Modern Methods in Software Development Using Algebraic Specifications

Prof (Dr.) Vijay Kumar, Dr. Rakesh Kumar

Abstract

With the help of Algebraic specification an object is identified by means of relations between the procedures which act upon object. Here, facts about specifications equations of the form $t_1 = t_2$, where $t_1$ and $t_2$ are members of term (∑), being signature of the specification are utilised. For executing specification to compute, first algebra for the specification utilized and then verified $t_1$ and $t_2$ belong in the same equivalence class or not. Utilisation of this method, techniques for software engineering has merit that correctness of the software can be validated before its final end of preparation. Algebraic specification is recognized specification method that interact with data structures in an implementation independent way. Algebraic specification methods used in software development for up keeping reliability and usability. There a better quality outcome may achieve with this methods.

Keywords- Algebraic Specifications, data types, software development.

Introduction

A specification may be observed as a kind of contract between software designers and its customers. It explains obligations and rights of both parties. Specification binds customers and designers by expressing the conditions under which services of a product are genuine and 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 and and functionalities of the software system to be designed. Therefore 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 projects, the language used is an imperative nature. Unlike specifications, programs deal with implementation details as memory representation, memory management and coding of the system services. A 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 should compact, consistent, precise and unambiguous. It observed that a natural language is not a good candidate as a specification. In industry a lot of effort devoted on 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. 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.

Methodology

Assets of the specification proved to be true just as theorems can be proved in mathematics. Therefore, design errors, inconsistencies and incompleteness detected in an early stage of the software development. Algebraic specification enables the designer to prove certain properties of the designed to prove that implementation meets its specification. Algebraic specification used in a process called rapid prototyping. In a design strategy algebraic specification used as top down approach. The notion of top-down means here that specification treated before any instruction of the implementation written. The benefit of making constructive formal specification certainly interest the practitioner, by rapid prototyping designers and customers 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. A module may serve as a specification for the lower level and at the same time as an implementation for the higher one. The new formalism based on concept abstract data types developed as many sorted algebras and underlying mathematical model, such specifications are called algebraic specifications. Algebraic specification techniques 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 involved in research in the field of Algebraic specifications. Algebraic methods in software engineering are one of 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.

Objective of 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 be 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.

Most of the software engineering methodology has one aspect in common, software 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. 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 new stack, push, is new stack, 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; 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. Specifications are designed in modular way. 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 reasoning as defined for these objects. Rigorous reasoning on algebraic specification is based on two important techniques called equation reasoning and induction. Both techniques enable the designer to derive theorems from algebraic specification. These theorems then represent properties of algebraic specification and of the software system described by it. The fact that such a theorem has been derived implies. 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 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. 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 Equation reasoning from the given axioms. The general and typical algebra is always initial. In literature, axioms are called equations, laws or identities, and the terms are sometimes called expressions or formulas.

Results and Discussions

Push: stack * Nat → stack; 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 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;

\[
s1: \text{pop}\left(\text{push}\left(\text{push}\left(\text{newstack},5\right),7\right)\right); \\
s2: \text{push}\left(\text{push}\left(\text{push}\left(\text{newstack},0\right),\text{top}(s1)\right),4\right); n: \text{top}\left(\text{pop\left(\text{pop}(s2)\right)}\right)\\
\]

By applying the axioms, successive implications may be performed. These algebraic simplifications can be carried out mechanically. After these simplifications are carried out, the

Above expression becomes:

\[
s1: = \text{push}\left(\text{newstack},5\right); \\
\text{Top}\left(s1\right): = 5; \\
s2: = \text{push}\left(\text{push}\left(\text{push}\left(\text{newstack},0\right),5\right),4\right); n: = 0;\\
\]

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

B. Maintaining the Integrity of the Specifications by Equation 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 proved to be true, even before the implementation 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 \(<\text{declaration part}>\) axiom

\[ t = t; \]

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;

\[ \text{Figure 1} \]

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

ii) Symmetry: if the axiom declare \(<\text{declaration part}>\) axiom

\[ t1 = t2; \]

if given or derivable, then declare \(<\text{declaration part}>\) axiom

\[ t2 = t1; \]

is derivable.

iii) Transitivity: if the axioms declare \(<\text{declaration part}>\) axiom

\[ t1 = t2; \]
\[ t2 = t3; \]

are given or derivable, then declare \(<\text{declaration part}>\) axiom

\[ t1 = t3; \]

is derivable.

iv). Abstraction: declare \(<\text{declaration part}>\) 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;
succ(pre(i) = = i;
add(zero,i) = = i;
add(succ(i),j) = = succ(add(i,j));
add(pre(i),j) = = pre(add(i,j));

Figure-2

Above presentation defines the abstract data type of the integers including the successor, predecessor and addition functions. An axiom derivable by induction is the commutative 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.

Conclusion

In the paper a new perception of abstract data types is analyzed via practical use of mathematics to assist in process of software development. The algebraic specifications of abstract data types explained disjointedly on level of specifications and on the semantic level of algebras. The main outcome of paper is different kinds of correctness criteria which applied to a number of illustrating examples. Algebraic specification utilized 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.

References


Author’s details

1Director-Principal, RDJ Group of Institutions, Basoli, Punjab, India

2Assistant Professor, PIT, Bhikhiwind, Punjab, India

Copy for Cite this Article- Prof (Dr.) Vijay Kumar, Dr. Rakesh Kumar, “Modern methods in software development using algebraic specifications”, International Journal of Science, Engineering and Technology, Volume 4 Issue 5: 2016, pp. 705-710.