Algebraic Specifications:Organised and focussed approach in software development

Rakesh.L
Research Scholar
Magadh University, Bodhgaya
Gaya, Bihar, India-824234
rakeshsct@yahoo.co.in

Dr. Manoranjan Kumar singh
PG Department of Mathematics
Magadh University, Bodhgaya
Gaya, Bihar, India-824234
drmksingh_gaya@yahoo.com

***Abstract* — Algebraic specification is a formal specification approach to deal with data structures in an implementation independent way. Algebraic specification is a technique whereby an object is specified in terms of the relationships between the operations that act on that object. In this paper we are interested in proving facts about specifications, in general, equations of the form t1 = t2 , where t1 and t2 are members of term (∑), being the signature of the specification. One way of executing the specification would be to compute the first algebra for the specification and then to check whether t1 and t2 belong in the same equivalence class. The use of formal specification techniques for software engineering has the advantage that we can reason about the correctness of the software before its construction. Algebraic specification methods can be used for software development to support verifiability, reliability, and usability. The main aim of this research work is to put such intuitive ideas into concrete setting in order for better quality product.**

Keywords- *Abstract data types (ADTs), Formal-Methods, Abstraction, Equational reasoning, Symbolic computation.*

I. Introduction

A specification can be considered as a kind of contract between the designers of the software and its customers. It describes the obligations and rights of both parties. A specification binds customers and designers by expressing the conditions under which services of a product are legitimate and by defining the results when calling these services. Specifications serve as a mechanism for generating questions. The construction of specifications forces the designers to think about the requirements definition and the intrinsic properties and functionalities of the software system to be designed. In this way the development of specifications helps the designers to better understand these requirements and to detect design inconsistencies, incompleteness and ambiguities in an early stage of software development. Specifications are obviously used for software documentation they describe the abstractions being made. Specifications are a powerful tool in the development of a program module during its software lifecycle. The presence of a good specification helps not only designers but also developers and maintainers. The modularity of the specification serves as a blueprint for the implementation phase, where a program is written in some executable language. In most software projects, the language used is an imperative nature. Unlike specifications, programs deal with implementational details as memory representation, memory management and coding of the system services. Writing a specification must not be seen as a separate phase in the development of software. Also, specification must be adapted each time modifications are introduced in any other phases of the software life cycle. Especially, specifications have to be adapted each time modifications are introduced in any of the other phases of the software life cycle. Especially, specifications have to be updated during the maintenance phase taking into account the evolution of the software system. With regard to the program validation, specifications may be very helpful to collect test cases to form a validation suite for the software system.

Specification must be at the same time compact, complete, consistent, precise and unambiguous. It has turned out that a natural language is not a good candidate as a specification language. In industry a lot of effort has been devoted to writing informal specifications for software systems, but little or no attention is paid to these specifications when they are badly needed during maintenance phase of the software life cycle [1].Specification in natural language rapidly become bulky, even to such extent that nobody has the courage to dig into them. Moreover, such specifications are at many places inaccurate, incomplete and ambiguous. It is very discouraging to discover after a long search that the answer can only be obtained by running the system with the appropriate input data. The tragedy in software development is that once a program modification is made without adapting the corresponding specification, the whole specification effort is lost. Having a non-existent or an obsolete specification is the reason why there exist so many software systems the behavior of which nobody can exactly derive in a reasonable lapse of time. Notice that running the program with the appropriate input can only give partial answers to questions about the system behavior. The entire idea is not to prove informal specification is useless. They are very useful as first hand information about the software product and as a comment to enhance the readability of the formal specifications.

II. Related work

Formal specifications, unlike the informal ones, enable designer to use rigorous mathematical reasoning. Properties of the specification can be proved to be true just as theorems can be proved in mathematics. In this way design errors, inconsistencies and incompleteness can be detected in an early stage of the software development [2]. Algebraic specification enables the designer to prove certain properties of the design and to prove that implementation meets its specification. Hence algebraic specification is used in a process called rapid prototyping. In a design strategy algebraic specification can be used as top down approach. The notion of top-down means here that specification is treated before any instruction of the implementation is written. The benefit of making constructive formal specification will certainly interest the practitioner, by rapid prototyping designers and customers will get user feedback and hands on experience with the software system before the implementation already gets started. In this way design errors due to misunderstandings between developers and customers, and lack of understanding of the services provided the product can be detected and corrected at an early stage. With the concept of constructive formal specifications and direct implementation, the boundaries between specifications and implementation are not very sharp. Both specifications and implementation are in fact programs, but the former are of a more abstract level than the latter. More over in the life cycle of a software product there may be more than two levels of abstraction [3]. A module may serve as a specification for the lower level and at the same time as an implementation for the higher one.

