Formal transformation from NFA to Z notation by constructing union of regular languages

Nazir Ahmad Zafar, Nabeel Sabir and Amir Ali

***Abstract*—**Capturing functionalities and modeling control behavior are primary requirements in design and development of a complex system. Automata theory plays a vital role in modeling behavior while Z notation is an ideal specification language for describing state space of a system. Consequently, integration of automata and Z notation will be a useful tool facilitating and increasing modeling power for complex systems. Further, nondeterministic finite automata (NFA) may have different implementations and therefore it is needed to verify the transformation from diagrams to code. If we describe formal specification of a given nondeterministic finite automata before implementing then confidence over transformation can be increased. In this paper, we have combined NFA and Z and a linkage is established between these approaches. At this level of integration, we have given a formal procedure to transform NFA to Z. A string accepter is designed and then extended to the language accepter. Finally, NFA accepting union of two regular languages is constructed by describing formal specification of their relationships. The specification is analyzed and validated using Z/EVES tool.

***Keywords*—**Integration of approaches, mathematical modeling, automata theory, Z notation.

I. Introduction

AUTOMATA have various applications in computer science and engineering. The traditional applications of automata theory include pattern matching, compiler constructions, software verification, defining a regular set of finite words of a language and modeling control behavior of complex systems. The human computer interaction, optimization of logic based programs, specification and verification of protocols [1] and cryptography are some modern applications of automata theory.

The Z notation is a formal language which is based on set theory and first order predicate logic used for describing and modeling the systems at an abstract level of specification [2]. Usually we use Z for specifying the abstract data types and sequential programs by defining state space of a system and then operations in terms of relations over the state space.

The design of complex systems requires functionality as well as its control behavior. The functional aspects of a software system can be decomposed in terms of its operations and constraints over the data types, and hence Z is an ideal application of it. The control behavior of a system can be visualized in terms of flows between the system's functions where automata theory is very powerful in this aspect. Consequently, an integration of automata and Z is required increasing modeling power for complex systems which is one of the major objectives of this research. In this paper, we describe and develop a formal semantics of transformation for a subset of automata to Z focusing on non-determinism.

The both types, deterministic finite automata (DFA) and nondeterministic finite automata (NFA), are equivalent in power, in a sense, that if a language is recognized by a DFA it is also recognized by an NFA and vice versa. The NFAs are sometimes useful because constructing an NFA is much easier than constructing a DFA. This is due to the use of less mathematical work in building NFA as compared to DFA. Further, many important properties in automata can be established easily by using NFA. To prove, for example, that concatenation or union of two regular languages is regular using NFA is easier than using deterministic finite automata.

Nondeterministic automata are abstract mathematical models of systems which can also be represented using diagrams. These mathematical models can be used to perform computations on a given input by a sequence of configurations. An NFA reads the entire input and for each subset of input it moves to a new state until all input has been read. After reading the entire input, if machine is able to reach any of the accepting state then the given input is accepted. Due to the different implementations of NFA [3], [4], its space and execution time must be different. Further, we require preserving semantics of transformation from diagrams in NFA to implementation. Therefore, if we give a formalization of an NFA then this transformation can be facilitated.

At this level of integration, first we have constructed NFA using Z notation. Then a string accepter is formalized and extended to the language accepter. Finally, union of two regular languages is described by formalizing the specification of their relationship which is further analyzed and validated using Z/EVES tool-set [5]. The main objective of this paper is

Manuscript received March 14, 2009. This work was supported in part by the Center for Research in Computer Science, University of Central Punjab, Lahore, Pakistan.

N. A. Zafar is with the Faculty of Information Technology, University of Central Punjab, Lahore, on leave from Pakistan Institute of Engineering Applied Sciences, Islamabad, Pakistan (phone: +92-51-9290273-4; fax: +92-51-2208070; e-mails: nazafar@pieas.edu.pk; dr.zafar@ucp.edu.pk).

N. Sabir is a PhD student of Dr Zafar. He is an assistant professor in the Faculty of Information Technology, University of Central Punjab, Lahore, Pakistan (e-mail: nabeel.bloch@ucp.edu.pk).

A. Ali is a MS (Computer Science) student of Dr Zafar in the Faculty of Information Technology, University of Central Punjab (e-mail: amiralishahid@ucp.edu.pk).

proposing an integration of automata and Z by giving their syntax and semantics relationships reducing implementation issues of nondeterministic finite automata.

Although linking formal approaches is a well researched area [6], [7], [8], [9], [10], [11] but there does not exist much work on formalization of graphical based notations. The research work [12], [13] of Dong et al. is close to ours in which they have combined Object Z and Timed Automata. Another piece of interesting work is listed in [14], [15] in which R. L. Constable has given a constructive formalization of some concepts of automata using Nuprl. Some work of interest is reported in [16]. In [17], combination of Z notation and statecharts is established. A relationship between Z and Petri Nets is also investigated in [18], [19]. An integration of UML and B is given in [20], [21].

