states \land a \in inputs
\exists s2: S \ | s2 \in states \land (s1, a, s2) \in transFunc
\forall s: S; a: I \ | (s, a) \in dom transFunc \land s \in states \land a \in inputs
\forall s: S \ | s \in ran transFunc \land s \in states

**B. String Accepter**

After designing DFA, we require to check whether a given string is accepted by it. For this purpose, we need to verify that the string must be based on the set of alphabets of DFA called strings.

A string accepter is designed which takes inputString and DFA as inputs and checks whether the given string is accepted by the DFA or not. A formal definition of a DFA is already given in [29]. Now we do the same but for computational power of it which we have called a string accepter.

Let DFA = (Q, \Sigma, \delta, q0, F) be a deterministic finite automata and let w = w1w2...wn be a string for each wi where i = 1, 2, ...n. Then we say that DFA accepts the string s if there exists a sequence of states s0, s1, ..., sn in Q satisfying the following conditions:

1. s0 = q0
2. \delta(si, wi+1) = si+1, \forall i = 0, 1, ..., n-1.
3. sn \in F

The first condition states that deterministic finite automaton starts from q0 which is a start state. The second condition means that the DFA goes from state to state according to the definition of transition function \delta. Finally, in the third condition, it is stated that the DFA accepts the string s if our machine ends up in an accepting state.

Now coming back to the formal definition of string accepter, we have three inputs, DFA, inputString? and strings. The symbol \Xi shows that the machine will not be changed but it will only be used. And the question mark ? after inputString means that inputString is given as input. The string accepter is represented as DFAsyncAccepter as a schema. The inputs are given in first part of the schema and constraints are defined in the second part of it.

**DFAsyncAccepter**

\Xi DFA
inputString?: seq I
strings: P (seq I)

Invariants:

1. The input string inputString? must be generated from the alphabets of DFA.
2. For a given string inputString? of length n, there must be a sequence of states of length, at most, n+1 in which the first element of this sequence is the initial state of DFA. The last element in this sequence must be an element of the set of final states of DFA. And, for every state si excluding final state, there must be an alphabet wi+1 such that:
   \[
   \text{transFunc} (s_i, w_{i+1}) = s_{i+1}, \forall i = 0, 1, \ldots, n-1.
   \]

**C. Language Accepter**

A language accepter is designed which takes a language and a DFA as inputs and checks if the language is accepted by it. In the language accepter, we have reused the string accepter designed above and extended over the set of all strings of a given language. Mathematically, we can define a language accepter as:

Language Accepter =

\{<DFA, language> \ | \forall w \in language, DFA accepts w\}.

Again we have three inputs for the schema given below, DFA, language? and strings. The symbols \Xi and ? are already defined. The language accepter is represented by LanguageAccepter as a schema. The formal specification with the constraints over it is given below:
### Language Acceptor

**DFA**  

<table>
<thead>
<tr>
<th>Language:</th>
<th>P (seq I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strings:</td>
<td>P (seq I)</td>
</tr>
</tbody>
</table>

\[
\forall w: \text{Strings} \mid w \in \text{language} \\
\forall w: w \in \text{strings}
\]

- \( \forall i: \text{N} \mid i \in 1 \ldots \#w \mid (i = \#w)? \)
- \( \Rightarrow (s_1, s_2): S \mid s_1 = \text{startState} \land s_2 = \text{states} \)
- \( (s_1, w, i), (s_2) \in \text{transFunc}) \land (i > 1 \land i < \#w)? \)
- \( \Rightarrow (s_1, s_2): S \mid s_1 \in \text{states} \land s_2 \in \text{states} \)
- \( (s_1, w, i), (s_2) \in \text{transFunc}) \land (i = \#w)? \)
- \( \Rightarrow (s_1, s_2): S \mid s_1 \in \text{states} \land s_2 = \text{finalStates} \)
- \( (s_1, w, i), (s_2) \in \text{transFunc}) \)

**Invariants:**

1. We must be able to generate every string of the language using alphabets of finite automata.
2. For every string \( w \) of length \( n \) of the language, there must be a sequence of states of length, at most, \( n+1 \) in which (i) first element of the sequence is the initial state of finite automata, (ii) The last element in the sequence must belong to the set of final states of finite automata and (iii) for every state \( s \), there must an alphabet \( w_{i+1} \) such that \( \delta(s, w_{i+1}) = s_{i+1} \), \( \forall i = 0, 1, \ldots, n-1 \).

