J. Software Engineering & Applications, **2009, 2: 77-85**
doi:10.4236/jsea.2009.22012 Published Online July 2009 (www.SciRP.org/journal/jsea)

Refinement in Formal Proof of Equivalence in Morphisms over Strongly Connected Algebraic Automata

Nazir Ahmad Zafar, Ajmal Hussain, Amir Ali

Faculty of Information Technology, University of Central Punjab, 31-A, Main Gulberg, Lahore, Pakistan
Email: {dr.zafar, ajmal, amiralishaihid}@ucp.edu.pk

Received January 13th, 2009; revised February 27th, 2009; accepted March 24th, 2009.

ABSTRACT

Automata theory has played an important role in computer science and engineering particularly modeling behavior of systems since last couple of decades. The algebraic automaton has emerged with several modern applications, for example, optimization of programs, design of model checkers, development of theorem provers because of having properties and structures from algebraic theory of mathematics. Design of a complex system not only requires functionality but it also needs to model its control behavior. Z notation is an ideal one used for describing state space of a system and then defining operations over it. Consequently, an integration of algebraic automata and Z will be an effective computer tool which can be used for modeling of complex systems. In this paper, we have combined algebraic automata and Z notation defining a relationship between fundamentals of these approaches. At first, we have described algebraic automaton and its extended forms. Then homomorphism and its variants over strongly connected automata are specified. Finally, monoid endomorphisms and group automorphisms are formalized, and formal proof of their equivalence is given under certain assumptions. The specification is analyzed and validated using Z/EVES tool.

***Keywords*:** *Formal Methods, Automata, Integration of Approaches, Z Notation, Validation*

1. Introduction

Almost all large, complex and critical systems are being controlled by computer software. When software is used in a complex system, for example, in a safety critical system its failure may cause a huge loss in terms of deaths, injuries or financial damages. Therefore, constructing correct software is as important as its other counterparts, for example, hardware or electromechanical systems [1]. Formal methods are approaches used for specification of properties of software and hardware systems insuring correctness of a system [2]. Using formal methods, we can describe a mathematical model and then it can be analyzed and validated increasing confidence over a system [3]. At the current stage of development in formal approaches, it is not possible to develop a system using a single formal technique and as a result its integration is required with other traditional approaches. That is why integration of approaches has become a well-researched area in computing systems [4,5,6,7,8,9,10]. Further, design of a complex system not only requires capturing functionality but it also needs to model its control behavior. There are a large variety of specification techniques which are suitable for specific aspects in the software development process. For example, algebraic techniques, Z, VDM, and B are usually used for defining data types while process algebra, petri nets and automata are some of the examples which are best suited for capturing dynamic aspects of systems [11]. Because of well-defined mathematical syntax and semantics of the formal techniques, it is required to identify, explore and develop relationships between such approaches for modeling of complete, consistent and correct computerized systems.

Although there exists a lot of work on integration of approaches but there does not exist much work on formalization of graphical based notations. The work [12,13] of Dong *et al*. is close to ours in which they have integrated Object Z and Timed Automata. Another piece of good work is listed in [14,15] in which R. L. Constab- le has given a constructive formalization of some important concepts of automata using Nuprl. A combination of Z with statecharts is established in [9]. A relationship is investigated between Z and Petri-nets in [16,17]. An integration of UML and B is given in [18,19]. Wechler, W. has introduced algebraic structures in fuzzy automata [20]. A treatment of fuzzy automata and fuzzy language theory is given when the set of possible values is a closed interval [0, 1] in [21]. Ito, M., has described formal languages and automata from the algebraic point of

view in which he has investigated the algebraic structures of automata and then a kind of global theory is treated [22]. Kaynar, D. K *et al*. has presented a modeling framework of timed computing systems [23]. In [24], Godsil, C. *et al*. has presented some ideas of algebraic graphs with an emphasis on current rather than classical topics of graphs.