It is mentioned here that few results of this research were presented at [22], [23], [24]. Those preliminary results are refined in this research. In the refinement process, we observed that there were many inconsistencies and errors in those papers. And the more important is that it was not made a good use of Z notation. For example, the strings defined in the above papers were assumed to be a component of nondeterministic finite automata which is not true. In our refined results, strings are taken as input in the definition of string recognizer and language accepter. It was a conceptual error. Similarly, in our refined results, we have used schema renaming to create a new schema with same components because this is exactly what we did by giving very lengthy and poor specification in the previous results. Finally, few more errors and redundant information were identified in the old specification, which are fixed in these refined results.

In section 2, a survey on applications of formal methods is given along with a case study of programming interface to introduce some basic constructs of Z. In section 3, applications and limitations of NFA are analyzed. Integration of automata and Z is given in section 4. Analysis of the formal specification is given in section 5. Finally, the conclusion and future work are discussed in section 6.

II. Formal Methods

A. An Introduction

Formal methods are mathematically based techniques and tools for the specification, design and verification of software and hardware systems [25]. Formal methods use mathematical notations for writing specification of a system to be developed. These notations are derived from the area of discrete mathematics such as logic, set theory or graph theory. The formal specification of a system is a set of mathematical expressions with well-defined syntax and semantic [26]. By techniques of mathematical refinements, formal methods can be used at every stage of systems development in software life cycle. Once formal specification of a system is written, it can further be refined into implemented system by a process of series of refinements. The validation and verification techniques in formal methods are applied at each phase of the software development process, which ensures consistency, correctness and completeness by increasing confidence over a system under construction. The traditional systems development approaches use natural languages or graphical notations to write a specification of a system which makes the specifications highly ambiguous. Unlike these traditional approaches, formal specification uses mathematical notations having same interpretation throughout the globe [27]. Further, the use of mathematics helps to have a deeper insight of system to be developed and provides an excellent medium for modeling of systems.

One of the major limitations of traditional software development approaches is that they lack the ability to prove the specifications ensuring the absence of errors. The errors are hidden under the graphical or textual notations [28], which penetrate to the later phases of development process and are usually identified at implementation and testing phases of a system. These errors are costly and difficult to fix at that stage [29]. On the other hand, formal specification enables us to carry out proofs and makes it possible analyzing properties of systems during early stages of the development and as a result errors and inconsistencies can be identified and removed thereat. Further, formal methods are being successfully applied for development of hardware and software systems. For instance, hardware engineers use VHSIC hardware description language to model integrated circuits [29].

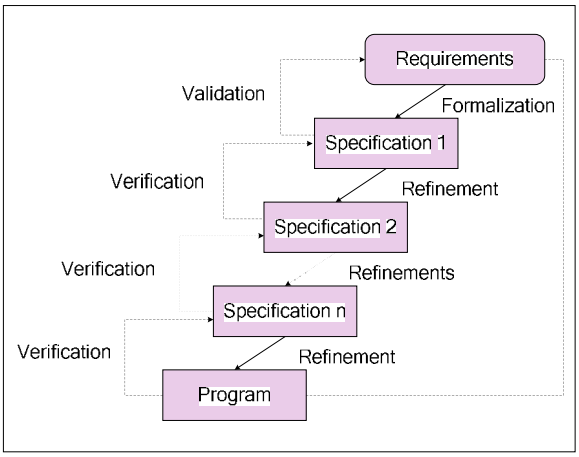

Fig. 1 Refinement process using formal methods

The process [30] of developing systems using formal methods is shown in Fig. 1. Based on the model of software life cycle, the process described in the figure can be understood as follows. The 'Requirements' are the result of requirements analysis and is normally described in informal language. The 'Specification 1' corresponds to stage of functional specification, and from 'Specification 2 to n' corresponds to the stage of design. The refinement from 'Specification n' to 'Program' corresponds to the stage of implementation or coding. Validation and verification are two basic principles that arise in systems development. Validation addresses whether the system that is produced actually fulfills

the requirements. On the other hand, verification attempts to establish whether the product of the particular phase of the software process meets the requirements established during the previous phases of the software development life cycle.

The Z notation is a model oriented approach based on set theory and first order predicate logic [31] used for specifying behavior of abstract data types and sequential programs. Z is usually used for systems development because it describes state space of a system and operations which can be performed over it. Although formal methods are being successfully applied in many research areas of computer science but at the current stage of development in formal methods, it requires an integration of formal and traditional approaches. In this paper, Z is selected to be integrated with automata theory because a natural relationship exists between these approaches.

B. Case Study: A Programming Interface

A brief overview of some important structures and operators of Z is given by taking a case from a book on "using Z: specification, refinement and proof" by Woodcock and Davies [32]. A programming interface is taken as case study for the file systems. A list of operations which is defined after defining file and an entire system can be described as: (i) read: used to read a piece of data from a file, (ii) write: used to write a piece of data to a file, (iii) access: may change the availability of a file for reading and writing over the file of the system and (iv) file and system are data types whereas the new system is defined for introducing the concept of renaming in Z.