### Design of Moore Machine

Now we give the definition of Moore machine before giving its formal conversion procedure to DFA. Moore machine consists of six components \((Q, q_0, \Sigma, \Gamma, \delta, O)\), where (i) \( Q \) is a finite non-empty set of states, (ii) \( q_0 \) is the initial state, (iii) \( \Sigma \) is a finite set of alphabets used for input, (iv) \( \Gamma \) is a finite set of characters used for output, (v) \( \delta: Q \times I \rightarrow Q \times O \) is the transition function which takes a state \( s \) and an alphabet \( a \) as inputs and produces a state and a character as output, (vi) \( O \) is the set of output characters. The sets \( Q, \Sigma, \Gamma \) are already declared as \( S, I, O \) respectively. Now a new variable \( O \) is added to represent output symbols. The Moore machine is defined as a schema and represented as Moore. The invariants over it are defined after giving formal description of it. \([S, I, O] \)

**Moore**

| states: | P S |
| inputs: | P I |
| outputs: | P O |
| q0: | S |
| transition: | S x I -> S x O |

\( q_0 \in \text{states} \)

\( \forall s: S \mid i \mid s \in \text{states} \land i \in \text{inputs} \)

- \( \forall s: S \mid o \mid s \in \text{states} \land o \in \text{outputs} \)
- \( (s, i, (s_1, o)) \in \text{trans} \)

\( \forall s: S \mid i \mid (s, i) \in \text{dom} \text{trans} \land s \in \text{states} \land i \in \text{inputs} \)

\( \forall s: S \mid o \mid (s, o) \in \text{ran} \text{trans} \land s \in \text{states} \land o \in \text{outputs} \)

**Invariants:**

1. The initial state \( q_0 \) must be an element of states.
2. For each input \((s_1, i)\), where \( s_1 \) is a state and \( i \) is an input alphabet, there must be a unique ordered pair \((s_2, o)\), where \( s_2 \) is a state and \( o \) is an output character such that: \( \text{transition} (s_1, a) = s_2 \).
3. For each element \((s, i)\) in the domain of \text{transition}-function, the state \( s \) must be in set of states and \( i \) must be in set of input alphabets.
4. For each element \((s, o)\) in the range of \text{transition}-function, the state \( s \) must be in set of states and \( i \) must be in set of output characters.

### E. Printing Output from Moore Machine

As we know that Moore machine is a special type of finite automata which can be used for generation of output as well. The schema \text{MooreOutput} showing a relationship for accepting input and generating output is given below.

**MooreOutput**

\( x^{\text{DFA}} \)

| input? | seq I |
| output! | seq O |
| o!: | O |

\( \exists \text{Moore} \)

- \( \text{ran input?} \subseteq \text{inputs} \)
- \( \text{ran output!} \subseteq \text{outputs} \)
- \# output! > 1
- \# output! = \# input? + 1
- \( \forall i: \text{N} \mid i \in \text{dom input?} \)
  - \( \exists s_1, s_2: S \mid s_1 \in \text{states} \land s_2 \in \text{states} \)
  - \( ((s_1, \text{input? } i), (s_2, \text{output}! (i + 1))) \in \text{transition} \)

**Invariants:**

1. The input \text{input?} string must be based on set of input alphabets.
2. The output \text{output!} string is based on set of output characters.
3. The number of \text{output!} characters are at least one.
4. The first character of the \text{output!} string is the character printed against the initial state of the Moore machine.
5. The number of output characters printed must be one more than the number of input alphabets.
6. For each integer \( i \) in domain of \text{input} sequence, there must be two states \( s_1 \) and \( s_2 \) such that: \((s_1, \text{input? } i), (s_2, \text{output}! (i + 1)) \in \text{transition} \).

### F. Conversion From Moore to DFA

The process of conversion of Moore machine to deterministic finite automata encompasses the steps involved in the mapping of various constituents of both machines. As there is no concept of final state in Moore machine so it is required to give string of input alphabets to be run on Moore machine and the last state generated is treated as the only final state of the deterministic finite automata. Similarly, on
providing a collection of strings of input alphabets to run on Moore machine, a set of final states is generated. Now, the specification of this conversion based on a single input string and a collection of input strings is explained in the subsequent subsections.

**Conversion Based on a String**

Here a formal conversion from Moore machine to DFA, based on a single string of input alphabets, is given in terms of a schema MooreToDFAString. A Moore machine and a string of input alphabets are given as input to the schema. The relationship between constituents is demonstrated in terms of invariants as given below.

