An Automated Technique to Support Software Validation

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

[1, 2, 3]Sudan University for Science and Technology, Faculty of Computer Science and Information Technology, Khartoum, Sudan

Abstract: ***Validation of software specification is a fundamental issue to make sure that the specification is demonstrates all software requirements. As the specification is the starting phase of software development lifecycle processes; a validation consist essentially of stated attributes about the specification, the validation then provides that the specification satisfies the all those attributes. Such attributes are completeness, minimality, and simplest. Abstract data types are software which has functional attributes can be specified uses a behavioral specification. This specification is represented by axioms and rules. Alneelain specification language which is uses behavioral based specification is used to specify abstract data types in form of axiomatic specification. Alneelain validation tool is created to use to validate the specification of abstract data types.***

Keywords: Validation of Specification; Software Requirement Specification; Abstract Data Types; Behavioral Specification model; axiomatic representation; Alneelain Specification Language; Alneelain Validation Tool.

1. Introduction

A validation of a software specification is a fundamental issue which by means of ensuring that requirements specification in a scale specify what are supposed to be specified. A specification is the starting point of the development process. It has the same status as axioms of a mathematical theory. Even the simplest of validation activities can improve the quality of a draft standard but a well-planned [1]and systematic validation process will identify many technical inaccuracies and completeness that might otherwise have been remained in the specified document. There are many specification languagesare proposed VDM [2],Z [3] [4], Alloy [5],OCL [6], and B which is a model state-based specification language [7] as well as many their validation techniques. A behavioral based specification languagecalled *Alneelain* [8]Specification languagealso proposed which based on behavioral model [9]for specify abstract data types.

The aim of this paper is to introduce a new automated technique to help in software specification and validation of the specification itself prior to move to next phase in software engineering process. The study adopted *Alneelain* specification language that we presented in previous work [8], which was based on axiomatic specification. Abstract data types is a type of software were specified by Alneelain specification language and checked by both of its two components' lexical analyser and syntax checker. A validation tool has been designed to use to validate independent validation data of abstract data types.

The rest of this paper organized as follows: The proposed behavioral specification model that Alneelain language based is mentioned in section 2. An axiomatic representation that the behavioral model is depended on is explained in section 3, in addition to a sample of queue specification using Alneelain specification language is presented. Section 4 the interface of Alneelain specification language and is described and how to use it to specify a queue and its output that resulted after execution of the language. A proposed method for validation is illustrated in Section 5. Section 6 explained how to validate the specification of a queue manually. Section 7 explained the automated validation prove supported by the validation tool interface that does the validation. An example of a queue that has a series of operations is illustrated. And at last, conclusion will be included in section 8.

2. A behavioral specification model description

The proposed a specification model it captures the behavior of software systems that have an internal state [10]. Abstract data types (ADT's) are a typical software product with an internal state, and other types of software products also can fall under this category.Alneelain specification language is based on this model which uses axiomatic representation [9] [10]. This model is described by specification of queue as an example of abstract data type as follows:

1) **An input space**, say X, which includes all the input symbols that may be submitted to the ADT. For the Queue, we have
   $X = \{init, deq, front, rear, size, empty\} \cup \{enq\} \times itemtype$,
   Where itemtype is the data type of the items we wish to store in the queue. We distinguish between two types of input symbols:
   - Those that change the state of the ADT and produce no outputs; in the case of the stack these include,
     XO = {init, enq, deq}.
   - Those that do not change the state of the ADT but return an output value; in the case of the stack ADT, these include
     XV = {front, rear, size, empty}.
2) From the input space X we compute set H as the set of sequences of X, and we refer to H as the set of input histories, or input sequences.
3) **An out spaceY,** which includes all the possible outputs that the ADT may return when the last symbol of an input history is in XV. For the queue ADT, the output space Y may be defined as follows:

Y = itemtype $\cup$ integer $\cup$ Boolean $\cup$ {error}.

