Software as math

(Or why you should pay attention in abstract math class)

Perry Alexander

Discrete mathematics and abstract algebra courses are frequently required for computer science and engineering undergraduates. One could say these subjects define the mathematical basis for computing. Unfortunately, interest in the topics discussed in these courses frequently ends with the final exam! This work presents a reason to continue looking and exploring.

Algebras and specification

One common means for describing programs is to use algebraic specifications. As one might infer, algebraic specifications center on using algebras to describe abstract data types. Formally, an *algebra* consists of a set (frequently called a sort) and a collection of operators defined over that set. An example is the set of natural numbers and the common addition, subtraction, multiplication and division operators. Here the sort is the set of natural numbers {0, 1, 2, . . .} and the operators are +, -, ×, and ÷. Each operator is defined over two elements of the sort. An algebra is *multi-sorted* if it includes multiple sorts.

The mathematical concept of a multi-sorted algebra corresponds directly with the Abstract Data Type (ADT) based programming style. Recall that when defining an abstract data type, one defines: 1) a collection of types, and 2) a set of functions and procedures that manipulates those types. The correspondence with algebras is obvious. Types from ADTs correspond to sorts while the functions and procedures correspond to operators.

Figure 1 shows a simple algebraic specification for a stack written in the Larch Shared Language (LSL). The structure of the specification parallels that of an ADT written in C or C++. The `introduces` section defines operator names and sorts used in the specification. The definition of each operator is called its *signature*. It consists of the operator's name, the sorts in its domain and the sort in its range separated by a function symbol. For example, the `push` operator is

```
stack(E) : trait
introduces
  push: E,S → S
  pop: S → S
  top: S → E
  isEmpty: S → Bool
  empty: → S
asserts
  S generated by empty, push
  ∀ e:S, s:S
    pop(push(e,s))==s
    top(push(e,s))==e
    isEmpty(empty)
    ¬isEmpty(push(e,s))
implies converts pop,top
  exempting pop(empty), top(empty)
```

Fig. 1 An algebraic specification for a stack written in Larch Shared Language

defined as accepting an element of type *E*, a stack of type *S* and producing another stack of type *S*.

Note the special operator `empty` has no domain. This is known as a *nullary* function and represents a constant. Because all operators in LSL are mathematical functions, `empty` must always "return" the same value.

The signatures of all operators together define the signature of the specification. They denote all sorts and operations visible outside the specification. As such, the `introduces` section corresponds to the public part or the interface of an abstract data type.

Just as the implementation of an ADT defines behaviors of functions and procedures, the `asserts` defines the behaviors of operators. Traditional programming languages use an operational approach to define behaviors. The operational approach defines how each function's execution and data structure is organized. The operational style has the advantage of being efficiently executable. However, it is difficult to manipulate mathematically.

Algebraic specifications use a declarative approach to define behaviors. The declarative approach defines what each operator does without saying how. The declarative style is easy to manipulate mathematically and is more abstract. However, it is difficult or nearly impossible to execute declarative representations.

The `asserts` section of Fig. 1 specifies operators by defining when terms are equal using axioms. For example, the axiom `pop(push(e,s))==s` states that after a pushed element on a stack is popped, the result is equal to the original stack.

Besides specifying behavior of operators, the `asserts` section defines constructors for the sort `S` in the statement `S generated by empty,push`. This specification states that all elements of type `S` can be generated by the operators `empty` and `push`. Furthermore, the `generated by` clause states that induction can be used to prove things about all stacks. Such information is extremely useful when attempting to prove characteristics of stacks.

Every behavior that a stack should exhibit must either be expressed as an axiom in the `asserts` section, or be provable as a theorem from the specification. The `implies` section defines theorems that must be provable from the axioms in the `asserts` section. As such, the `implies` section defines conditions that must be true for the specification to be correct or for theorems that might be useful later.