A file is represented as a schema using a relation between storage keys and data elements. For simple specification, basic set types are used. In the formal notation, name, type, keys and data elements of a file are represented as *Name*, *Type*, *Key*, and *Data* respectively in Z notation as given below. An axiomatic definition is used here to define a variable null which is used to prove that the type of a file cannot be null even there are no contents on a file.

$[Name, Type, Key, Data]; \mid null: Type$

A file consists of two components, i.e., file contents and its type which are specified by *contents* and *type* respectively. The schema structure is usually used because of keeping specification both flexible and extensible. In the predicate part, an invariant is described proving that the file type is non null even there are no contents on it. As a file can associate a key with at most one piece of a data and hence the relation contents is supposed to be a partial function.

File

$contents: Key \rightarrow Data$
$type: Type$

$type \neq null$

The read operation is defined to interrogate the file state of the system. A successful read operation requires an existing key as input and provides the corresponding data as output. The symbol Ξ is used when there is no change in the state of a component. Now the structure $\Xi File$ means that the bindings of *File* and *File'* are equal. The decorated file, *File'*, represents the next state of the file. Here, it is in fact unchanged because the *k?* is given as input and the output is returned in output variable *d!*. The symbols *?* and *!* are used with input and output variables respectively in the schema given below. In the predicate part of the schema, first it is ensured that the input key *k?* must be in the domain of contents which is a partial function. Then the value of data against the given key is returned in the output variable *d!* of type *data*.

Read

$\Xi File$
$k?: Key$
$d!: Data$

$k? \in dom\ contents$
$d! = contents\ k?$

Another operation is defined to write contents over a given file of the system. The symbol Δ is used when there is a change in the state of a component. In the schema defined below, the $\Delta File$ gives a relationship between *File* and *File'* representing that the binding of file are now changed. The meaning of *File'* is same as defined above. In this case, the write operation defined below replaces the data stored under an existing key and provides no output to the operation. The old value of the contents is updated with the maplet $k? \mapsto d?$. It is to be noted that file type remained unchanged as defined in the predicate part of the schema. The symbol $\oplus$ an override operator is used to replace the previous value of a key with the new one in the contents of the given partial function *contents*.

Write

$\Delta File$
$k?: Key$
$d?: Data$

$k? \in \text{dom}\ contents$
$contents' = contents \oplus \{(k? \mapsto d?)\}$
$type = type'$

The structure *file* is reused in description of a file system. As a system may contain a number of files indexed using a set of names and some of which might be open. Hence, the system consists of two components namely collection of known files and set of files that are currently open. The variable *file* is used as a partial function to associate the file name and its contents. The variable *open* is of type of power set of *Name*. Finally, it is defined that the set of files which are open must be a subset of set of total files of the system as described in the predicate part of the schema *System* as given below.

System

$file: Name \rightarrow\!\!\!\!\!\!+ \; File$

$open: \mathbb{P}\, Name$

$open \subseteq \mathrm{dom}\, file$

As the open and close operations neither change name nor add and remove files of the system and hence both of these are the only access operations. On the other hand, it may change the availability of a file for reading or writing. The schema described below is used for such operations. The variable *n?* is used to check if a file which is required to be accessed exist in the system. It is also described that the file is left unchanged.

FileAccess

$\Delta System$

$n?: Name$

$n? \in \mathrm{dom}\, file$

$file' = file$

Renaming is another important concept in Z notation which we have used in this paper. For example, if we require creating another system with same pattern but with different components then renaming can be used rather than creating the new system from scratch. Renaming is sometimes useful because in this way we are able to introduce a different collection of variables with the same pattern. For example, we might wish to introduce variables *newfile* and *newopen* under the constraint of existing system *System*. In this case, the new system named as *NewSystem* can be created by renaming in horizontal form by defining: $NewSystem \triangleq System[newfile/file, newopen/open]$ which is equivalent to the schema *NewSystem* as given below in the vertical form.

NewSystem

$newfile: Name \rightarrow\!\!\!\!\!\!+ \; File$

$newopen: \mathbb{P}\, Name$

$newopen \subseteq \mathrm{dom}\, newfile$

III. Finite Automata

A. An Introduction

Automata theory has various applications and it plays an important role in many areas of computer science. The syntax analysis, natural language processing, speech recognition, modeling control behavior are some of the traditional applications of it. Automata have emerged with several modern applications in which, optimization of logic based programs, design, specification and verification of protocols [2] and human computer interaction are some other interesting examples of it. There are two major types of automata, i.e., deterministic and nondeterministic. Both have their own pros and cons. For example, deterministic automata makes implementation easy while nondeterministic automata are easy to construct and are used when the abstraction is required.

Both the deterministic and nondeterministic finite automata are equivalent in power, in a sense, that if a language is recognized by one it is also recognized by the other and vice versa. The NFAs are sometimes useful because its construction is much easier than constructing of a DFA. This is due to the use of less mathematical work in building NFA as compared to DFA. Further, many important properties in automata can be established easily by using NFA. To prove, for example, that concatenation or union of two regular languages is regular using NFA is easier than using deterministic finite automata.