A Relation R from H to Y, which to each input history assigns an output. In the case of the queue ADT, this relation may include the following pairs:

queue(init.enq(a).enq(b).front.deq.size)=1
queue(init.enq(a).deq.enq(b).enq(c).rear)=c
queue(init.enq(a).enq(b).front.deq.empty)= false
queue(init.enq(a).enq(b).deq.deq.front) = error

3. Axiomatic Representation

The model is adopted an axiomatic notation, which represents the specification by prodding of convention on the reach of the input history:

- Axioms describe the behavior of the ADT for simple input histories,
- Rules relate the behavior of the ADT for simple input histories to its behavior for more complex/ longer input histories.

As an example, we show below how we represent the specification of the queue ADT in the axiomatic notation.

Axioms for the queue:

1) **Size axiom:**
   a. queue(init.size)= 0.
   The size of an empty queue is zero.
2) **Empty axioms:**
   a. queue(init.empty)= true
   b. queue(init.enq(a).empty)= false

An initial queue is empty. A queue in which an element has been enqueued is not empty.

3) **Front axioms:**
   a) queue(init.front)= error
   Invoking front on an empty queue returns an error.
   b) queue(init.enq(a).enq(_)*.front)= a
   Where enq(_)* designates an arbitrary number (including zero) of enq operations. Interpretation: Invoking front on a non-empty queue returns the first element enqueued.

4) **Rear axioms:**
   a) queue(init.rear)= error
   Invoking rear on an empty queue returns an error.
   b) queue(init.enq(_)*.enq(a).rear)= a
   Invoking rear on a non-empty queue returns the last element enqueued.

Rules of Queue:

1) **Init rule:**
queue(h.init.h') = queue(init.h')
The init operation reinitializes the queue, i.e. renders all past input history irrelevant.

2) **Init deq rule**
a) queue(init.deq.h) = queue(init.h)
A deq operation executed on an empty queue has no effect.

3) **Enq deq rule**
queue(init.enq(a).enq(_)*.deq.h+)=queue(init.enq(_)*.h+)
A deq operation cancels the first enq, by virtue of the FIFO policy of queues.

4) **Size rule:**
queue(init.h.enq(a).size) =1+ queue(init.h.size)
An enq operation increases the size of the queue by 1

5) **Empty rules:**
a. queue(init.h.enq(a).h'.empty) => queue(init.h.h'.empty)
b. queue(init.h.empty) => queue(init.h.deq.empty)
Removing an enq or adding a deq to the input history of a queue makes it emptier.

6) **VX-Operation rules:**
a) queue(init.h.front.h^{+}) = queue(init.h.h^{+})
b) queue(init.h.rear.h^{+}) = queue(init.h.h^{+})
c) queue(init.h.size.h^{+}) = queue(init.h.h^{+})
d) queue(init.h.empty.h^{+}) = queue(init.h.h^{+})

VX operations leave no trace of their passage; once they are serviced and another operation follows them, they are forgotten: whether they occurred or did not occur has no impact on the future behavior of the queue.

