AxSL: An Easy-to-Read, Easy-to-Write Specification Language for Object-Oriented Components

Alyce Faulstich Brady
Kalamazoo College, Kalamazoo, MI
abrady@kzoo.edu

Abstract

This paper describes AxSL, an Axiomatic Specification Language that supports object-oriented concepts such as procedural operations, object state, encapsulation, operation overloading, and inheritance. In particular, AxSL supports a number of features that make its specifications easy to read and write: it uses natural language keywords, class operations (methods) are specified in an object-oriented notation, and developers may choose between axioms and post-conditions based on readability. AxSL also uses the concept of sufficient-completeness to help guide the development and verification of specifications.

Introduction

Application developers use specifications written in formal specification languages to improve the quality of programs. In particular, formal specifications are often used when safety and security are at risk. They may also be used to logically verify aspects of a design or to specify interfaces for reusable components. Many projects do not use formal specifications, however. One reason is that developing and verifying formal specifications is more time-consuming than developing more informal design methods, and project managers may feel that the benefits do not outweigh the costs for their own particular project. Another reason is that many designers and implementers are not familiar with formal specification methods.

This paper describes AxSL, an Axiomatic Specification Language for object-oriented applications. AxSL integrates features of several existing formal specification languages, and is designed to be easy to read and write by programmers with little training in formal methods. In particular, AxSL is keyword-based, its class operations (methods) are specified in an object-oriented notation, and it allows developers to choose between the two most common specification methods — axioms and post-conditions — for improved readability in a given situation. AxSL also supports the notion of sufficient-completeness, first defined by Guttag and Horning [1], which can help guide in the development of specifications or can be used to verify the completeness of specifications once they are written.
Existing Specification Languages

Specification languages generally fall into two categories: algebraic axiom specifications and mathematical models. This section presents a simple example in one language of each type. A stereotypical Stack class is used to focus on the notation rather than the intended behavior of the example.

Algebraic axiom specification languages allow a user to define an abstract data type by giving the function signatures (function name, return type, and formal parameter types) for the type and a set of equations that define the behavior of the functions in terms of one another. If the set of functions includes all those necessary to generate all possible instances of the abstract data type (the generating set of functions), then the equations, along with the axioms and rules of inference of first-order predicate calculus with equality, can be used to do proofs by induction.

Larch [2] is a two-tiered specification system, consisting of the language-independent Larch Shared Language and a set of language-specific Larch Interface Languages. The Larch Shared Language (LSL) is a classic algebraic axiom specification language for specifying abstract data types, known as traits. Each Larch Interface Language supports more detailed specifications tied to a specific programming language. The LSL equations that define the behavior of trait functions (i.e., that define the trait semantics) do not have to be repeated in the interface specification, but they must hold in order for the specification to be considered correct. Although LSL is a functional language, operations in a Larch Interface Language type specification may be functions, procedures, or a mix of the two, depending on the types of operations supported by the associated programming language.

Figure 1 demonstrates the two levels of a Larch specification for a Stack type: an abstract, language-independent specification written in LSL, and a language-specific interface specification written, in this case, in Larch/C++ [3]. The LSL specification defines the behavior of its functional operations in terms of one other, using algebraic axioms in the asserts section. The Larch/C++ specification defines operation signatures and behavior together using a model-based style with specification variables, pre-conditions, and post-conditions (indicated by the spec, requires, and ensures clauses, respectively). Primed variables in the post-conditions refer to modified (final) values while carats refer to initial values.

In mathematical modeling languages, the user defines an abstract data type by specifying an abstract data representation for the type and then defining the behavior of the type's operations in terms of their effect on the abstract data representation. Mathematical modeling notations use post-conditions, as opposed to axiomatic equations, to define operation behavior. VDM (Vienna Development Method) [4] and Z [5] are well-known mathematical modeling languages (although Z can also be used to express algebraic axiom concepts).