B. Nondeterministic Finite Automata

Nondeterministic finite automata (NFA) are abstract mathematical models of systems based on mathematical tools, techniques and notations which have graphical representation as well. These models can be used to perform computations on input in a predefined mechanism. An NFA reads a string of input symbols and moves to a new state after reading a part of input. The process continues and if we are able to reach any of the accepting state by using a series of movements then the given input is accepted. The driving function also called transition function decides the next state based on the input symbol and the current state given to it.

DFA and NFA differ only in terms of their transition functions. In case of a DFA, the transition function transforms the control of a machine from one state to a unique state for every symbol of the input. In case of NFA, for any input symbol, its next state is not uniquely determined by the transition function. There might be a single or a set of states, which can be empty, and this is why NFA is called nondeterministic automata. In formal theory of automata, it can be proved that both DFA and NFA are equivalent in power. It means that for any given DFA, one may construct an equivalent NFA and vice-versa. Conversion from NFA to DFA may cause exponential grow in size of DFA and, consequently, storage space may require proportional to the number of states in the given NFA. This is one of the major issues in representation of systems using NFA.

If in a given NFA, we are able to move from one state to another without consuming any input symbol then the resultant NFA is called NFA with null string (ε) and is defined by NFA $\cup \{\varepsilon\}$. When we move from one state to another by reading the null string, it creates an ambiguity in the NFA. But, on the other hand, it reduces a mathematical work in description of the properties and operation over the given automata. If we are enough comfortable in mathematical details of an NFA, then the above ambiguity can be removed by introducing a new possible set of states in which the transition function enters. In this paper, we have supposed that our NFA is based on the set of alphabets in addition to the null string. The inclusion of ε in

the alphabets increases more complexity in conversion from nondeterministic to deterministic finite automata.

IV. Transformation from NFA to Z

A semantics transformation from NFA to Z is given here. In the transformation, at first, we will give a formal specification of an NFA. Then a formalism of the string accepter will be described. A string and an NFA will be given as inputs to the string accepter and it will check whether the given string is accepted by it. Then our machine, string accepter, will be extended to the language recognizer. Here, a language and the NFA will be inputs, and the language recognizer will check if the given language is recognized by it. At the end, we will give formal construction of a more powerful tool which recognizes union of two regular languages. As a result, our transformation from NFA to Z will be based on:

- transformation of NFA,
- string accepter,
- language recognizer and
- formalization of union of regular languages.

It is mentioned here that the definitions used, in this paper, are based on some well known books on automata and computation theory [33], [34].

A. Transformation of NFA

We start with the definition of NFA which is a 5-tuple (Q, ∑, δ, q0, F), where Q is a finite non-empty set of states, ∑ is a finite set of alphabets including empty string, δ is a transition function which takes a state and an alphabet as inputs and produces a set of states as output, q0 is the initial state and F is a finite set of final states. The above 5-tuple (Q, ∑, δ, q0, F) is an NFA because it takes a state, and an input symbol or an empty string and produces a set of states. Now we start a procedure for formal construction of NFA using Z. The sets of states and alphabets in the 5-tuple are represented as *Q* and *Sigma* respectively in the Z.

[Q, Sigma]

In modeling sets using Z, 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. As a consequent, our *Q* and *Sigma* are sets over which we cannot define any operation. For example, cardinality to know the number of elements in a set, subset or complement operations cannot be defined.

A variable *states* is introduced to describe a set of states for NFA. Since a given state q is of type Q, therefore *states* is of type of power set of Q. To describe a set of alphabets for the same NFA, a variable *alphabet* is used which is of type of power set of Sigma. The empty string is represented as *apsi* and is of type of *Sigma*. As we know that for any input symbol, the next state of NFA can only be uniquely determined by the transition function if we suppose that its output is a set of states. As a result the δ relation can be defined as a function. This is because for each input (q, a) where q is a state and a is an alphabet, there must be a unique set s of states, which is image of (q, a), under the transition function δ. Therefore, we can declare δ as: $\delta: Q \times Sigma \leftrightarrow \mathbb{P} Q$. The set of dead states, usually not shown in NFA, is represented as *null* and is of type of power set of Q. The initial state *q0* is of type Q. Our last one construct F is represented by *finals* and is of type of power set of Q. After designing NFA, we will be required to check whether a given string is accepted by the NFA. For this purpose, we define a new variable strings which can be generated from the set of alphabets of NFA, and its type is:

$Strings == seq\ Sigma$

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

NFA

$states: \mathbb{P} Q$
$alphabets: \mathbb{P} Sigma$
$apsi: Sigma$
$delta: Q \times Sigma \leftrightarrow \mathbb{P} Q$
$q0: Q$
$null: \mathbb{P} Q$
$finals: \mathbb{P} Q$

$apsi \in alphabets$
$q0 \in states$
$finals \subseteq states$
$\forall q1, q2: Q; a: Sigma \mid q1 \in states \land q2 \in states \land a \in alphabets$
$\bullet\ \exists s1, s2: \mathbb{P} Q \mid s1 \subseteq states \land s2 \subseteq states \land ((q1, a), s1) \in delta$
$\land ((q2, a), s2) \in delta \bullet (q1, a) = (q2, a) \Rightarrow s1 = s2$

Invariants: (1) The empty string is a member of alphabets. (2) The initial state q0 is an element of states. (3) The set of final states is a subset of total states. (4) For each (q, a), where q is a state and a is an alphabet, there is a unique set of states s such that: *delta(q, a) = s*.