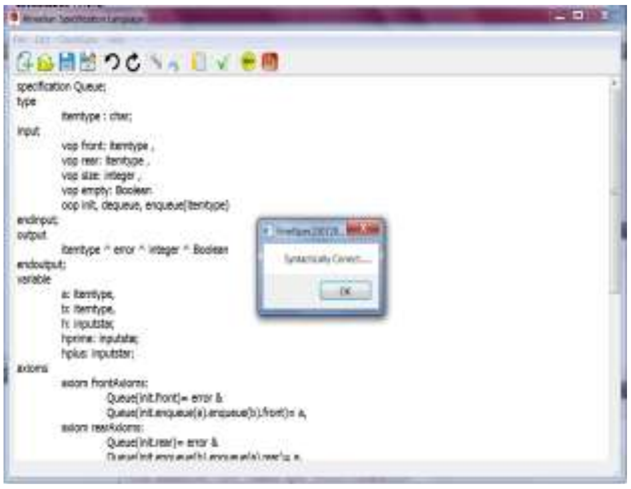
Alneelain specification language [8] is used to specify abstract data type and then the specification of abstract data types are checked into phases: Lexical analysis and syntax checker [8]. A following Fig 1 showed the specification of queue in Alneelain specification language. It includes constant, input, output, variables, axioms, and rules.

```
specification Queue;
constant
        x = 10;
type
        itemtype : char;
input
        vop front: itemtype ,
        vop rear: char ,
        vop size: integer ,
        vop empty: Boolean
        oop init, deq, enq(char)
endinput;
output
        char ^ Boolean ^  integer ^ error
endoutput;
variable
        a: char,
        b: char,
        h: inputstar,
        hprime: inputstar,
        hplus: inputstar;
axioms
        axiomfrontAxiom:
                    Queue(init.front)= error &
                    Queue(init.enq(a).enq(b).front)= a,
        axiomrearAxiom:
                    Queue(init.rear)= error &
                    Queue(init.enq(b).enq(a).rear)= a,
        axiomsizeAxiom:
                    Queue(init.size)= 0,
        axiomemptyAxiom:
                    Queue(init.empty)= true &
                    Queue(init.enq(a).empty)= false
     endaxioms;
     rules
          ruleinitRule:
                    Queue(h.init.hprime)                    =
Queue(init.hprime),
          ruleinitdeqRule:
                    Queue(init.deq.h) = Queue(init.h),
          ruleenqdeqRule:
                    Queue(init.enq(a).enq(b).deq.hplus)=
                    Queue(init.enq(b).hplus),
          rulesizeRule:
                    Queue(init.h.enq(a).size)=
Queue(init.h.size),
          ruleemptyRule:
                    Queue(init.h.enq(a).hprime.empty)=>
                    Queue(init.h.hprime.empty) &
                    Queue(init.h.empty)=>
                    Queue(init.h.deq.empty),
          ruleVopRule:
                    Queue(init.h.front.hplus)=
                    Queue(init.h.hplus)&
                    Queue(init.h.rear.hplus)=
                    Queue(init.h.hplus) &
                    Queue(init.h.size.hplus)=
                    Queue(init.h.hplus) &
                    Queue(init.h.empty.hplus)=
                    Queue(init.h.hplus)
     endrules;
     endspecification
```

Figure 1: Specification of queue using Alneelain language

4. Alneelain specification language lexical and syntax checkerinterface

To use the Alneelain specification language, we can open the language interface as shown in Fig 2 where the user can edits and writes the specification of the abstract data type according to language rules or, the user can open a file that already exists which was written in structureof the Alneelain language as shown in Fig 1, then the user check the specification by clicking the (√) sign or clicking on check command in checkSpec menu to show analert message confirming that The specification of the queue is syntactically correct according to the rules of the Alneelain language. An out file is created as a result of the checking process. This file consists only of the axioms and rules that belong to the queue. This file is used in the validation stage of the specifications. The Al-Neelain ValidationTool relies on axioms and rules to automatically simplify and reduce the user queries that have independent validation data.



```
specification Queue;
type
          itemtype : char;
input
          vop front: itemtype ,
          vop rear: itemtype ,
          vop size: integer ,
          vop empty: Boolean
          oop init, dequeue, enqueue(itemtype)
endinput;
output
          itemtype ^ error ^ integer ^ Boolean
endoutput;
variable
          a: itemtype,
          b: itemtype,
          h: inputstar,
          hprime: inputstar,
          hplus: inputstar;
axioms
          axiom frontAxioms:
                    Queue(init.front)= error &
                    Queue(init.enqueue(a).enqueue(b).front)= a,
          axiom rearAxioms:
                    Queue(init.rear)= error &
```

Figure 2: Al-Neelain Specification Language Interface

A log file is created as an output result. This output file shows all axioms and rules for that abstract data type, for example after checking a queue as shown in Fig 3.

```
frontAxioms:
          Queue(init.front)= error &
          Queue(init.enq(a).enq(b).front)= a,
rearAxioms:
          Queue(init.rear)= error &
          Queue(init.enq(b).enq(a).rear)= a,
sizeAxiom:
          Queue(init.size)= 0,
emptyAxioms:
          Queue(init.empty)= true &
          Queue(init.enq(a).empty)= false

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

initRule:
          Queue(h.init.hprime) = Queue(init.hprime),
initdeqRule:
          Queue(init.deq.h) = Queue(init.h),
enqdeqRule:
          Queue(init.enq(a).enq(b).deq.hplus) =
          Queue(init.enq(b).hplus),
sizeRule:
          Queue(init.h.enq(a).size) = 1 +
          Queue(init.h.size),
emptyRules:
          Queue(init.h.enq(a).hprime.empty) =>
          Queue(init.h.hprime.empty) &
Queue(init.h.empty) => Queue(init.h.deq.empty),
VopRules:
          Queue(init.h.front.hplus) = Queue(init.h.hplus) &
          Queue(init.h.rear.hplus) = Queue(init.h.hplus) &
          Queue(init.h.size.hplus) = Queue(init.h.hplus) &
          Queue(init.h.empty.hplus) = Queue(init.h.hplus)
```

