International Journal of Engineering Sciences Paradigms and Researches (IJESPR)
Volume 46, Issue 01, Quarter 04 (2017)
An Indexed, Referred and Impact Factor Journal
ISSN: 2319-6564
www.ijesonline.com

Formal Specification Language

Amal A. Mirghani[1], Nahid A. Ali[2] and Abdelrasoul Y. Ibrahim[3]

[1]Faculty of Computer Science and Information Technology,
Sudan University of Science and Technology
Khartoum, Sudan
aamy_22@hotmail.com

[2]Faculty of Computer Science and Information Technology,
Sudan University of Science and Technology
Khartoum, Sudan
nahidahmedali@hotmail.com

[3]Faculty of Computer Science and Information Technology,
Sudan University of Science and Technology
Khartoum, Sudan
abdosh67@hotmail.com

Publishing Date: December 20, 2017

Abstract

The specifications studied so far are adequate for specifying programs that take an input and map it onto an output. However, they inadequate to represent programs whose response depends not only on their input, but on their internal state. So for specifications should have two key attributes: formality and abstraction. Axiomatic representation represents the relation of a specification by means of an inductive notation to achieve simplicity, formality, and abstraction. This notation includes Axioms and Rules. This paper represents *Alneelain* specification language that checks the syntax of Abstract Data Types specification based on Axiomatic representation.

Keywords: *Alneelain Specification Language, Axiomatic Specification, Abstract Data Types.*

1. Introduction

“The specification of a software product is a description of the functional requirements that the product must satisfy”. The word Specification refers to both a process and a product. As a product, the specification plays two key roles: first, specification is the contract that binds the user requirements and the designer. Second, it is the primary working document for the designer. As a process, the specification takes place in two steps: the specification generation step, when the specification is progressively constructed from the user requirements; the specification validation step, when the specification is matched against redundant requirements data elicited from the user [1]. Whereas specifications studied so far are adequate for specifying programs that take an input and map it onto an output, they are inadequate to represent programs whose response depends not only on their input, but on their internal state; the subject of this section is to explore ways to specify such systems.

This paper will discuss how to build Alneelain compiler which checks the syntax of Abstract Data Types ADT Specification. A Compiler “is a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language)” [2] [3]. Compiler important role is to report any errors in the source program that it detects during the translation process [3]. This work is part of a global project, where my colleague Abdelrasoul Yahia works on specifications generation and validation, and my colleague Nahid Ahmed works on specification verification and my work is on specification testing, and can be used as an educational tool in an integrated programming environment to teach students how to define and validate a formal specification, verify and test them against the specifications.

2. Checking Alneelain specification language

Alneelain is Specification Language that represents the relation of a specification by means of an inductive notation, with induction on the structure of the input history. Figure 1 blew illustrates the steps of checking Alneelain Specification Language, for abstract data types and will take stack as example.