B. String Accepter

In this section, a string accepter is designed which takes a string and an NFA as input and checks whether the given string is accepted by it. Let NFA = (Q, ∑, δ, q0, F) be an NFA and $w = w_1 w_2 \ldots w_n$ be a string, for each $w_i \in \sum$ and i = 1, 2, . . ., n. We say that NFA accepts the string w if there exists a sequence of states $s_0, s_1, \ldots, s_n$ in Q satisfying:

- $s_0 = q_0$,
- $s_{i+1} \in \delta(s_i, w_{i+1})$, $\forall$ i = 0, 1,… , n-1 and

- $s_n \in F$.

The first condition states that NFA starts from the initial state q0. The second condition means that s_{i+1} is one of the allowable next states when NFA is in state s_i and reads w_{i+1}. It is to be noted that $\delta(s_i, w_{i+1})$ is a set consisting of allowable next states and hence s_{i+1} is a member of that set. Finally, it is stated that our machine accepts the input if it ends up in an accepting state. Now coming back to our formal definition of string accepter, we have two inputs NFA and a word w?. The symbol Ξ before NFA shows that the machine will not change but it will only be used. The symbol question mark ?, after the string w means that w is given as input. This is simply syntax of defining input in Z notation. The string accepter is represented as *StringAcceptedByNFA* as a schema given below. The inputs are given in the first part of the schema and constraints are defined in the second part of it.

StringAcceptedByNFA

ΞNFA
$w?: Strings$

$w? \in strings$
$\exists s: \text{seq } Q \mid \text{ran } s \subseteq states \land \# s = \# w? + 1 \bullet \# s \geqslant 1$
$\Rightarrow s\ 1 = q0 \land s\ (\# s) \in finals \land$
$(\forall i: \mathbb{N} \mid \# w? \geqslant i + 1 \bullet (i \in 1 .. \# s - 1$
$\Rightarrow (\exists ss: \mathbb{P}\ Q \mid ss \subseteq states \land$
$((s\ i, w?\ (i + 1)), ss) \in delta \bullet s\ (i + 1) \in ss)))$

Invariants: (1) The input string w?, is based on the set of alphabets of nondeterministic finite automata. (2) For a given string w? of length n, there must be a sequence of states of length, at most, n+1 which satisfies the conditions: (i) its first element is the initial state of the given NFA, (ii) the last element is a member of final states, (iii) finally, for state s_i and an alphabet w_{i+1}, the transition function is defined from (s_i, w_{i+1}) to s_{i+1}, $s_{i+1} \in$ ss such that: $\delta(s_i, w_{i+1}) =$ ss or $s_{i+1} \in \delta(s_i, w_{i+1})$, $\forall$ i = 0, 1,…, n-1.

C. Language Recognizer

In language recognizer, we have reused the definition of string accepter designed above and extended over the set of all the strings of a language. A language recognizer is designed which takes a language and an NFA as inputs and returns the value true if the language is recognized by it. Mathematically, we can define it as:

Language Recognizer =
{<NFA, language> | $\forall$ w $\in$ language, NFA accepts w}.

We have two inputs named as an NFA and a language. The recognizer is denoted by *LanguageAcceptedByNFA* in Z notation as a schema. Its formal specification along with invariants over it is given below. The invariants of string accepter are generalized over language accepter.

LanguageAcceptedByNFA

ΞNFA
$language?: \mathbb{P}\ Strings$

$\forall w: Strings \mid w \in language? \bullet w \in strings$
$\forall w: Strings \mid w \in language?$
$\bullet \exists s: \text{seq } Q \mid \text{ran } s \subseteq states \land \# s = \# w + 1 \bullet \# s \geqslant 1$
$\Rightarrow s\ 1 = q0 \land s\ (\# s) \in finals \land (\forall i: \mathbb{N} \mid \# w \geqslant i + 1$
$\bullet (i \in 1 .. \# s - 1 \Rightarrow (\exists ss: \mathbb{P}\ Q \mid ss \subseteq states \land$
$((s\ i, w\ (i + 1)), ss) \in delta \bullet s\ (i + 1) \in ss)))$

Invariants: (1) Every input string of language is based on alphabets of NFA. (2) Every string of the language satisfies the conditions as in string accepter.