Automata theory has proved to be a cornerstone of theoretical computer science since last couple of decades. Compiler constructions, modeling of finite state systems, natural language processing, defining a regular set of finite words are some of the traditional applications of automata. The algebraic automaton is an advanced form of automata having properties and structures from algebraic theory of mathematics. It has emerged with several modern applications, for example, optimal representation and efficient implementation of algorithms, optimization of programs, speech recognition, design of model checkers, image processing and verification of protocols. The applications of algebraic theory of automata are not limited to computers but are being seen in many other disciplines of science and engineering, for example, representing characteristics of natural phenomena in biology [25]. Modeling of chemical systems using cellular automata is another important application area of it [26]. Another interesting application of automata is presented in [27] in which it is described a system for specifying and automatically synthesizing physics-based animation programs based on hybrid automata.

It is obvious that theory of automata has various application areas as discussed above. Because of having some interesting properties from algebra, the algebraic automata has increased its importance in some special application domain of computer science. For example, algebraic automata can be used in static as well as in dynamic part of distributed systems. To understand it, we can suppose objects or entities representing a collection of distributed systems. As in this case, for many applications, there is no precedence of order in computation that means the entities can be concatenated in any order and hence the associative property is satisfied. Further, after identifying the identity element in this collection of objects, the structure produced is called a monoid which is an abstract algebraic structure. It is to be mentioned that the identity element can be identified if it exists based on the nature of the problem. Because of having these special algebraic characteristic, an automaton can be extended to develop algebraic automata which can be used for specification and capturing control behavior over such systems. After understanding the importance of algebraic automata, a relationship is identified and proposed, in this paper, between algebraic automata and Z notation thus facilitating the modeling techniques for complex systems. To achieve the objective of proposed integration, at first, we have given formal description of algebraic automaton and its other extended forms. The strongly connected automaton is described by reusing the structure defining algebraic automata. Then homomorphism, which is an important structure in verifying symmetry of the algebraic structures, and its other variants over strongly connected automata are formalized. Next, monoid endomorphisms and group automorphisms are described. Finally, a formal proof of their equivalence is given under certain assumptions. The specification is analyzed and validated using Z/EVES tool. The major objectives of this paper are: 1) to identify a linkage between automata and Z notation to be useful for modeling the systems and 2) providing a syntactic and semantic relationship between Z and algebraic automata. In Section 2, an introduction to Z notation is given. In Section 3, an overview of algebraic automata is provided. Formal construction of proof showing equivalence of algebraic structures is given in Section 4. Finally, conclusion and future work are discussed in Section 5.

2. An Introduction to Z Notation

Formal methods are approaches, based on the use of mathematical techniques and notations, for describing and analyzing properties of software systems [28]. 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 such as logic, set theory, graph theory and algebraic structures. There are several ways in which formal methods may be classified. One frequently-made distinction is between property oriented and model oriented methods [29]. Property oriented methods are used to describe the operations which can be performed on a system and then defining relationships between these operations. Property oriented methods usually consist of two parts. The first one is the signature part which is used for defining the syntax of operations and the second one is an equations part used for defining the semantics of the operations by a set of equations called the rules. Algebraic specification of abstract data types [30] and the OBJ language [31] are examples of property oriented methods.

Model oriented methods are used to construct a model of a system's behavior and then allow us to define operations over it [32]. Z notation is one of the most popular specification languages which is a model oriented approach based on set theory and first order predicate logic [33]. It is used for specifying behavior of abstract data types and sequential programs.

A brief overview of some important structures and operators of Z, used in our research, is given by taking a case from a book on "Using Z: specification, refinement and proof" by Woodcock J. and Davies J., [34]. A programming interface is taken as case study for file systems.

A list of operations which is defined after defining file and an entire system is, 1) read: used to read a piece of data from a file, 2) write: used to write a piece of data to a file, 3) access: may change the availability of a file for reading and writing over the file of the system.

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 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*]; | *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 be-cause 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 con-tents in it. As a file can associate a key with at most one piece of a data and hence the relation *contents* is sup-posed to be a partial function.

File

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

$type \neq null$

The read operation is defined over the file to interrogate the file state. 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 Ξ *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!*.