Figure 3: A log file output of Queue after checking.

The output summarizes only axioms and rules that will later be input to the validation tool, where these axioms and rules are stored as basis for comparison process of simplification and reduction of the independent data entered by the user.

The results of user queries are shown after repeated execution of the rewriting operations. The operation compares the left-hand side of user's query with the rules that stored in the tool.If a matching is found, this meansthe left-hand sideis equal to the right-hand side of the specific rule after searching all the rules from the beginning. If no such rule was matching, the compare it takes place to all axioms. If one finds an intuitive one identical, the left side of the axiom is considered the final result.

5. Method of Validation process

Our simplest method is a part of a whole method and flow chart of designing Alneelain specification language that was introduced in [8]. Fig illustrate our model, this model can be added to model in [8] to be as a complete model for software specification and validation.

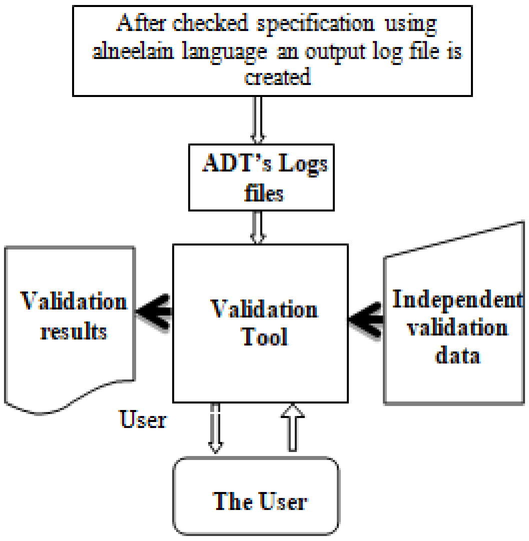

After checked specification using alneelain language an output log file is created
ADT's Logs files
Validation results
Validation Tool
Independent validation data
User
The User

Figure 4: Method of Validation process

6. Manual Validation

In the previous section we have written specifications of the two of ADT's, namely queue. How do we know that our specifications are valid, i.e. that they capture all the properties we want them to capture such as completeness and nothing else such as minimality? To bring a measure of confidence in the validity of these specifications, we envision a validation process, though by now we focus gradually on completeness; so our confidence in the validity of the specification increases. We imagine that while we are writing these specifications, an independent validation data was generated formulas of the form:

Queue(h)=y

For different values of *h* and *y*, on the grounds that whatever we 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 we can conclude that our specification is complete with respect to the proposed formulas; if not, then we need to check with the validation data which had been generated independently to see whether our specification is incomplete, or perhaps the validation data is erroneous.

For the sake of illustration, we check whether our specification is valid with respect to the formulas written as in Fig 3 as sample pairs of input and output of our queue specification.

1- queue(init.enq(a).enq(b).front.deq.size)=1
2- queue(init.enq(a).deq.enq(b).enq(c).rear)=c
3- queue(init.enq(a).enq(b).front.deq.empty)= false
4- queue(init.enq(a).enq(b).deq.deq.front) = error

Figure 5: Sample pairs of input and output validation data of queue

We can do a manual validation for this specification shown in fig 3; let us take the first formula

1- queue(init.enq(a).enq(b).front.deq.size)=**1**
By virtue of Vop rule:
queue(init.enq(a).enq(b).front.deq.size) =
queue(init.enq(a).enq(b).deq.size)
By virtue of enqdeqRule:
queue(init.enq(a).enq(b).deq.size) =
queue(init.enq(a).size)
By virtue of the size Rule
queue(init.enq(a).size)= 1+ queue(init.size)
By virtue of the size axiom
1+ 0 {Mathematically} is equal to**1**Qed.