The following literature reveals the historical review on Algebraic specifications development and its significance.

In the Axiomatic method the behavior of the program is characterized by pre and post conditions. Its pioneers are Floyd, Hoare and Dijkstra.

Another well-known formalism is denotational semantics, especially the use of high order functions is very useful to describe the powerful control structures of programming languages its pioneers are Stoy and Gordon.

The new formalism based on the concept abstract data types has been developed as many sorted algebras and underlying mathematical model, such specifications are called algebraic specifications.

The pioneers of algebraic specifications are Zilles, Guttag, and the ADJ group consisting of Gougen, Thatcher, Wagner and Wright. They all consider a software module representing an ADT as many sorted algebra. The basic argument for the algebraic approach is that software module has exactly the same structure as algebra. The various sorts of data involved form sets and the operations of interest are functions among these sets.

The idea of behavioral equivalence was introduced by Giarratana.Gougen of ADJ research group presented the theory of many sorted algebra. Final algebra semantics were discovered by Wand.

The central idea of Sannella and Tarlecki is based on the fact that much work on algebraic specifications can be done independently of the particular logical system on which the specification formalism is based. The Munich CIP- group represented by Partsch took the class of all algebra fitting to a given specification as its semantics under one category.

The first specification language based on algebraic specifications was CLEAR invented by Burstall, where it was used to serve the needs of few product features.

Later on the concept of parameterized specifications in algebraic specification languages was encouraged the most popular one are ACT ONE founded by Ehrig and OBJ family by Goguen and Futatsugi, both based on many sorted algebra.

Algebraic specification languages may be considered as strongly typed functional languages like HOPE or as rewrite rule by Burstall and Huet. A combination of initial algebra semantics with Horn clause logic resulted in EQLOG and LPG.

The literature related to algebraic specifications discussed here includes topics like correctness, theorem proving, parameter zing, error handling and abstract implementations. Algebraic specification techniques and languages have been successfully applied to the specification of systems ranging from basic data types as stacks and natural numbers to highly sophisticated software systems as graphical programming language and the Unix file system. Algebraic specification techniques are used in wide spectrum of applications which allows the derivation of correct software from formal requirements through design specifications down to machine oriented level using jumps and pointers.

At the moment, many researchers all over the world are involved in research in the field of Algebraic specifications. Algebraic methods in software engineering are one of the fertile areas of research under one popular name Formal methods. Conceptually, algebraic specification provides a framework to formally describe software design. This framework allows for a better understanding of the software development process providing methodological insight concerning different issues. Algebraic specification is a formal specification approach that deals with data structures in an implementation-independent manner.

III. Intuitive Approach using ADTs

The aim of the software engineering is to develop software of high quality. By software we mean large programs. Quality sometimes called software engineering criteria are divided into two categories external and internal qualities. The external qualities we are particularly interested in are correctness, robustness, extendibility, reusability and efficiency. The internal qualities are modularity and continuity.

- *Correctness and reliability*: is the ability of the software system to perform its services as defined by its requirements definition and specification.
- *Robustness*: is the ability of a software system to continue to behave reasonably even in abnormal situations.
- *Efficiency*: is the ability of software to make good use of hardware resources and operating system services.
- *Modularity*: is the property of software to be divided into more or less autonomous components connected with a coherent and simple interface. Modularity is not only important at implementation level but also at specification level.
- *Continuity*: is a quality criterion that yields software systems that won't need drastic modifications because of small changes in the requirements definition.

Abstract data type is a class of data structures described by an external view, i.e. available services and properties of these services [4].