0278-6648/97/$10.00 © 1997 IEEE

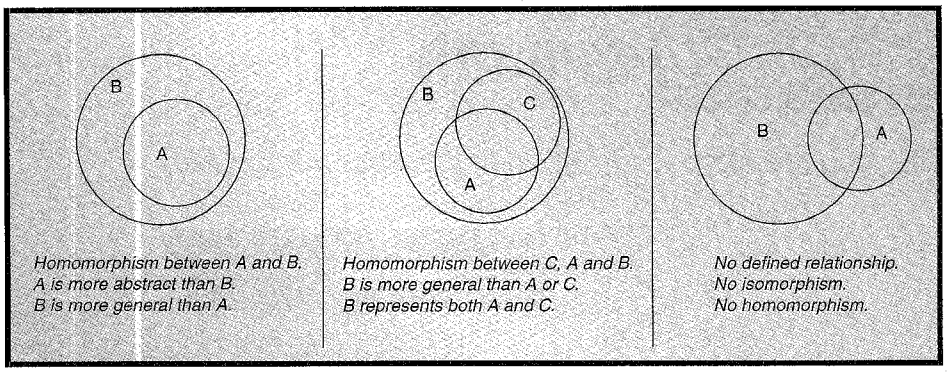

Fig. 2 Venn diagrams depicting various relationships between theories

Theories and models

Imagine a computer program existed that would take an algebraic specification and generate from it all its implications. The program would take a single theory and from it generate every theorem provable from its axioms. The resulting set of statements says everything that can be said about the specification. This set of true statements is the theory associated with the algebraic specification.

The mapping between an algebraic specification and its associated theory is a function. Only one theory can be associated with a single specification. However, many specifications may generate the same theory. Just as one can implement functions using many techniques, one can describe a single theory using many algebraic specifications. Given an algebraic specification and its associated theory, the specification is called a presentation of the theory. Any given theory may have many presentations.

A theory models or is a model of a system if: 1) every observable behavior of the system is a theorem in the theory, and 2) every theorem in the theory corresponds to a behavior in the system. The first condition is called completeness; the second is called consistency. For example, the Theory of Relativity is a model of the interaction between mass and energy. This is because every result obtained from it is a correct description, and because it says everything necessary about the interaction.

Specification morphisms

Why should software developers care at all about algebras and theories? Software development is all about changing models. One starts with a requirements model. One verifies that model and transforms, or morphs, that model into an architecture model. The architecture model is then morphed into a collection of module specifications. Continued morphism generates a model that is executable. Effectively, the morphing process generates a program that satisfies the initial model. Understanding specifications as theories and the morphing of theories enables profound understanding of software and system development.

A *specification morphism* represents the transformation of one specification into another. Specifically, a specification morphism: 1) renames existing operators and sorts; 2) adds new operators, sorts and axioms, and 3) deletes operators, sorts and axioms.

Assume that A and B are specifications and that $\rightarrow$ is a morphism. The notation $A \rightarrow B$ denotes that a morphism exists between specification A and specification B. By examining the theories T_A and T_B associated with A and B, special properties of the morphism and theories can be defined.

One desirable outcome is that every theorem T_A is also a theorem in T_B. This situation is called *logical containment*. If the theories are viewed as sets, then $T_A \subseteq T_B$. If such a characteristic holds, then $\rightarrow$ defines a homomorphism. If a second homomorphism, $\leftarrow$, exists such that $T_A \leftarrow T_B$, then $T_B \subseteq T_A$ because a subset is antisymmetric: $T_A = T_B$. In this situation, an isomorphism exists between A and B. Exploring homo and isomorphisms reveals much about the software development process.

Homomorphisms

When a homomorphism exists between specification A and specification B, everything true in T_A is true in T_B. Intuitively, the system B models exhibit every behavior the system A models exhibit. It is possible that some behaviors of B are not present in A because $T_A \subseteq T_B$ holds. If A and B are specifications of software systems, and B exhibits every behavior from A at a lower abstraction level, then B could be a correct refinement of A.

Verification is a process of determining that a morphism is in fact a homomorphism. When a representation is transformed, the desired behaviors of the original should be present in the new representation. Thus, if it can be shown that the subset relationship holds, then all the behaviors in the first representation are present in the second.