D. Formalism of Union of NFAs

Now we start with two given NFAs, NFA1 = (Q1, ∑, δ1, q01, F1) and NFA2 = (Q2, ∑, δ2, q02, F2). These NFAs are inputs in design of union of NFA accepting all the strings that are either accepted by NFA1or NFA2. All the components of resultant NFA are listed in the schema given below, and a relationship between NFA1, NFA2 and their resultant is established in terms of predicates in the second part of the schema. Before describing union of two nondeterministic automata, we describe NFA1 and NFA2 by renaming the components of NFA as given below. An introduction to renaming operator is given in section 2.

$NFA1 \mathrel{\widehat{=}} NFA[states1/states, alphabets1/alphabets, apsi1/apsi, delta1/delta, q01/q0, null1/null, finals1/finals]$

$NFA2 \mathrel{\widehat{=}} NFA[states21/states, alphabets2/alphabets, apsi2/apsi, delta2/delta, q02/q0, null2/null, finals2/finals]$

NFA1uNFA2

$NFA1; NFA2$
$states: \mathbb{P}\ Q$
$alphabets: \mathbb{P}\ Sigma$
$apsi: Sigma$
$delta: Q \times Sigma \leftrightarrow \mathbb{P}\ Q$
$q0: Q$
$null: \mathbb{P}\ Q$
$finals: \mathbb{P}\ Q$

$apsi \in alphabets$
$q0 \in states$
$finals \subseteq states$
$\forall q1, q2: Q; a: Sigma \mid q1 \in states \land q2 \in states \land a \in alphabets$
$\bullet \exists s1, s2: \mathbb{P}\ Q \mid s1 \subseteq states \land s2 \subseteq states \land ((q1, a), s1) \in delta$
$\land ((q2, a), s2) \in delta \bullet (q1, a) = (q2, a) \Rightarrow s1 = s2$
$states = states1 \cup states2 \cup \{q0\}$
$finals1 = finals1 \cup finals2$
$\forall q:Q; a:Sigma; ss1: \mathbb{P}Q \mid q \in states \land a \in alphabets \land$
$((q,a), ss1) \in delta$
$\bullet (q \in states1 \Rightarrow (\exists ss2: \mathbb{P}\ Q \mid ((q, a), ss2) \in delta1 \bullet ss1 = ss2))$
$\land (q \in states2 \Rightarrow (\exists ss3: \mathbb{P}\ Q \mid ((q, a), ss3) \in delta2 \bullet ss1 = ss3))$
$\land (q = q0 \land a = apsi \Rightarrow ss1 = \{q01, q02\})$
$\land (q = q0 \land \neg\ a = apsi \Rightarrow ss1 = null)$

Invariants: (1) The empty string is a member of set of alphabets of NFA. (2) The initial state q0 must be an element of set of total states of the resultant NFA. (3) The set of final states is a subset of set of total states of the given NFA. (4) For each (q, a), where q is a member of set of states and, a is a member of set of alphabets, there is a unique set of states s such that: delta (q, a) = s. (5) The set of states of the resultant NFA is equal to the union of sets of states of NFA1, NFA2 and a set consisting of a single element q0 which is newly created. In fact, q0 is a new state introduced at the time of union and is initial state in the NFA. (6) The set of final states of the NFA is equal to the union of the sets of final states of NFA1 and NFA2. (7) For any state and an element of the alphabets of NFA, the transition function is modified as given above in the schema *NFA1uNFA2*.

V. Analysis and Proof Techniques

The Z/EVES is a powerful tool which can be used for analyzing a specification written in Z. There is a much tool support for analyzing the Z specification. Parsing, type and domain checking, schema expansion, precondition calculation, refinement proofs, and theorem proving are major facilities of it. Formal models of nondeterministic finite automata in Z are checked and strengthened using this Z/EVES tool-set. Formal models described in the form of schemas are analyzed with four major techniques offered by Z/EVES tool-set namely syntax and type checking, domain checking, reduction and prove by reduce.

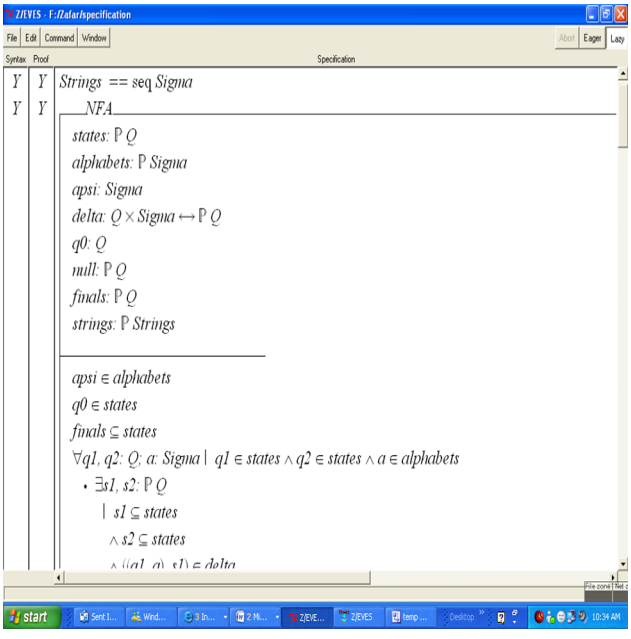

Fig. 2 Snapshot of specification analysis

A snapshot of the results of checking and analyzing the formal specification is given above in Figure 2. It is to be mentioned here that at first an abstract model was proposed and then refined to remove inconsistencies and ambiguities. While proving the formal models using Z/EVES, two types of results were obtained. Firstly, some schemas were well written and proved automatically without any prove assistance of the tool. And then, some schemas were proved using the prove assistance of the tool by reduction.