In Z, a specification may be decomposed into smaller pieces, called schemas, each

\[^{1}\text{This paper uses operation to encompass pure functions, pure procedures, and procedures with return values.}\]
template <class T>
class Stack {

public:
    // 0 uses STACK (T for ELT, Stack for SD)
    // 0 spec int MAXSIZE;
    // 0 spec int elt;
    // 0 spec T dataRep[MAXSIZE];
    // 0 invariant mElts >= 0;

    Stack();
    // 0 behavior {
    //   // guarantee mElts = 0;
    //   // 0 }

    void push (const T & elt);
    // 0 behavior {
    //   // requires assigned(mElts, pre)
    //   // 0 /
    //   // modifies mElts, dataRep;
    //   // ensures mElts = mElts + 1
    //   // 0 /
    //   // forall i,
    //   // 0 (0 <= i && i < mElts) \implies
    //   // 0 dataRep[i] = dataRep[i - 1];
    // 0 }

    T pop () ;
    // 0 behavior {
    //   // requires assigned(mElts, pre)
    //   // 0 /
    //   // modifies mElts, dataRep;
    //   // ensures result = dataRep[mElts - 1]
    //   // 0 /
    //   // forall i,
    //   // 0 (0 < i && i < mElts) \implies
    //   // 0 dataRep[i] = dataRep[i];
    // 0 }

    T top () const;
    // 0 behavior {
    //   // requires assigned(mElts, pre)
    //   // 0 /
    //   // modifies nothing;
    //   // ensures result = dataRep[mElts - 1];
    // 0 }

    int size () const;
    // 0 behavior {
    //   // requires assigned(mElts, pre)
    //   // 0 /
    //   // modifies nothing;
    //   // ensures result = mElts;
    // 0 }

    bool isEmpty () const;
    // 0 behavior {
    //   // requires assigned(mElts, pre)
    //   // 0 /
    //   // modifies nothing;
    //   // ensures result = mElts = 0;
    // 0 }
};

Figure 1: Larch Shared Language and Larch/C++ Specification for Stack
of which consists of a declaration part, in which variables are declared, and an axiom part, in which a relationship between the values of the variables is defined. Figure 2 shows Z schemas for the Stack data type, similar to the Larch specification in Figure 1, but modeled on a mathematical sequence. In this example, \( \mathbb{N} \) represents the natural numbers, \( \mathbb{B} \) represents Booleans, square brackets enclose sequences, and the arc in the post-condition on Push is the sequence concatenation operation. The first two schemas in the specification establish the \( \text{Stack}[T] \) data type and specify its initial state, while the remaining schemas specify its operations. The \( \Delta \) and \( \Xi \) symbols indicate whether or not the operation modifies the state of the Stack instance. Operation arguments are represented by variables followed by a question mark; operation results are indicated by exclamation points. Operations may have multiple results as demonstrated in the Pop schema, which, like the Larch/C++ pop operation, removes an element from the underlying data representation and returns that element.

In the axiom part of a schema, primed variables represent final values while unprimed variables represent initial values. An operation axiom specifies both the pre-condition and post-condition for the operation; pre-condition sub-expressions are ones that have neither primed variables nor operation results. For example, the Pop axiom

\[
q \neq [] \land q' = \text{tl}(q) \land e = \text{hd}(q)
\]

contains three sub-expressions; the first represents the pre-condition.

**Comparing Techniques**

At first glance, the Larch specification appears easier to read than the Z specification, at least for a novice. This is because Larch is keyword-based, not because of any inherent difference between algebraic axioms and mathematical modeling. In fact, only the Larch Shared Language specification is built on algebraic axioms; the Larch/C++ class, like the Z example, is based on a mathematical model.

Both specification styles have advantages. The primary advantage of the mathematical modeling technique is that post-conditions are generally easier to understand than axioms, both because they appear immediately after the operations whose behavior they define, and because they define operations in terms of a concrete concept, the mathematical model, rather than in terms of other behavior. Another advantage is that post-conditions can often be easily translated into assertions in an ordinary programming language if the developer wants to use them to test the implementation. A third advantage is that although the mathematical model is usually understood to be abstract, the method can be used to specify the actual data representation if that is appropriate for the particular application.