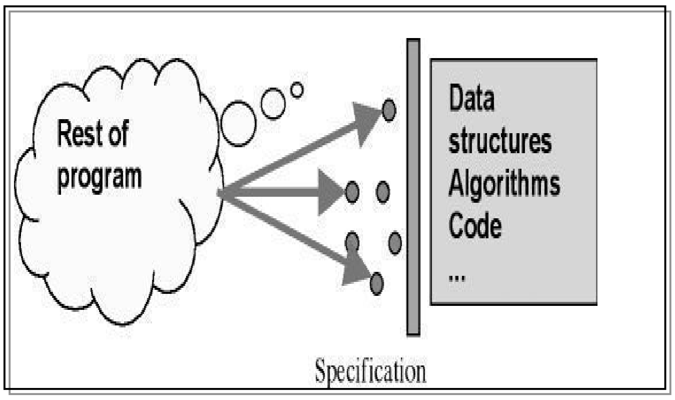

Figure. 1

The notion of an abstract data type is quite simple. It is a set of objects and the operations on those objects. The specification of those operations defines an interface between the abstract data type and the rest of the program. The interface defines the behavior of the operations – what they do, but not how they do it. The specification thus defines an abstraction barrier that isolates the rest of the program from the data structures, algorithms, and code involved in providing a realization of the type abstraction[5].Most of the software engineering methodology has one aspect in common, software is structured around data rather than around functions. The reason for this choice is that functions are not the most stable part of a system.Structuring around data yields systems with a higher degree of continuity and reusability. The key point in structured design of software systems is to look for abstract data types, abbreviated as ADTs. Roughly speaking, a specification of an ADT describes a class of data structures by listing the services available on the data structures, together with the intrinsic properties of these services. By specifying ADT, we do not care how a data structure is actually represented or how each operation is implemented [7]. What matters is what the data structure signifies at the level of a customer who wants to make instantiations of the data type for further use in his program.

To illustrate the concept of ADT, let us take the class of stacks of natural numbers, called stack. The specification of the stack will list the services newstack, push, isnewstack, pop and top. Furthermore, given an object of type stack, it describes how these services must be called for that object and it describes the intrinsic properties of these services. An example of such a property of stack is,

pop (push(s, n)) = = s;   (1)

where s is any stack object and n is any natural number. This property simply expresses that pushing a natural number on a stack s. The identifiers s and n are variables ranging over instantiations that is objects of types stack and Nat respectively.

Writing specification of ADTs is an activity that is located in the design phase of the software life cycle [8]. Specifications are designed in a modular way. Roughly speaking, with each specification module in the design phase corresponds a program module in the implementation phase. Specification modules, unlike program modules, make abstraction of all irrelevant details of data representation and procedure implementation. An important remark is that finding the appropriate set of specification modules is not an easy job. The choice of the modules must be such that complexity of the module interfaces is minimal and that continuity of the software system is maximal. Mostly a trade-off between these criteria has to be strived for. The main reason why we are so interested in modeling ADTs by mathematical objects is that we can profit from rigorous reasoning as defined for these objects. Rigorous reasoning on algebraic specification is based on two important techniques called equational reasoning and induction. Both techniques enable the designer to derive theorems from algebraic specification. These theorems then represent properties of the algebraic specification and of the software system described by it. The fact that such a theorem has been derived implies that the property it represents has been proved to be true. Due to the mathematical foundation of the chosen model, namely many sorted algebras, designers are able to give well defined and implementation independent meanings to ADTs [6]. A many sorted algebra is an abstract structure consisting of a family of sets of objects and a number of functions whose arguments and results belong to these sets. Due to this mathematical frame work, algebraic specifications can be made accurate and unambiguous. Initial algebras are often characterized by their properties of having no junk and having no confusion. Having no junk means that each object of the algebra can be denoted by at least one variable free term. Having no confusion means that two variable free terms denote the same object if they can be proved to be equal by equational reasoning from the given axioms [9]. The general and typical algebra is always initial. In literature, axioms are also called equations, laws or identities, and the terms are sometimes called expressions or formulas.

IV. RESULTS AND DISCUSSIONS

An algebraic specification is a mathematical description of an Abstract data type. Reasoning about the correctness of programs is made possible only by having a way to express their intended behavior. This is the object of algebraic specification -- programs are regarded as algebras consisting of data types and operations, and the intended behavior of a program is specified by means of formulas (say, equations) concerning these operations.