Read

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

$k? \in \mathrm{dom}\ contents$
$d! = contents\ k?$

Another operation is defined to write contents over the file. The symbol Δ is used when there is a change in the state of a component (schema). In the schema defined below, the structure $\Delta File$ gives a relationship between *File* and *File*', representing that the binding of *File* is now changed. The meaning of *File*' is the same as defined above. In this case, the write operation defined below replaces the data stored under an existing key and provides no output. The old value of contents is updated with 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 $\oplus$ is an override operator which is used to replace the previous value of a key with the new one in a given function.

Write

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

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

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 files known to the system 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*. The set of files which are open must be a subset of set of total files as described in the predicate part of the schema given below.

System

$file: Name \pfun File$
$open: \mathbb{P}\ Name$

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

As the open and close operations neither change name of any file nor add and remove files from the system. It means both of these are access operations. 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 the file to be accessed exist in the system. In the schema, 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 of Z which we have used in our research. 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 the 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 *NewSystem* can be created in horizontal form by defining: *NewSystem* A *System*[*newfile*/*file*, *newopen*/*open*] which is equivalent to the schema given below in the vertical form.

NewSystem

$newfile: Name \pfun File$

$newopen: \mathbb{P}\ Name$

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

3. Algebraic Automata

As we know that automata theory has become a basis in the theoretical computer science because of its various applications and playing a vital role in modeling scientific and engineering problems [35]. Modeling control behavior, compiler constructions, modeling of finite state systems are some of the traditional applications of it [36,37,38]. Automata can be classified because of its deterministic and nondeterministic nature. Both types of automata have their own pros and cons in modeling of systems. Both of the automata are equivalent in power because if a language is accepted by one, it is also accepted by the other. Nondeterministic finite automata (NFA) are useful because constructing an NFA is easier than deterministic finite automata (DFA). On the other hand, DFA is useful when implementation is required. Consequently, both of the automata can be used based on the requirements and nature of a problem.

Algebraic automaton which is an abstract form of automata, however, has some properties and structures from algebraic theory of mathematics. The algebraic automata have emerged with several modern applications in computer science. Further, the applications of algebraic theory are not limited to computers but are being seen in other disciplines of science, e.g., representing chemical and physical phenomena in chemistry and biology. It is a well known fact that a given automata may have different implementations and consequently its time and space complexity must be different, which is another issue in modeling using automata. Therefore it is also required to describe the formal specification of automata for its optimal implementation. Further, this relationship will result a useful tool at academic as well as industrial level. A formal verified linkage of algebraic automata and Z is given in the next section.

4. Formal Proof of Equivalence

Now we give formal description of some important concepts of algebraic automata using Z notation. And an equivalence of endomorphisms and automorphisms over strongly connected automata is formalized. The definitions used in this section are based on a book on "Algebraic Theory of Automata and Languages" [22].

4.1 Formalizing Automaton and its Extensions

An algebraic automaton (AA) is a 3-tuple (Q, Σ, δ), where 1) Q is a finite nonempty set of states, 2) Σ is a finite set of alphabets and 3) δ is a transition function which takes a state and an alphabet and produces a state. To formalize AA, Q and Σ are denoted by *S* and *X* respectively.

[Q, *X*]

In modeling using sets in Z, we do not impose any restriction upon 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 *Q* and *X* are sets over which we cannot define any operation. For example, cardinality to know the number of elements in a set cannot be defined. Similarly, subset and complement operations over these sets are not defined as well.

To describe a set of states for AA, a variable *states* is introduced. Since, a given state q is of type Q therefore *states* must be of type of power set of Q. For a set of alphabets, the variable *alphabets* is used which is of type of power set of *X*. As we know that δ is a function because for each input (*q1*, *a*), where *q1* is *a* state and a is an alphabet there must be a unique state, which is image of (q1, a) under the transition function δ. Hence we can declare δ as, *delta*: $Q \times X \rightarrow Q$.

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 the relationships between the objects. This relationship will be used in terms of constraints after composing the objects. The schema structure is used for composition because it is very powerful at abstract level of specification and helps in describing a good specification approach. All of the above components are encapsulated and put in the schema named as *Automaton*. The formal description of it is given below.