One of the main advantages of an algebraic axiom notation is precisely that it does not require a mathematical model. Although there are many classes for which there are obvious, or at least comprehensible, mathematical models, there are many others for which there are not. What is the appropriate mathematical model for an automated teller machine, for example? On the other hand, algebraic axioms can be used to model
Figure 2: Z Specification for Stack
mathematical models when that is desirable. Thus algebraic axioms are a more general specification technique than mathematical modeling. A third important feature of an algebraic axiom language is the ability it provides to formally prove certain characteristics of the specification, such as whether it is consistent and sufficiently complete (a concept described in more detail below [1]).

Unfortunately, a major disadvantage in using algebraic axioms to specify object-oriented classes is the fact that algebraic axiom specifications are inherently functional whereas objects are not. Mathematical model specifications, on the other hand, can handle the side-effects inherent in performing operations on objects. Another disadvantage with Larch, in particular, is that it requires two levels of specification, and the second level is not really language-independent.

Introducing AxSL

AxSL, an Axiomatic Specification Language developed at Rensselaer Polytechnic Institute for specifying object-oriented library components, blends the algebraic axiom and mathematical modeling approaches, combining the advantages of each. AxSL is based on the Larch Shared Language, but supports the specification of data representations when appropriate and allows the specification developer to choose whether to represent the behavior of non-modifying functions in axioms, as in Larch, or in post-conditions. AxSL also supports object-oriented operations and inheritance. It does this without requiring a second level of specification, and without being tied to any one object-oriented language. One of AxSL's design goals was to be easy to read and write, particularly for people trained in programming but not in formal specification methods. AxSL is easier to read than VDM, Z, and the object-oriented Z derivatives.

AxSL is language-independent, but uses an object-oriented signature notation for specifying the calling sequence of class operations and their return types, in which the object is an implicit parameter to all operations and an implicit return value or result from all modifying operations. It supports three distinct categories of object-oriented inheritance: subtype inheritance, parameterized type inheritance, and selective inheritance [6]. AxSL also supports dynamic binding and operation overriding. A template for creating AxSL specifications can be found in the appendix.

Figure 3 shows the AxSL specification of the Stack abstract data type. Although the AxSL specification also defines operation signatures and behavior, there are significant differences between the Larch and AxSL specifications. The most obvious is that the Larch version consists of two levels, whereas AxSL requires only one. Another difference is that the AxSL specification declares the signatures and behavior to be part of an EXTERNAL INTERFACE. This means that the operations are public. AxSL also supports other types of interfaces to a class, such as a subclass-visible interface, a private interface, and "friend" interfaces.

A third difference, particularly between the LSL specification and the AxSL version, is the procedural signature notation of the latter. AxSL operations may be pure functions, with a single return value, procedures that modify one or more parameters (including the underlying object), or a combination the two, as in the pop
CLASS Stack[T TYPE]
EXTERNAL INTERFACE
HAS OPERATIONS:
CON: mk-Stack (T TYPE) \rightarrow Stack[T]
CONSTRUCTS: SELF
POST-CONDITION: size() = 0
MOD: push (elt: T) \rightarrow
MODIFIES-AT-MOST: SELF
LET: s' = SELF
POST-CONDITION: size() = (s'.size() + 1) AND top() = elt AND SELF = auxStackAfterPush(s')
MOD: pop () \rightarrow elt: T
PRE-CONDITION: not empty
MODIFIES-AT-MOST: SELF
LET: s' = SELF
POST-CONDITION: elt = s'.top() AND SELF = auxStackAfterPop(s')
Obs: top () \rightarrow elt: T
PRE-CONDITION: not empty
Obs: size () \rightarrow Integer
Obs: empty () \rightarrow b: Boolean
POST-CONDITION: b = (size() = 0)
BASIC MODIFIERS: mk-Stack(TYPE), push(T)
BASIC OBSERVERS: top(), size()
HAS AUXILIARY FUNCTIONS:
auxStackAfterPush (Stack[T], T) \rightarrow Stack[T]
auxStackAfterPop (auxStack[T]) \rightarrow Stack[T]
PRE-CONDITION: not s.empty
BASIC MODIFIERS: mk-Stack(TYPE), auxStackAfterPush (Stack[T], T)
BASIC OBSERVERS: top(), size()
AXIOMS:
OBJECT INVARIANT: size() \geq 0
FORALL T TYPE, s:Stack[T], elt:T
auxStackAfterPop (auxStackAfterPush (s, elt)) = s
END Stack[T TYPE]