A. *Algebraic specification in Rapid prototyping*

Let us consider a abstract data type stack formally described by algebraic specification:

```
Sort stack;
operations
        newstack:→ stack;
    push:stack ▯  Nat→stack;
    isnewstack:stack→Bool;
        pop:stack→stack;
    top:stack→Nat;
declare s:stack;   n:Nat;
 axioms
   isnewstack(newstack)= =true;
   isnewstack(push(s,n) = = False;
   pop(newstack)= = newstack;
   pop(push(s,n) = = s;
   top(newstack) = = zero;
   top(push(s,n) = = n;
```

Figure.2

The sort(s) part lists the names of the abstract data types being described. In this example there is only one type, namely stack. The operations part lists the services available on instances of the type stack and syntactically describes how they have to be called. These parts are called the signature of the algebraic specification. For instance,

Push: stack * Nat → stack; (2)

means that push is a function with two arguments, with respective types Stack and Nat, and yields a result of type stack. It is also called constant. The term function here is used in the mathematical sense, not in the context of programming. So functions in the algebraic specification have no side effects. The axioms part formally describes the semantic properties of the algebraic specification. The specification can be applied to any data structure with the services described by functions with the same signature. The algebraic specification of stack expresses only the essential properties of the stack services without over specifying. It makes abstraction from any stack representation and service implementation details. It is the over specification that makes verification and rigorous reasoning difficult. Algebraic specifications provide a computational model with ADTs. As an example of such computations, consider the following expressions,

declare s1, s2 : stack ; n:Nat;

s1: pop(push(push(newstack,5),7));

s2: push(push(push(newstack,0),top(s1)),4);

n:top(pop(pop(s2))); (3)

By applying the axioms, successive simplications may be performed. These algebraic simplifications can be carried out mechanically. After these simplifications are carried out, the above expression becomes:

s1:= push(newstack,5);

Top(s1):= 5;

s2:= push(push(push(newstack,0),5),4);

n:= 0; (4)

This kind of symbolic computation is heavily related to concepts such as constructivity, term rewriting and rapid prototyping.

B. *Maintaining the Integrity of the Specifications by Equational reasoning*

Equational reasoning is one of the techniques that enable the software developer to use so called rigorous mathematical reasoning. Properties of the specification of the software can be proved to be true, even before the implementation has been started. Such proofs of properties are very similar to proofs of theorems in mathematics. Proofs about specifications of programs serve two purposes. They constitute the program documentation by excellence and they enhance software correctness and reliability. Given a presentation, equational reasoning is the process of deriving new axioms by applying the following rules.

i) Reflexivity : If t is a term of the presentation,

declare <declaration part>

axiom

t= = t;

is derivable by reflexivity if the variables used in the term t are listed in the declaration part.

ii) Symmetry : if the axiom

```
declare <declaration part>
axiom
t1= = t2;
```

if given or derivable, then

```
declare <declaration part>
axiom
t2= = t1;
```

is derivable.

iii) Transitivity: if the axioms

```
declare <declaration part>
axiom
t1= = t2;
t2= = t3;
```

are given or derivable, then

```
declare <declaration part>
axiom
t1= = t3;
```

is derivable.

iv). Abstraction:

```
declare <declaration part>
axiom
t1= = t2;
```

is given or derivable, x is a variable of sort Sj and x is not declared in the declaration part, then

```
declare x : Sj; <declaration part>
axiom
t1= = t2;
```

is derivable.

v) Concretion: if the axiom

```
declare x : Sj; <declaration part>
axiom
 t1= = t2;
```

is given or derivable, the set of variable- free terms of sort Sj is not empty and x does not appear in t1 nor t2, then

```
declare <declaration part>
axiom
 t1= = t2;
```

is derivable.

Given a presentation, deriving new axioms by equational reasoning always yields axioms that are satisfied by all algebras of the variety over the presentation. A second important property is that every axiom satisfied by all algebras of the variety over the presentation can be deduced using these rules. This above is a generic discussion that can be applied to any data structure in a software specification to check for consistency and soundness depending on the functionality and applicability.