Automaton

$states: \mathbb{P}\, Q$
$alphabets: \mathbb{P}\, X$
$delta: Q \times X \rightarrow Q$

$\forall q: Q; x: X \mid q \in states \wedge x \in alphabets$
$\bullet\ \exists t: Q \mid t \in states \bullet delta\ (q, x) = t$

Invariants: For each input (s, x) where s is an element of *states* and x is a member of *alphabets*, there is a unique state t such that: *delta* $(s, x) = t$.

An extended form of algebraic automaton is denoted by *EA*. In the extended form, two more components are added in addition to the three components of the algebraic automaton defined above. In the schema given below, the variables *states* and *alphabets* have the same meaning but the third one *delta* function is refined. In the extended form, the *delta* function takes a states and a string as inputs and produces the same state or new state as output. We also need to compute the set of all the strings based on the set of alphabets and hence a fourth variable is used and denoted by *strings* which is of type of power set of set of all the sequences (strings: $\mathbb{P}$ (seq X)). As we know that a sequence can be empty and hence a fifth variable is used representing it and is denoted by *epsilon* of type *seq X*.

EA

$states: \mathbb{P}\, Q$
$alphabets: \mathbb{P}\, X$
$strings: \mathbb{P}\, (\text{seq}\, X)$
$delta: Q \times \text{seq}\, X \rightarrow Q$
$epsilon: \text{seq}\, X$

$epsilon \in strings$
$\forall q: Q \mid q \in states \bullet delta\ (q, epsilon) = q$
$\forall q: Q; a: X; u: \text{seq}\, X \mid q \in states \wedge a \in alphabets \wedge u \in strings$
$\bullet\ delta\ (q, (\langle a \rangle \frown u)) = delta\ ((delta\ (q, \langle a \rangle)), u)$

Invariants: 1) The null string is an element of strings. 2) If transition function takes a state and the null string epsilon, and it produces the same state. 3) For each input where s is an element of states, a is an alphabets and u is an element of strings, delta function is defined as: $delta\ (s, (\langle a \rangle \circ u)) = delta\ ((delta(s, \langle a \rangle)), u)$.

Another extended form of algebraic automaton is a strongly connected automaton. A strongly connected is a one for which if for any two given states there exists a string s such that the delta function connects these states through the string s. The strongly connected automaton is represented by SCEA as a schema in Z notation and described as given below. It has the same number of components and properties in addition to one more constraint defined here.

SCEA

$states: \mathbb{P}\, Q$
$alphabets: \mathbb{P}\, X$
$strings: \mathbb{P}\, (\text{seq}\, X)$
$delta: Q \times \text{seq}\, X \rightarrow Q$
$epsilon: \text{seq}\, X$

$epsilon \in strings$
$\forall q: Q \mid q \in states \bullet delta\ (q, epsilon) = q$
$\forall q: Q; a: X; u: \text{seq}\, X \mid q \in states \wedge a \in alphabets \wedge u \in strings$
$\bullet\ delta\ (q, (\langle a \rangle \frown u)) = delta\ ((delta\ (q, \langle a \rangle)), u)$
$\forall q1, q2: Q \mid q1 \in states \wedge q2 \in states$
$\bullet\ \exists s: \text{seq}\, X \mid s \in strings \bullet delta\ (q1, s) = q2$

Invariants: 1) The null string is an element of *strings*. 2) If the transition function takes a state and null string as input, then it produces the same state. 3) For each $(s, (\langle a \rangle \circ u))$, where s is state, a an alphabet and u is a string, the *delta* function is defined as: $delta\ (s, (\langle a \rangle \circ u)) = delta\ ((delta(s, \langle a \rangle)), u)$. 4) For any two states q1 and q2, there exists a string s such that: $delta(q1, s) = q2$.

4.2 Homomorphism and its Variants

The word homomorphism means "same shape" and is an interesting concept because a similarity of structures can be verified by it. It is a structure in abstract algebra which preserves a mapping between two algebraic structures, for example, monoid, groups, rings, vector spaces. Now we give formal specification of it and its variants over strongly connected automata. In [22], Ito M. has given a concept of homomorphism and its variants over algebraic automata.