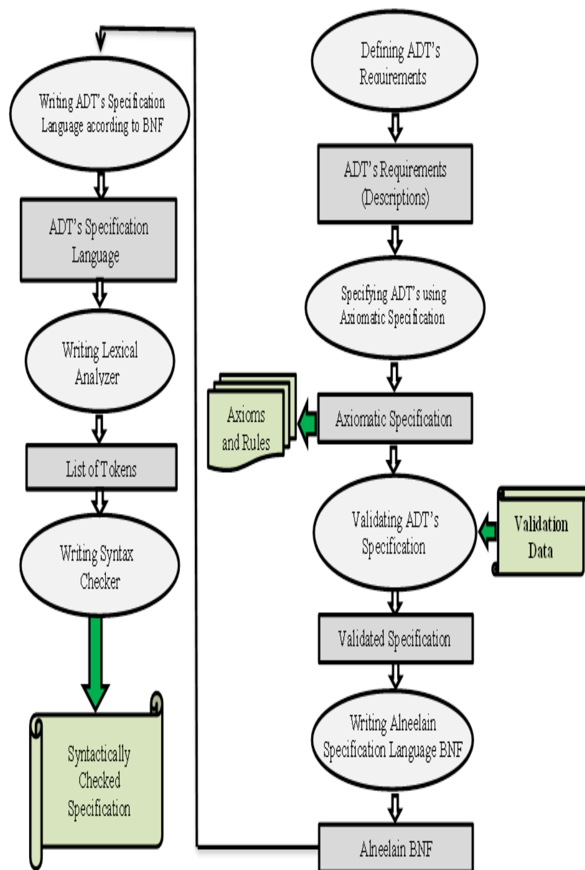

Figure 1: Checking Alneelain specification language

2.1 Defining ADT's Requirements

A stack is a data type that is used to store items (through operation push()) and to remove them in reverse order (through operation pop()). Then returns the most recently stored item that has not been removed (through operation top()), operation size() returns the number of stored items and operation empty() tells whether the stack has any items stored; and finally operation init() reinitializes the stack to an initial situation [4] [5].

The specification of abstract data types represented by means of three attributes:

- An input space, say X that represents the symbols that may be fed into the ADT as inputs. For a stack ADT, for example, these would be:

  X = {init, pop, top, size, empty} ∪ {push} × itemtype.

  From this set, we build set H of sequences of elements of X, and we refer to H as the set of input histories, or input sequences.

- An output space, which represents the set of symbols that the ADT returns on output. For the stack ADT, this would be:

  Y = itemtype ∪ integer ∪ Boolean ∪ {error}.

- A relation (often a function) from H to Y, which associates an output for each input sequence. For the stack ADT, this relation would include pairs such as:

Stack(init.push(a).push(b).top.push(c).pop.top) = b

Stack(init.pop.push(a).pop.push(a).pop.push(a) .size) = 1

2.2 Specifying ADT's using Axiomatic Specification

In order to represent specifications in closed form, we use an axiomatic notation that includes:

- Axioms, which represent the behavior of the ADT for elementary input sequences. As an example, consider the following axiom for the stack specification:
  1. *Top axioms.*
     a. stack(init.top) = error.
     b. stack(init.h.push(a).top) = a.
  2. *Size axiom.*
     a. stack(init.size) = 0.
  3. *Empty axioms.*
     a. stack(init.empty)=true.
     b. stack(init.push(a).empty)=false.

- Rules, which define the behavior of the ADT for complex input sequences as a function of their behavior for simpler input sequences.

As an example, consider the following rule for the stack specification:

1. *Init rule*:
   stack(h.init.h')=stack(init.h').
   Init reinitializes the stack state: even if there a history h prior to init or not.
2. *Init Pop rule*:
   stack(init.pop.h) = stack(init.h).

Pop has no impact on an empty stack.

3. *Push pop rule*:
   stack(init.h.push(a).pop.h+) = stack(init.h.h+).

A pop operation cancels the push before it.

4. *Size rule*:

stack(init.h.push(a).size)=1+stack(init.h.size)
Push operation raises the size of the stack by 1 because the stack size is not restricted.

5. *Empty rules*
   a. stack(init.h.push(a).h'.empty)==> stack(init.h.h'.empty).

   If, despite having operation push(a) in its history, the stack is empty, then a fortiori it would empty without push(a).

   b. stack(init.h.empty)==> stack(init.h.pop.empty).

   If the stack is empty, then a fortiori it would be empty if an extra pop operation was performed in its past history.

6. *V-operation rules*
   a. stack(init.h.top.h+)=stack(init.h.h+)
   b. stack(init.h.size.h+)=stack(init.h. h+)
   c. stack(init.h.empty.h+)= stack(init.h.h+).

   V-operations have no impact on the future behavior of the stack.

Where *h* is an arbitrary input history and *h+* is an arbitrary non null input history.