C. Proof by Induction for technical soundness

Like equational reasoning, induction is a mathematical technique that can be used to derive new axioms from a given presentation. Axioms derivable by equational reasoning are satisfied by every algebra of the variety over the presentation. Axioms derivable by induction will be satisfied by every term algebra of the variety over the given presentation. As equational reasoning, induction is a very important technique to prove theorems of abstract data types. The main idea behind Induction is that one assumes instances of property being proved during its own proof. One of the hardest problems in discovering an inductive proof is finding an appropriate induction scheme that is complete and sound. Let us consider a classical example:

```
Sort Z
Operations
        Zero:  Z→ Z;
        Succ : Z→ Z;
        Pre :   Z→ Z;
        Add : Z* Z→ Z;
declare  i, j : Z;
axioms
        pre(succ(i) = = i;                      - 1 -
        succ(pre(i) = = i;                      - 2-
        add(zero,i) = = i;                        -3-
        add(succ(i),j) = = succ(add(i,j));       -4-
        add(pre(i),j) = = pre(add(i,j);           -5-
```

Figure.3

The presentation in Fig.3 defines the abstract data type of the integers including the successor, predecessor and addition functions. An axiom derivable by induction is the commutativity of the addition:

```
declare i,j:Z;
axiom
```

add(j , i) = = add(i, j);

It is provable by induction over j as well as over i.

These are some of the algebraic techniques that are useful in software engineering for verification and validation of specification and enhance confidence early in the lifecycle.

V. Conclusion

In this paper a novel concept of abstract data types is proposed through sensible use of mathematics to assist in the process of software development. The algebraic specifications of abstract data types are defined separately on the syntactic level of specifications and on the semantic level of algebras. The main results of the paper are different kinds of correctness criteria which are applied to a number of illustrating examples. Algebraic specification are used here to model prototypes, techniques like Equational reasoning and proof by Induction serve as uniqueness and completeness criteria and provides technical soundness for the specification. Properties of the specification of the software can be proved to be true, even before the implementation of software. Such proofs of properties are very similar to proofs of theorems in mathematics.

References

[1] F. Brooks, The Mythical-Man Month, Anniversary Edition: *Essays on Software Engineering*, Addison-Wesley (2000).
[2] O-J., Dahl, K. Nygaard, and B. Myhrhaug, "*The SIMULA 67 Common Base Language,"* Norwegian Computing Centre, Forskningsveien 1B, Oslo (1999).
[3] R.W. Floyd, "Assigning Meaning to Programs," *Proceedings of Symposium in Applied Mathematics*, vol. IX, International symposium at New York (2002), pp-20-30.
[4] J.V. Guttag, The Specification and Application to Programming of Abstract Data Types, Ph.D. *Thesis*, Dept. of Computer Science, University of Toronto (1975).
[5] J.V. Guttag and J.J. Horning, "Formal Specification as a Design Tool," *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas (1998), pp-2-9.
[6] J.V. Guttag, "Notes on Type Abstraction, Version 2,'' *IEEE Transactions on Software Engineering,*pp-46-49, vol. SE-6, no. 1 (1980).
[7] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10 (1985).
[8] A.Igelais, "Proofs of Correctness of Data Representations*," Acta Informatica*, pp- 56,vol. 1, no. 4 (2006).
[9] C.James, "Monitors: An Operating System Structuring Concept," *Communications of Information system*,pp-*193,*vol. 5, no7, 2005.

Authors Profile

L.Rakesh received his M.Tech in Software Engineering from Sri Jayachamarajendra College of Engineering, Mysore, India in 2001 as an honour student. He is a member of International Association of Computer Science and Information Technology, Singapore. He is also a member of International Association of Engineers, Hong Kong. He is pursuing his Ph.D degree from Magadh University, India. Presently he is working as a Assistant professor and Head of the Department in Computer Science & Engineering, SCT Institute of Technology, Bangalore, India. He has presented and published research papers in various National, International conferences and Journals. His research interests are Formal Methods in Software Engineering, 3G-Wireless Communication, Mobile Computing, Fuzzy Logic and Artificial agents.

Dr.Manoranjan Kumar Singh received his Ph.D degree from Magadh University, India in 1986. This author is Young Scientist awardee from Indian Science Congress Association in 1989. A life member of Indian Mathematical society, Indian Science congress and Bihar Mathematical society. He is also the member of Society of Fuzzy Mathematics and Information Science, I.S.I. Kolkata. He was awarded best lecturer award in 2005. He is currently working as a Senior Reader in post graduation Department of Mathematics, Magadh University. He is having overall teaching experience of 26 years including professional colleges. His major research Interests are in Fuzzy logic, Expert systems, Artificial Intelligence and Computer-Engineering. He has completed successfully Research Project entitled, Some contribution to the theory of Fuzzy Algebraic Structure funded by University Grants Commission, Kolkata region, India. His seminal work contribution to Fuzzification of various Mathematical concepts was awarded prestigious Doctor in Science degree.