\[
\text{MooreToDFAStrings}
\]

\[
\exists \text{Moore}
\]

\[
\text{statesFA}: \mathbb{F} S
\]

\[
\text{inputsFA}: \mathbb{F} I
\]

\[
\text{prints!}: \text{seq } O
\]

\[
\text{startStateFA}: S
\]

\[
\text{finalStatesFA}: \mathbb{F} S
\]

\[
\text{transFuncFA}: \mathbb{I} \times I \rightarrow S
\]

\[
\text{string?}: \text{seq } I
\]

\[
\text{statesFA} = \text{states}
\]

\[
\text{inputsFA} = \text{inputs}
\]

\[
\text{startStateFA} = q0
\]

\[
\text{ran string?} \subseteq \text{inputs}
\]

\[
\text{# prints!} = \# \text{string?} + 1
\]

\[
\forall i: \mathbb{N} \cup i \in \text{dom string?}
\]

\[
\exists s1, s2: S \mid s1 \in \text{states} \land s2 \in \text{states}
\]

\[
\text{transition } \cdot \text{ states? } \subseteq \text{states?}
\]

\[
\forall s1, s2: S, i: I, o: O \mid s1 \in \text{states} \land s2 \in \text{states}
\]

\[
\text{transFuncFA} = (s1, i, o) \in \text{output!}
\]

\[
\text{transFuncFA} = (s1, i, o) \in \text{transFuncFA}
\]

\[
\text{Invariants:}
\]

1. The total states in Moore machine and resultant DFA are equal.
2. The input alphabets of resultant DFA are same as in Moore machine.
3. The start state of Moore machine will be supposed as the start state of the DFA.
4. The input string must be based on set of input alphabets of Moore machine.
5. The number of output symbols printed by Moore machine must be one more than the number of input alphabets accepted by it.
6. For each integer \( i \) in the domain of input sequence, there must be two states \( s1 \) and \( s2 \) such that:

\[
(s1, \text{input! } i, (s2, \text{output! } (i + 1))) \in \text{transition}
\]

7. The set of final states in DFA is a set of the states of Moore machine where input string is ended after running on it.
8. For each state and input alphabet, the transition function of DFA behaves similar to the transition function of Moore machine ignoring the output characters printed in case of Moore machine.

**Conversion Based on a Collection of Strings**

Now we present a formal conversion of Moore machine to DFA based on a collection of strings. A collection can be a set or sequence. As per requirement it is more appropriate to use sequence as a collection of strings of input alphabets. The Z specification of the conversion can be given as follows and is explained in terms of its invariants.

\[
\text{MooreToDFALanguage}
\]

\[
\exists \text{Moore}
\]

\[
\text{statesFA} = \text{states}
\]

\[
\text{inputsFA} = \text{inputs}
\]

\[
\text{startStateFA} = q0
\]

\[
\text{ran string? } \subseteq \text{inputs}
\]

\[
\text{# strings!} = \# \text{string?} + 1
\]

\[
\forall i: \mathbb{N} \cup i \in \text{dom string?}
\]

\[
\exists s1, s2: S \mid s1 \in \text{states} \land s2 \in \text{states}
\]

\[
\text{transition } \cdot \text{ states? } \subseteq \text{states?}
\]

\[
\forall s1, s2: S, i: I, o: O \mid s1 \in \text{states} \land s2 \in \text{states}
\]

\[
\text{transFuncFA} = (s1, i, o) \in \text{output!}
\]

\[
\text{transFuncFA} = (s1, i, o) \in \text{transFuncFA}
\]
Invariants:
1. The total states in Moore machine and resultant DFA are equal.
2. The input alphabets of resultant DFA are same as in Moore machine.
3. The start state of Moore machine will be supposed as start state of the DFA.
4. All strings in the sequence of input strings must be based on the alphabets of the Moore machine.
5. The number of output symbols in each output string of the output sequence prints!, printed by Moore machine, must be one more than the number of input alphabets of the corresponding input string of the input sequence language?.
6. For each input alphabet of language?, there must be two states s1 and s2 such that : \((s1, \text{language?} \cdot i, j), (s2, \text{prints!} \cdot (j + 1))) \in \text{transition}\).
7. The set of final states in DFA is a set of states of Moore machine where each input string ends after running on it.
8. For each state and input alphabet, the transition function of DFA behaves similar to the transition function of Moore machine ignoring the output characters.

VI. Conclusion
In this paper, a formal design of deterministic finite automata and string accepter was presented. Formal description of Moore machine and its output printing procedure was proposed. Finally, a procedure of formal conversion from Moore machine to deterministic finite automata was investigated using a single input string as well as a collection of input strings also referred as a language. We observed that the process of constructive formal conversion of Moore machine to deterministic finite automata using Z notation yields an integration of these approaches. The Z/Eves tool was used to analyze and validate the specification.

We also observed that finite automata is best suited in modeling behavior of a system while Z notation is an ideal specification language to be used describing state space and then defining operations over it. Based on this observation, we investigated and identified the linkage between Z notation and finite automata. We believe that this integration will enhance the modeling power for a complex system. We also believe that this combined approach can also be useful in development of an integrated and automated toolset.

REFERENCES