2.3 Validating ADT's Specification

It is important to validate specifications for completeness and minimality, and to invest the necessary resources to this effect before proceeding with subsequent phases of the software lifecycle. The only way to ensure a measure of confidence in the validation of the specification is to separate the team that generates the specification from the team that validates it. To this effect, the study proposes the following two-team, two-phase approach [4] [6] as in table 1 blew

Table 1: Validation Approach

Phase \ Activity	Specification Generation	Specification Validation
Specification Generation	Generating the Specification from sources of requirements	Generating validation data from the same sources of requirements
Specification Validation	Updating the specification according to feedback from the validation team	Testing the specification against the validation data generated above

For the sake of simplicity, this work focuses solely on completeness. And while writing these specifications, an independent verification and validation team is generating formulas of the form
stack(h) = y
For different values of h and y, on the grounds that whatever the team write in our specification should logically imply these statements. Then the validation step consists in checking that the proposed formulas can be inferred from the axioms and rules of our specification. If they do, then the researchers can conclude that our specification is complete with respect to the proposed formulas; if not, then he/she needs to check with the verification and validation team to see whether our specification our specification is incomplete, or perhaps the validation data is erroneous.
For the sake of illustration, the researchers check whether our specification is valid with respect to the formulas written in section 2.1 as sample input /output pairs of our stack specification.

- **V_1**: stack(pop.init.top.pop.push(a).size. push(b).top.pop.push(c).top.pop.size.top)=**a**
- V2: stack(push(2).init.init.pop.push(3).top .size.push(2).push(5).top.pop.push(3). size)=3

For **V_1**, one find:
= {by virtue of the init rule}
stack(init.top.pop.push(a).size.push(b).top.pop.push(c).top.pop.size.top)
= {by virtue of the V-op rules}

stack(init.pop.push(a).push(b).pop.push(c).pop.top)
= {by virtue of the push-pop rule, applied twice}
stack(init.pop.push(a).top)
= {by virtue of the second top axiom, with h = < pop >}
= a

For $\mathbf{V_2}$, one find:
= {by virtue of the init rule}
stack(init.pop.push(3).top.size.push(2).push(5).top.pop.push(3).size)
= {by virtue of the V-op rules}
stack(init.pop.push(3).push(2).push(5). pop.push(3).size)
= {by virtue of the push-pop rule}
stack(init.pop.push(3).push(2). push(3).size)
= {by virtue of the size rule, with h = <pop.push(3).push(2)>}
1 + stack(init.pop.push(3).push(2).size)
= {by virtue of the size rule, with h = <pop.push(3)>}
1 + 1 + stack(init.pop.push(3). size)
= {by virtue of the size rule, with h = <pop>}
1 + 1 + 1 + stack(init.pop.size)
= {by virtue of the init-pop rule}
1 + 1 + 1 + stack(init.size)
= {by virtue of size axiom}
1 + 1 + 1 + 0
= {arithmetic}
= 3

2.4 Alneelain specification's language Backus Naur Form

Backus Naur Format (BNF) "is a formal metalanguage for describing language syntax" [7] [8]. BNF language Different from English language because BNF is not open to any one interpretations, and There is only one method to read its description [7]. Table 2 blew list the notation and their meaning [9].

Table 2: BNF Notation

BNF Notation	Meaning
< >	Non-terminal symbol
::=	Defining symbol
|	Alternative
[ ]	Optional symbols
{ }	Grouping

The following is the Alneelain main function According to the mentioned notation.
<alneelain>::= <header>**;** <body>
endspecification