VI. Conclusion

In this paper, we have described formal specification of an algorithm which can be used to construct NFA accepting union of regular languages. A new approach by NFA and Z notation is proposed. Although, a part of the integration is treated but we have observed that this approach can be extended to give formal specification of more powerful tools. We have identified a relationship between some structures of these approaches and proved it in a constructive way. After integrating automata and Z notation, we have observed that a natural relationship exists there which proves the importance and originality of this research. It is observed that after extending the NFA by adding some functionalities of Z, the modeling power for complex systems can be increased.

An exhaustive survey of existing work was done before starting this research. There exists a lot of research work on integration of approaches as discussed in the introduction part of this paper. The most relevant work reported in [15], [16], [19], [20], [25] was found but our research is different of it because of conceptual level integration of Z notation and automata. Further, formalizing graphical based notations is not easy [35] which increase importance of this research.

Behavior of a system can be best captured by automata whereas its state space can be described ideally using Z notation. This is why Z is integrated with automata in this research. It is believed that this linkage will be useful in the development of automated integrated tools and techniques for software engineering processes. Formalization of some other important concepts in automata is under progress and will appear soon in our future work. Further, we have taken some assumptions in this integration. For example, in union operation, it was assumed that the set of alphabets in both of the automata are same. These assumptions were taken for simplicity of construction. In our future work, such assumptions will be relaxed and a more generic integration will be proposed between Z and automata.

Acknowledgment

This research is supported by Centre for Research in Computer Science, University of Central Punjab, Lahore, Pakistan. Specially, the authors are thankful to Associate Dean Professor Ajmal Hussain, Faculty of Information Technology and pro-rector Dr. Fehmida Sultana for providing research funds and facilities in the centre.

References

[1] M. Y. Vardi, T. Wilke, "Automata from Logic to Algorithms," Logic and Automata, 2007.

[2] J. M. Spivey, "The Z notation, A Reference Manual," Englewood Cliffs, NJ, Prentice-Hall, 1989.

[3] I. J. Holub, "Finding Common Motifs with Gaps using Finite Automata," In Implementation and Application of Automata, Springer-Verlag, pp: 69-77, ISBN 3-540-37213-X, 2006.

[4] K. Brouwer, W. Gellerich, E. Ploedereder, "Myths and Facts about the Efficient Implementation of Finite Automata and Lexical Analysis," Springer-Berlin, 2006.

[5] I. Meisels, M. Saaltink, "The Z/EVES Reference Manual, TR-97-5493-03," ORA Canada, CANADA, 1997.

[6] E. A. Boiten, J. Derrick, G. Smith, "Integrated Formal Methods (IFM 2004)," Canterbury, UK, Springer, 2004.

[7] J. Davies, J. Gibbons, "Integrated Formal Methods (IFM 2007)," Oxford, UK, Springer-Verlag, 2007.

[8] J. Romijn, G. Smith, J. v. d. Pol, "Integrated Formal Methods (IFM 2005)," The Netherlands, Springer, 2005.

[9] K. Araki, A. Galloway, K. Taguchi, "Integrated Formal Methods (IFM 99)," York, UK, Springer-Verlag, 1999.

[10] M. Butler, L. Petre, K. Sere, "Integrated Formal Methods (IFM 2002)," Turku, Finland, Springer-Verlag, 2002.

[11] W. Grieskamp, T. Santen, B. Stoddart, "Integrated Formal Methods (IFM 2000)," Germany, Springer-Verlag, 2000.

[12] J. S. Dong, R. Duke, P. Hao, "Integrating Object-Z with Timed Automata," 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2005), pp: 488-497, 2005.

[13] J. S. Dong, et al., "Timed Patterns, TCOZ to Timed Automata," 6th International Conference on Formal Engineering Methods (ICFEM'04), LNCS, pp: 483-498, 2004.

[14] R. L. Constable, et al., "Formalizing Automata II: Decidable Properties," Cornell University, 1997.

[15] R. L. Constable, et al., "Constructively Formalizing Automata Theory," Foundations Of Computing Series, MIT Press, ISBN:0-262-16188-5, 2000.

[16] R. Bussow, W. Grieskamp, "A Modular Framework for the Integration of Heterogeneous Notations and Tools," Integrated Formal Methods (IFM 99), York, UK, Springer-Verlag, pp: 211–230, 1999.