Let SCEA1 = $(Q1, \sum 1, \delta 1)$ and SCEA2 = $(Q2, \sum 2, \delta 2)$ be two strongly connected automata, and let ρ be a mapping from Q1 into Q2. If $\rho\,(\delta 1(q, x)) = \delta 2(\rho(q), x)$ holds for any $q \in Q1$ and $x \in \sum 1$, then ρ is called a homomorphism from Q1 to Q2. The above pair of schemas is represented by $\Delta SCEA$ in Z which shows a relationship between SCEA1 and SCEA2. A formal definition of homomorphism from SCEA1 into SCEA2 in terms of a schema is given below. It consists of two components $\Delta SCEA$ and *row*. The variable *row* is a mapping from *Q*1 into *Q*2. The sets *Q*1 and *Q*2 are used for states of *SCEA*1 and *SCEA*2 respectively.

Homomorphism

$\Delta SCEA$
$row: Q \rightarrow Q$

$\forall q: Q; s: \text{seq}\, X \mid q \in states \wedge s \in strings$
$\bullet\ row\ (delta\ (q, s)) = delta'\ ((row\ q), s)$

Invariants: 1) For every q in set of states and s in set of strings of the first automata, if the mapping *row* satisfies the following condition then it conforms a homomorphism from automata SCEA1 into SCEA2:

$$row\,(deltal(q,s)) = delta2\,((rowq), s)\,.$$

If SCEA1 = SCEA1 in the homomorphism then it is called an endomorphism. The mapping *row* is defined from set of states *S* into itself. We have induced formal definition of endomorphism from the definition of homomorphism because it is a special case of it.

Endomorphism

$\Xi SCEA$
$row: Q \rightarrow Q$

$\forall q: Q; s:$ seq $X \mid q \in states \wedge s \in strings$
$\bullet$ $row\,(delta\,(q, s)) = delta\,((row\,q), s)$

Invariants: 1) For every state q and string s, if the mapping *row* satisfies the condition: $row\,(deltal(q,s)) = delta2\,((rowq), s)$, then it conforms an endomorphism between *SCEA* into itself.

Let us define the bijection over two given sets. Let A and B are two nonempty sets. A mapping π from A to B is called one to one if different elements of A have different images in B. That is $\forall$ a1, a2 ∈A; b ∈B • π (a1) = b and π(a2) = b ⇒ a1 = a2. The mapping π is called onto if each element of B is an image of some element of A. If a mapping is one to one and onto then it is called bijective. If the mapping defined in case of homomorphism is bijective from algebraic automata SCEA1 to SCEA2 then it is called an isomorphism and the automata are said to be isomorphic. Now we give a formalism of isomorphism from SCEA1 to SCEA2 using the schema given below. For this purpose, we simply define the required constraints of bijection over the homomorphism and it results an isomorphism.

Isomorphism

$\Xi Homomorphism$

$\forall q1, q2: Q1; q: Q2 \mid q1 \in states \wedge q2 \in states \wedge q \in states'$
$\bullet$ $(q1, q) \in row \wedge (q2, q) \in row \Rightarrow q1 = q2$
ran $row = states'$

Invariants: 1) For all q1, q2 in set of states of SCEA1 and q in set of states of SCEA2, if images of q1 and q2 are same under the mapping row then the elements q1 and q2 must be same. 2) Each element of the set of states of automata SCEA2 is an image of some element of automata SCEA1 under the mapping *row*.

If SCEA1 = SCEA2 then an isomorphism becomes an automorphism. Its formal description is given below along with its invariants which are not explained here because it is a repetition as given above in the schema *Isomorphism*.

Automorphism

$\Xi Endomorphism$

$\forall q1, q2, q: Q \mid q1 \in states \wedge q2 \in states \wedge q \in states$
$\bullet$ $(q1, q) \in row \wedge (q2, q) \in row \Rightarrow q1 = q2$
ran $row = states$

4.3 Formalizing Endomorphisms

Let G be a nonempty set. The structure (G,*) under binary operation * is monoid if 1) $\forall$ x, y ∈G, x*y ∈G, 2) $\forall$ x, y, z ∈G, (x*y)*z = x*(y*z), that is associative property is satisfied, 3) $\forall$ x ∈G, there exists an e ∈G such that x*e = e * x = x, e is an identity of G.