This activity precisely describes testing. Tests are developed that represent an important subset of behaviors. Those tests are then evaluated to determine if the new representation does in fact exhibit those behaviors. Effectively, testing determines if the behaviors are theorems in the new theory.

Synthesis finds a function that implements a homomorphism and uses it to generate a new theory. Instead of starting with two representations, synthesis starts with an abstract representation. It then attempts to transform it into a new representation. If a transformation can be found that guarantees a homomorphism, then that transformation can be used to automatically generate a new, correct representation.

Compilers are an excellent example of a synthesis algorithm. Here the initial representation is a computer program. The result is an executable model guaranteed to perform the task described by the high level program.

Informally, one description is more abstract than another if it contains less detail. When a homomorphism exists between A and B and none exists between B and A, then A is more abstract than B.

Consider a stack structure and a sequence type. It is possible to define a homomorphism between stacks and sequences: Push becomes the catenation to the beginning, top becomes the head and pop becomes the tail. Intuitively, stacks are more abstract than arrays because they specify less detail.

For example, sequences allow random catenation and random access where stacks do not. The behaviors for a stack are a subset of the behaviors of a sequence. Thus, $T_{stack} \subseteq T_{seq}$ and a homomorphism exists.

One description is more general than another if it can be used for more tasks. When a homomorphism exists between A and B and none exists between B and A, B may be more general than A.

Consider a queue structure and a

sequence type. It is possible to define a homomorphism between queues and sequences: Push becomes the catenation to the end, first becomes the head and rest becomes the tail. This implies that both queues and stacks have homomorphisms defined with sequences. Also, that the behaviors for both queues and stacks are a subset of the behaviors of sequences. No homomorphism exists between queues and stacks. This results in $T_{stack} \subseteq T_{seq}$, $T_{queue} \subseteq T_{seq}$.

The homomorphism concept gives us several useful results. First, it defines verification and synthesis. Finding a homomorphism assures us that one theory exhibits all the behaviors of another. Generating a homomorphism allows correct transformation of one theory into another. Second, it defines traditionally fuzzy concepts such as abstraction and generality. One theory is more abstract than another if a homomorphism exists between them. The complement of an abstraction is generally where a theory can be used in more circumstances.

Isomorphisms

When an isomorphism exists between *A* and *B*, the two specifications represent the same theory. In certain applications, it is necessary to demonstrate that two systems implement *exactly* the same behaviors. In such situations, an isomorphism is the goal. Intuitively, two theories that are isomorphic are also equivalently abstract and general. Isomorphisms tend to be used less often in software development because they are too strong.

Final thoughts

As software engineering evolves into a true engineering discipline, mathematics will play an increasingly large role. Mathematics hopefully will approach the role played in other engineering disciplines. It is difficult to say for certain if algebras, theories and morphisms all will play roles in that mathematics. However, these concepts do represent important tools in today's advanced software development activities. And it is a safe bet their roles will continue to blossom further.

Read more about it

• J. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, 23(9):8-24, Sept. 1990.

• John V. Guttag and James J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, New York, NY, 1993.

• I. Van Horebeek and J. Lewi, *Algebraic Specifications in Software Engineering: An Introduction*, Springer-Verlag, Berlin, 1989.

• Douglas R. Smith, "Constructing specification morphism," *Journal of Symbolic Computation*, 15:571-606, 1993.

• Douglas R. Smith, "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, 16(9):1024-1043, 1990.

• M. Lowry, A. Philpot, T. Pressburger and I. Underwood, "A Formal Approach to Domain-Oriented Software Design Environments," In *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, pages 48-57, Monterey, CA, September 1994 IEEE Computer Society Press.

About the author

Dr. Perry Alexander received his PhD in ECE, and his MSEE, BSEE and BSCS from the University of Kansas, Lawrence. He is an assistant professor of ECE and CS and director of the Knowledge-Based Software Engineering Laboratory at the University of Cincinnati (OH).