[17] R. Büssow, R. Geisler, M. Klar, "Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study," Fundamental Approaches to Software Engineering, Springer Berlin, ISBN, 978-3-540-64303-6, 2004.

[18] M. Heiner, M. Heisel, "Modeling Safety Critical Systems with Z and Petri nets," International Conference on Computer Safety, Reliability and Security, LNCS, pp: 361–374, 1999.

[19] X. He, "Pz nets a Formal method Integrating Petri nets with Z," Information & Software Technology, 43(1), pp: 1–18, 2001.

[20] H. Leading, J. Souquieres, "Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B," Asia-Pacific Software Engineering Conference (APSEC02), Australia, 2002.

[21] H. Leading, J. Souquieres, "Integration of UML Views using B Notation," Workshop on Integration and Transformation of UML models (WITUML02), 2002.

[22] N. A. Zafar, N. Sabir, and A. Ali, "Semantics Transformation of NFA to Z Notation by Constructing Union of Regular Languages", the 8th WSEAS Int'l Conference on Applied Computer Science (ACS'08), pp. 70-75, Italy, 2008.

[23] N. A. Zafar, N. Sabir, and A. Ali, "Construction of Intersection of Nondeterministic Finite Automata using Z Notation", International Journal of Computer Science, vol. 3(2). pp. 96-101 2008.

[24] S. Riaz, and N. A. Zafar, "Constructive Formal Conversion of Moore Machine to Deterministic Finite Automata", The 10th WSEAS International Conference on Mathematical Methods, Computational Techniques and Intelligent Systems (MAMECTIS'08), Greece, 2008.

[25] C. Heitmeyer, "On the Need for Practical Formal Methods," LNCS, Vol.1486, pp: 18-26, 1998.

[26] E. Ciapessoni, et al., "From Formal Models to Formally-Based Methods: An Industrial Experience," TOSEM, vol.8 (1), pp: 79-113, 1999.

[27] J. P. Bowen, "Ten Commandments of Formal Methods," IEEE Computer, Vol.28, No.4, pp: 56-63, 1995.

[28] J. P. Bowen, M. G. Hinchey, "The Use of Industrial-Strength of Formal Methods," Proceedings of 21st International Computer Software & Application Conference (COMPSAC'97), pp: 332-337, 1997.

[29] M. Barjaktarovic, "The State-of-the-Art in Formal Methods," AFOSR Summer Research Technical Report for Rome Research Site, Formal Methods Framework-Monthly Status Report, F30602-99-C-0166, WetStone Technologies, 1998.

[30] S. Liu, and R. Adams, Limitations of Formal Methods and an Approach to Improvement, Technical Report, Hiroshima City University, 1995.

[31] R. W. Butler, "What is Formal Methods?," NASA LaRC Formal Methods Program, 2001.

[32] J. Woodcock, and J. Davies, "Using Z: Specification, Refinement and Proof," Prentice Hall International, 1996.

[33] J. E. Hopcroft, R. Motwani, J. D. Ullman, "Introduction to Automata Theory, Language and Computation," Addison-Wesley, Reading, 2001.

[34] M. Sipser, "Introduction to the Theory of Computation," Course Technology, ISBN-13: 9780534950972, 2005.

[35] C. T. Chou, "A Formal Theory of Undirected Graphs in Higher Order Logic," 7th Int'l Workshop on Higher Order Logic Theorem Proving and Application, pp: 144-157, 1994.

Nazir A. Zafar was born in 1969 in Pakistan. He received his M.Sc. (Math. in 1991), M. Phil (Math. in 1993), and M.Sc. (Nucl. Engg. in 1994) degrees from Quaid-i-Azam University, Islamabad, Pakistan. He was awarded PhD degree in Computer Science from Kyushu University, Japan, in 2004. He is the regular employee of Pakistan Institute of Engineering and Applied Sciences (PIEAS). Dr. Zafar has served at various universities and scientific organizations in Pakistan. Currently, he is on leave from PIEAS and working with University of Central Punjab (UCP) in the Faculty of information Technology, Lahore, Pakistan. He is the founder of Formal Methods Research Group at UCP. His current research interests are modeling of complex systems using formal approaches, integration of approaches etc. He is an active member of Pakistan Mathematical Society. He is member of various societies and organizations. He is also member of editorial boards of many international journals. Dr. Zafar has lectured at national level promoting use and applications of formal methods in Pakistan.

Nabeel Sabir was born in 1979 in Pakistan. He received his Master in Computer Science in 2003 from U.C.P., MS (Computer Science) in 2007 from the same university. Currently, he is a faculty member and PhD student of Dr. Zafar at UCP, Lahore.

Amir Ali was born in 1979 in Pakistan. He received his M. Sc. (Math. in 2003) from B.Z.U., Multan, Post Graduate Diploma (Computer Science) in 2005 from Q.A.U., Islamabad, Master in Computer Science in 2007 from NUML, Islamabad. And, currently, he is a student (MS leading to PhD) of Dr. Zafar at UCP, Lahore.