Let A be a automaton and E (A) = a set of all the endomorphisms over an automaton A. It is proved in [21] that E (A) forms a monoid which is an algebraic structure as defined in Section 1. To formalize it, two variables are assumed. The first one is a set of all endomorphisms which is of type of power set of *Endomorphism*. The second one is a binary operation which is denoted by the variable *boperation*. It takes two endomorphisms as input and produces a new endomorphism as an output. The formal definition of the above structure is given below.

Invariants: 1) This property defines binary operation over the set of endomorphisms. 2) Associative is verified here in this property. 3) This property ensures the existence of an identity element in the collection of endomorphisms.

EndomorphismsSCEA

endomorphisms: ℙ *Endomorphism*
boperation: Endomorphism × *Endomorphism* → *Endomorphism*

$\forall e1, e2:$ *Endomorphism* $\mid e1 \in endomorphisms \wedge e2 \in endomorphisms \bullet \exists e3:$
Endomorphism $\mid e3 \in endomorphisms \bullet boperation\,(e1, e2) = e3$
$\forall e1, e2, e3:$ *Endomorphism* $\mid e1 \in endomorphisms \wedge e2 \in endomorphisms \wedge e3 \in$
$endomorphisms \bullet boperation\,((boperation\,(e1, e2)), e3) = boperation\,(e1,(boperation\,(e2, e3)))$
$\forall e:$ *Endomorphism* $\mid e \in endomorphisms \bullet \exists ee:$ *Endomorphism* $\mid ee \in endomorphisms$
$\bullet$ $boperation\,(e, ee) = e \wedge boperation\,(ee, e) = e$

4.4 Formalizing Automorphisms

The algebraic structure (G, *) is called a group if 1) it is a monoid and 2) for any element x in the set G, there exists an element y in G such that x*y = y*x = e. That is the inverse of each element of G exists. Now let us suppose that G (A) = set of all the automorphisms over the algebraic automata A. For its formal specification, three variables are assumed. The first one is a set of all automorphism which is of type of power set of *Automorphism*. The second one is an identity element. And the last one is binary operation denoted by *boperation*. It takes two automorphisms as input and produces a new automorphism as an output.

AutomorphismsSCEA

$automorphisms: \mathbb{P}\ Automorphism$
$ae: Automorphism$
$boperationa: Automorphism \times Automorphism \rightarrow Automorphism$

$\forall a1, a2: Automorphism \mid a1 \in automorphisms \land a2 \in automorphisms \bullet \exists a3: Automorphism \mid a3 \in automorphisms \bullet boperationa\ (a1, a2) = a3$
$\forall a1, a2, a3: Automorphism \mid a1 \in automorphisms \land a2 \in automorphisms \land a3 \in automorphisms \bullet boperationa\ ((boperationa\ (a1, a2)), a3) = boperationa\ (a1, (boperationa\ (a2, a3)))$
$\forall a: Automorphism \mid a \in automorphisms \bullet boperationa\ (a, ae) = a \land boperationa\ (ae, a) = a$
$\forall a: Automorphism \mid a \in automorphisms \bullet \exists ai: Automorphism \mid ai \in automorphisms \bullet boperationa\ (a, ai) = ae \land boperationa\ (ai, a) = ae$

Invariant: The last one property verifies existence of inverse of each element in set G (A). The first three properties are same as in case of endomorphisms.

4.5 Proof of Equivalence

In this section, a formal proof of equivalence of endomorphisms and automorphisms is described. That is we have to verify that the set of all endomorphisms and set of all automorphisms over strongly connected algebraic automata are same. This verification is done in terms of a schema named as *Equivalence*. There are three inputs to this schema which are *Endomorphism*, *EndomorphismsSCEA* and *AutomorphismsSCEA*. At first, it is described that set of all endomorphisms are bijective over the strongly connected automata. Then it is verified that the number of elements must be same in both of the endomorphisms and automorphisms.