2- queue(init.enq(a).deq.enq(b).enq(c).rear)=**c**
By virtue enqdeqRule:
queue(init.enq(a).deq.enq(b).enq(c).rear)=
queue(init.enq(b).enq(c).rear)
By virtue rear axiom:
queue(init.enq(b).enq(c).rear) = **c** Qed.

3- queue(init.enq(a).enq(b).front.deq.empty)= **false**
By virtue of Vop rule:
queue(init.enq(a).enq(b).front.deq.empty)=
queue(init.enq(a).enq(b).deq.empty)
By virtue of enqdeqRule:
queue(init.enq(a).enq(b).deq.empty)=
queue(init.enq(a).empty)
By virtue of the empty Axiom:
queue(init.enq(a).empty)= **false** Qed.

4- queue(init.enq(a).enq(b).deq.deq.front) = **error**
By virtue of enqdeqRule:
queue(init.enq(a).enq(b).deq.deq.front)=
queue(init.enq(a).deq.front)
By virtue of enqdeqRule:
queue(init.enq(a).deq.front) = queue(init.front)
By virtue of initAxiom:
queue(init.front) = **error** Qed.

7. Automated Validation Proving

A validation of specification is fall in the concepts of an automated theorem prover. Anautomatic theory focuses on the aspects of "Finding"[11]. Solution theorem prover provides means to validate formulas in the proportional and first-order logic. However, some other systems provide search procedures and decision-making procedures for languages and specific scope, such as linear[12],[13] or non-linear expressions[13] on integers or real numbers. In the other hand abstract data types are special case that have algebraic behavioral, they differ from the rest but they can have a similarity in such attributes. Alneelain Validation Tool is created to automatically use to validate independent validation data. The tool actually implement rewriting system algorithm[14]. The most important process in the implementation of the rewrite algorithm is that it receives the data in the form1:

Stack(push(a).init.pop.push(a).top.pop.size,

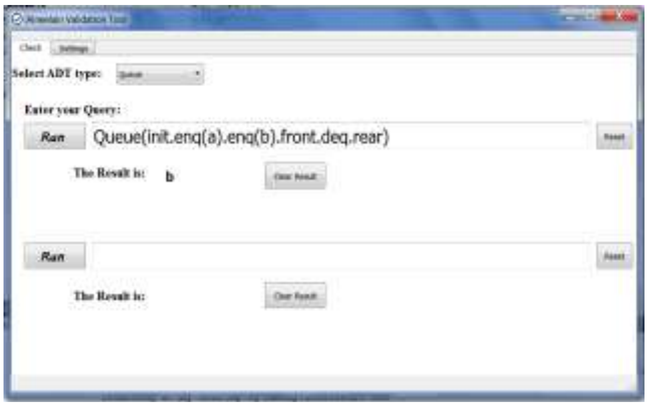
Then the tool simplifies it until it get 0,

Or takes data in the form2:

Stack(push(a).init.pop.push(a).top.pop.size)=0,

andsimplifies it until it finds 0.

Fig shows Alneelain Validation Tool interface and the validation of independent data in form of query of queue that have a series of operations


Select ADT type: Queue

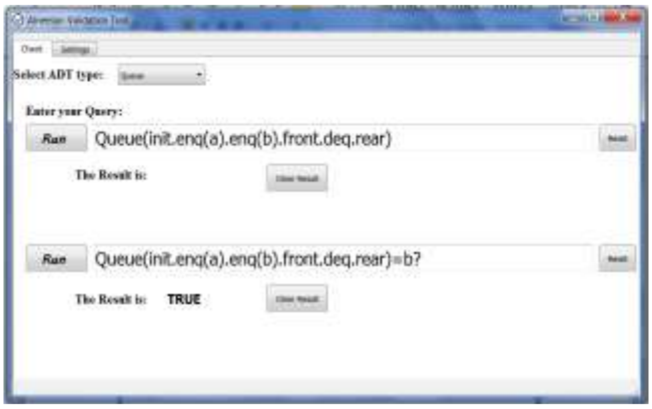
Enter your Query:

Queue(init.enq(a).enq(b).front.deq.rear)

The Result is: b

The Result is:

Figure 6a: Result of validation of queue query, form1.