2.5 Alneelain specification language according to the BNF

The following is Stack specification language as example.

```
specification Stack;
  constant
        x = 5;
        type
              itemtype : char;
        input
                vop top: itemtype ,
                vop size: integer ,
                vop empty: boolean
                oop init, pop, push(char)
        endinput;
        output
            char ∧ Boolean ∧ integer ∧ error
        endoutput;
        variable
             a: char ,
            h: inputstar ,
            hprime: inputstar ,
            hplus: inputplus ;
        axioms
          axiom topAxiom:
                Stack(init.top) = error &
                Stack(init.h.push(a).top) = a ,
          axiom sizeAxiom:
                Stack(init.size) = 0,
          axiom emptyAxiom:
                Stack(init.empty)=true &
                Stack(init.push(a).empty)=
                false
        endaxioms;
        rules
            rule initRule:
                Stack(h.init.hprime)=Stack(init
                . hprime) ,
            rule initpopRule:
                Stack(init.pop.h)            =
                Stack(init.h) ,
            rule pushpopRule:
                Stack(init.h.push(a).pop.hplus)
                = Stack(init.h.hplus) ,
           rule sizeRule:
```

```
            Stack(init.h.push(a).size)    =1+
            Stack(init.h.size) ,
        rule emptyRule:
            Stack(init.h.push(a).hprime.
            empty)=>
            Stack(init.h.hprime.empty)&
            Stack(init.h.empty)           =>
            Stack(init.h.pop.empty) ,
        rule vopRule:
            Stack(init.h.top.hplus)=Stack
            (init.h.hplus) &
            Stack(init.h.size.hplus)=Stack
            (init.h.hplus) &
      Stack(init.h.empty.hplus)=Stack(init.h.
      hplus)
    endrules;
endspecification
```

2.6 Writing Lexical Analyzer

Lexical analysis is the first phase of a compiler construction stages. Also called a lexical scanner because, it scans the input string without backtracking (i.e. by reading each symbol once and processing it correctly) [9]. Lexical analysis attempts to isolate a lexical token. Lexical token “is a string of input characters which is taken as a unit and passed on to the next phase of compilation” [2]. The following Table 3 illustrates the tokens.

Table 3: List of Tokens

Token type	Tokens
Keywords	endspecification, specification, constant, type, input, endinput, vop, oop, output, endoutput, variable, axioms, endaxioms, axiom, rules, endrules, rule.
Special characters	, ‘ . ‘ ; ‘ (‘) ‘ : ‘ ^ ‘ & ‘ { ‘ } ‘ -
Operators	= ‘ + ‘ =>
Relation operators	> ‘ < ‘ >= ‘ <=
Digits	0-9
Identifiers	[a-z A-Z][a-z A-Z]*

2.7 Writing Syntax Checker

The syntax analysis phase is often called the parser. The input to this phase consists of a stream of tokens produce by the lexical analysis phase. Then the tokens checked for proper syntax, i.e. the compiler checks to make sure the statements and expressions are correctly formed. When the compiler encounters such an error, it should put out an informative message for the user [2] [3]. The method used for implementing the parser is Recursive Descent parsers, in this method, the parser written using a procedure-oriented language, such as Pascal or C. A function is written for each nonterminal in the grammar. The purpose of this function is to scan a portion of the input string until an example of that nonterminal has been read. By an example of a nonterminal, we mean a string of terminals or input symbols which can be derived from that nonterminal. This is done by using the first terminal symbol in each rule to decide which rule to apply. The function then handles each succeeding symbol in the rule; it handles nonterminals by calling the corresponding functions, and it handles terminals by reading another input symbol [2] [3].

The pseudo code for syntax checker main function is looks like:

```
// <alneelain> ::=  <header> ; <body>
endspecification

FUNCTION alneelain()
{
    SET diagnosis=true
    FUNCTION header()
    FUNCTION checktoken(semicolon)
    FUNCTION body()
    FUNCTION checktoken (endspecification)
    IF (diagnosis = true)
        PRINT “Syntactically correct”
    ELSE
        PRINT “Syntactically incorrect”
```

International Journal of Engineering Sciences Paradigms and Researches (IJESPR)
Volume 46, Issue 01, Quarter 04 (2017)
An Indexed, Referred and Impact Factor Journal
ISSN: 2319-6564
www.ijesonline.com