Equivalence

$\Xi Endomorphism$
$\Xi EndomorphismsSCEA$
$\Xi AutomorphismsSCEA$

$\forall e: Endomorphism \mid e \in endomorphisms \bullet \forall q1, q2, q: Q \mid q1 \in states \land q2 \in states \land q \in states \bullet (q1, q) \in e\ .\ row \land (q2, q) \in e\ .\ row \Rightarrow q1 = q2$
$endomorphisms \in \mathrm{dom}\ \# \land automorphisms \in \mathrm{dom}\ \#$
$\# endomorphisms = \# automorphisms$

5. Conclusions

The main objective of this paper was proposing an integration of some fundamental concepts in strongly connected automata and Z notation. To achieve this objective, first we described formal specification of strongly connected automaton. Then a relationship was identified and proposed between automata and Z structures. Next we described two important concepts of homomorphism and isomorphism between algebraic structures. Extended forms over strongly connected automata were formalized for these structures. Finally, a formal proof of equivalence between endomorphisms and automorphisms over strongly connected automata was described. It is to be mentioned here that preliminary results of this research were presented in [39].

Why and what kind of integration is required, were two basic questions in our mind before initiating this research. Strongly connected algebraic automaton is best suited for modeling behavior while Z is an ideal one used describing state space of a system. This distinct in nature but supporting behavior of Z forces us to integrate it with strongly connected algebraic automata.

An extensive survey of existing work was done before initiating this research. Some interesting work [40,41,42] in addition to given in Section 2 was found but our work and approach are different because of abstract and conceptual level integration of Z and automata. We believe that this work will be useful in development of integrated tools increasing their modeling power. It is to be mentioned that most of the researchers discussed here have either taken some examples in defining integration of approaches or addressed only some aspects of these approaches. Further, there is a lack of formal analysis which can be supported by computer tools. Our work is different from others because we have given a generic

approach to link Z and algebraic automata with a computer tool support.

Our idea is original and important because we have observed after integrating that a natural relationship exists there. This work is important because formalizing graph based notation is not easy as there has been little tradition of formalization in it due to concreteness of graphs [43]. Our work is useful for researchers interested in integration of approaches. We believe that this research is also useful because it is focused on general principles and concepts and this integration can be used for modeling systems after required reduction. Formalization of some other concepts in algebraic automata is under progress and will appear soon.

REFERENCES

[1] Hall, "Correctness by construction: Integrating formality into a commercial development process," LNCS, Springer, Vol. 2391, pp. 139−157, 2002.

[2] J. Burgess, "The role of formal methods in software engineering education and industry," Technical Report: CS-EXT-1995-045, University of Bristol, UK, 1995.

[3] A. L. Gwandu and D. J. Creasey, "The importance of formal specification in the design of hardware systems," School of Electron & Electrical Engineering, Birmingham University, UK, 1994.

[4] H. A. Gabbar, "Fundamentals of formal methods," Modern Formal Methods and Applications, Springer, 2006.

[5] A. Boiten, *et al.*, "Integrated formal methods (IFM04)," Springer, 2004.

[6] J. Davies and J. Gibbons, "Integrated formal methods (IFM07)," UK, Springer, 2007.

[7] J. Romijn, G. Smith, and J. V. D. Pol, "Integrated formal methods (IFM 05)," Springer, 2005.

[8] K. Araki, A. Galloway, and K. Taguchi, "Integrated formal methods (IFM99)," Springer, 1999.

[9] R. Bussow and W. Grieskamp, "A modular framework for the integration of heterogeneous notations and tools," Proceedings of the 1st International Conference on Integrated Formal Methods, UK, Springer, pp. 211−230, 1999.

[10] W. Grieskamp, T. Santen, and B. Stoddart, "Integrated formal methods (IFM 2000)," Germany, Springer, 2000.

[11] T. B. Raymond, "Integrating formal methods by unifying abstractions," LNCS, Springer, Vol. 2999, pp. 441−460, 2004.

[12] J. S. Dong, R. Duke, and P. Hao, "Integrating object-Z with timed automata," in Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, pp. 488−497, 2005.

[13] S. Dong, *et al.*, "Timed patterns: TCOZ to timed automata," in the 6th IEEE International Conference on Formal Engineering Methods, USA, 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, 2000.