Select ADT type: Queue

Enter your Query:

Queue(init.enq(a).enq(b).front.deq.rear)

The Result is:

Queue(init.enq(a).enq(b).front.deq.rear)=b?

The Result is: TRUE

Figure 6b: Result of validation of queue query, form2.

When we find that manual validation is identical to the automated method, the validation is goes to more significant, and increases our confidence that our specifications are absolutely correct.

8. Conclusion

Software requirement specification is a major process in software engineering lifecycle which involves eliciting requirements, classifying requirements, resolving conflicts, capturing requirements, and specifying them. A validation of specification is needed to ensure that the specification is valid against completeness, consistency prior to proceed the next phase, because any error arises in this phase it will affect all other subsequent phase.

The most two important works that this paper has introduced is design specification language called Alneelain which use to specify abstract data types and creation of a validation tool that is used to validate the specification, a good result of queue abstract data type was shown include both specification by the language and validation by the tool which support or aim and philosophy. However, there is a need for integration both the language and the tool to be as whole work as well as not only abstract data types but include similar to abstract data type such as mathematical formulas.

9. Acknowledgement

We would like to thank Dr. Ali Mili at NJIT, USA, for his valuable guidance and Dr. Aisha Hassan at International Islamic University Malaysia IIUM, Malaysia for her help and support; we are a very grateful staff of IIUM for their help.

References

[1] M. Broy, E. Dneret, "Software poineers," Springer-Verlag, pp. 442-452, 2002 Berlin Heidelberg.
[2] Jones, Cliff B, Systematic software development using VDM, 2nd ed.: Citeseer, 1990.
[3] Bowen, Jonathan P and Dunne, Steve and Galloway, Andy and King, Steve, ZB 2000: Formal Specification and Development in Z and B: First International Conference of B and Z Users York, UK, August 29-September 2, 2000 Proceedings. UK, UK: Springer, 2003.
[4] Woodcock, Jim and Cavalcanti, Ana, "A Concurrent Language for Refinement.," in IWFM., 2001, p. 5th.
[5] Jackson, Daniel, Software Abstractions: logic, language, and analysis.: MIT press, 2012.
[6] Bajwa, Imran Sarwar and Bordbar, Behzad and Lee, Mark G, "OCL constraints generation from natural language specification," in Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International, 2010, pp. 204-213.
[7] Abrial, Jean-Raymond and Abrial, Jean-Raymond, The B-book: assigning programs to meanings.: Cambridge University Press, 2005.
[8] Ali, Nahid A and Mirghani, Amal A and Ibrahim, Abdelrasoul Y, "Alneelain: A formal specification language," in Communication, Control, Computing and Electronics Engineering (ICCCCEE), 2017 International Conference on, 2017, pp. 1-9.
[9] Abdelrasoul Yahya Ibrahim, "Specifying abstract data types a behavioral model, an axiomatic representation," in Computing, Electrical and Electronics Engineering (ICCEEE), 2013 International Conference on, Khartoum, SUDAN, 2013, pp. 225-228.
[10] Mili, Ali and Tchier, Fairouz, Software testing: Concepts and operations.: John Wiley \& Sons, 2015.
[11] Ulrik Buchholz, Nathan Carter, Amine Chaieb, Floris van Doorn, Anthony Hart, Sean Leather, Christopher John Mazey, Daniel Velleman, and Théo Zimmerman. (2012) Theorem Proving in Lean. [Online]. https://leanprover.github.io/tutorial/

[12] Tamura, Naoyuki, "User's guide of a linear logic theorem prover (llprover)," Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University, Japan, pp. 1-12, May 1998.
[13] Hunt, Warren A and Krug, Robert Bellarmine and Moore, J, "Linear and nonlinear arithmetic in ACL2," in Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer, 2003, pp. 319-333.
[14] Paul, Hong Nai, Huan Zhang. (2007) Swansea Univeristy and McMaster Unverity. [Online]. https://www.meta-environment.org,http://www.cas.mcmaster.ca