3. The User Interface

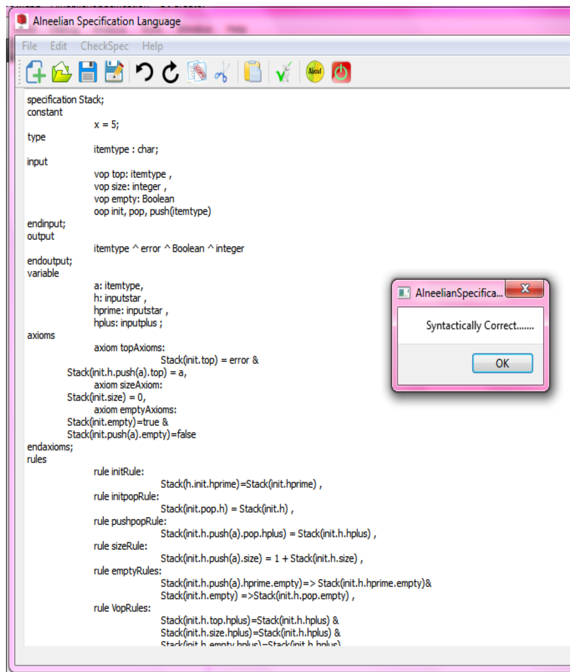

Figure 2: Checking Stack Specification

The Alneelain Specification Language allows user to create a file that contains a specification of any abstract data type. Figure 2 shows Alneelain specification language interface. As illustration, If the user had the specification file saved in specific folder, he/she can loads it from that folder throw open menu, or he/she can add new file and write the specification. The specification is then executed by clicking on CheckSpec. The Alneelain Specification Language results is a message shows if the ADT specification is correct by typing "Syntactically correct", or gives a detailed message of the place of error if it is Syntactically incorrect as in **Error! Reference source not found.**.

4. Conclusions

This paper discusses the design of a new specification language based on axiomatic Specification called *Alneelain*. It shows the steps of designing Alneelain specification language. From defining ADT's requirements, then specifying this requirements using axiomatic specification, and after that shows how to validate the ADT's specification, then using BNF notation to describe the syntax of the specification. The last step the syntax checker checks the tokens which scanned from the lexical analyzer for proper syntax. Then according to this language the user can check his specification if it is syntactically correct or not.

Acknowledgments

We would like to express our deep gratitude to our supervisor Prof. Ali Mili For his persistent constant guidance and wise counsel. Also, we appreciate the assistance offered to us by Mr. Mugtaba Ali.

References

[1] Noureddine, Bourdriga; Mili, Ali; Zalila, R; Mili, Fatma;, "Relational model for the specification of data types," computer languages, vol. 17, no. 2, pp. 101-131, 1992.
[2] Seth D. Bergmann, Compiler Design: Theory, Tools, and Examples C/C++ Edition, 2010.
[3] Aho, Alfred V.; Lam , Monica S.; Sethi, Ravi; Ullman, Jeffrey D.;, compiler principles techniques and tools, 2nd ed.: Addison wesley, 2007.
[4] Mili, Ali; Tchier, Fairouz, Software Testing: Concepts and Operations: John Wiley & Sons, 2015.
[5] Michael S. Jenkins, Abstract Data Types in Java: McGraw-Hill School Education Group, 1997.
[6] Tchier, Fairouz; Rabai, Latifa Ben Arfa; Mili, Ali, "Putting Engineering into Software Engineering: Upholding Software Engineering Principles in the Classroom," Computers in Human Behavior, vol. 48, pp. 245-254, 2015.
[7] Kent D. Lee, Programming Languages/ an active learning approach: Springer US, 2008.
[8] Fischer, Charles N; LeBlanc, Richard J; Cytron, Ronald K;, Crafting a compiler.: Addison-Wesley, 2009.
[9] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D., Introduction to Automata Theory, Languages, and Computation, 3rd ed.: Pearson, 2006.