[16] X. He, "Pz nets a formal method integrating Petri nets with Z," Information & Software Technology, Vol. 43, No.1, pp. 1–18, 2001.

[17] M. Heiner and M. Heisel, "Modeling safety critical systems with Z and Petri nets," International Conference on Computer Safety, Reliability and Security, LNCS, Springer, Vol. 1698, pp. 361–374, 1999.

[18] H. Leading and J. Souquieres, "Integration of UML and B specification techniques: Systematic transformation from OCL expressions into B," in the Proceedings of Asia-Pacific Software Engineering Conference (APSEC 02), Australia, 2002.

[19] H. Leading and J. Souquieres, "Integration of UML views using B notation," in the Proceedings of Workshop on Integration and Transformation of UML Models, Spain, 2002.

[20] W. Wechler, "The concept of fuzziness in automata and language theory," Akademic-Verlag, Berlin, 1978.

[21] N. M. John and S. M. Davender, "Fuzzy automata and languages: Theory and applications," Chapman & HALL, 2002.

[22] M. Ito, "Algebraic theory of automata and languages," World Scientific Publishing, 2004.

[23] K. Kaynar and N. Lynchn, "The theory of timed I/O automata," Morgan & Claypool Publishers, 2006.

[24] C. Godsil and G. Royle, "Algebraic graph theory," Springer, 2001.

[25] Z. Aleksic, "From biology to computation," IOS Press Publishers, 1993.

[26] L. B. Kier, P. G. Seybold, and C. K. Cheng, "Modeling chemical systems using cellular automata," Springer, 2005.

[27] T. Ellman, "Specification and synthesis of hybrid automata for physics-based animation," Automated Software Engineering, Vol. 13, No. 3, pp. 395–418, 2006.

[28] D. Conrad and B. Hotzer, "Selective integration of formal methods in development of electronic control units," in Proceedings of 2nd International Conference on Formal Engineering Methods, Vol. 9, No. 11, pp.144–155, 1998.

[29] M. Brendan and J. S. Dong, "Blending object-Z and timed CSP: An introduction to TCOZ," in Proceedings of the 1998 International Conference on Software Engineering, Vol. 19, No. 25, pp. 95–104, 1998.

[30] J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," Acta Informatica, Springer Berlin, pp. 27–52, 2004.

[31] A. T. Nakagawa, *et al.*, "Cafe an industrial-strength algebraic formal method," Elsevier Science & Technology, 2000.

[32] J. M. Spivey, "The Z notation: A reference manual," Prentice Hall, 1989.

[33] J. M. Wing, "A specifier: Introduction to formal methods," IEEE Computer, Vol. 23, No. 9, pp. 8–24, 1990.

[34] J. Woodcock and J. Davies, "Using Z: Specification, refinement and proof," Prentice Hall International, 1996.

[35] J. A. Anderson, "Automata theory with modern applications," Cambridge University Press, 2006.

[36] L. L. Claudio, *et al.*, "Applications of finite automata representing large vocabularies," Software Practice & Experience, Vol. 23, No. 1, pp. 15–30, 1993.

[37] Y. V. Moshe, "Nontraditional applications of automata theory," Theoretical Aspects of Computer Software, 1994.

[38] D. I. A. Cohen, "Introduction to computer theory," 2nd Edition, John Wiley & Sons Inc., 1996.

[39] N. A. Zafar, A. Hussain, and A. Ali, "Formal proof of equivalence in endomorphisms and automorphisms over strongly connected automata," International Conference on Computer Science and Software Engineering (CSSE 2008), Wuhan, China, 2008.

[40] D. P. Tuan, "Computing with words in formal methods," University of Canberra, Australia, 2000.

[41] J. P. Bowen, "Formal specification and documentation using Z: A case study approach," International Thomson Computer Press, 1996.

[42] S. A. Vilkomir and J. P. Bowen, "Formalization of software testing criterion," South Bank University, London, 2001.

[43] C. T. Chou, "A formal theory of undirected graphs in higher order logic," in Proceedings of 7th International Workshop on Higher Order Logic, Theorem Proving and Application, LNCS, Vol. 859, pp. 144–157, 1994.