Figure 3: AxSL Stack Specification

operation, which pops the top element from the stack and returns that element to the calling operation. Since AxSL operations are not functions, the specification in Figure 3 makes a distinction between constructor operations, those that create new instances of a class, and modifier operations, those that modify the state of an existing object. The list of BASIC MODIFIERS (a subset of operations which can be used to create every possible instance, analogous to the generated by list in Larch) may contain both constructors and modifiers.

Another difference between the two specifications is that some of the axioms in Figure 1, those that define the behavior of observer operations, are replaced with post-conditions in Figure 3. Let expressions introduce names that can be used in post-conditions to refer to the initial values of modifiable objects. These include the implicit first parameter, called SELF and possibly other parameters as well. The MODIFIES-AT-MOST clause specifies which of these may be altered by the operation. (Note that primed variables are used to denote original values of variables in this
example, whereas they were used to denote final values in the Larch/C++ and Z examples.)

The final significant difference between the Larch and AxSL specifications is the introduction in the latter of auxiliary functions. Axiomatic specifications are expressed as functional equations that define the behavior of operations in terms of one another. For example, the LSL specification for STACK includes

$$\text{POP}([\text{PUSH}(s, e)]) = s$$

which defines the behavior of POP in terms of its effect on a stack created using PUSH. Side effects and object identity, however, are inherent to the object-oriented paradigm. Thus, object-oriented operations cannot be used in functional-style algebraic axioms. Auxiliary functions bridge the gap, providing a way for developers to define procedural operations in terms of functional axioms. For example, in the AxSL Stack specification, the axiom

$$\text{auxStackAfterPop}([\text{auxStackAfterPush}(s, e)]) = s$$

defines the same behavior as the LSL axiom, using the auxiliary functions associated with push and pop rather than with the operations themselves.

Creating AxSL Specifications

This paper focuses on low-level class specifications, which are an intermediate step between the traditional object-oriented analysis and design phases and the implementation phase. Therefore, we will assume that the first step in developing a specification is to develop an object-oriented design of the application using a traditional object-oriented design method. The centerpiece of object-oriented design methods is the identification of objects, the abstraction of their classes, and the identification of their operations or responsibilities. Thus, the design should include a list of classes, the operations for each class that have been identified so far, and specification of relationships among classes or instances, including inheritance relationships and "using" relationships (which objects know about, and can send messages to, or call operations on, which other objects). For each class, the specification process then consists of:

1. Identifying the public operations, specifying pre-conditions and modifies-at-most clauses.

2. Completing the specification of public class behavior by filling in missing pre-conditions, post-conditions, axioms, and invariants.

3. Identifying operations (and sometimes state information) and defining the behavior of subclass-visible and internal interfaces.
This is not a strict ordering, however. For example, some operations may not be identified until late in the process because their usefulness may not be obvious until then. Similarly, some post-conditions, axioms, and invariant clauses may be obvious right from the start. Furthermore, although most classes will probably be identified during the object-oriented design phase, some classes may not be identified until well into the specification phase. As Booch observes, design is an incremental and iterative process [7, p. 21].

**Identifying Classes and Inheritance Relationships**

The classes to be specified will usually come straight from the object-oriented design. Classes may be abstract or concrete; objects can be created from concrete classes, but not from abstract ones. Abstract classes provide a way to specify some conceptual behavior that will be used by other classes. In AxSL, abstract classes are indicated with the `ABSTRACT` keyword phrase.

If a class is a subclass of one or more other classes then that information should be found in the design as well, although the design may not distinguish among various types of inheritance. Therefore, for each superclass the specifier must determine what kind of inheritance relationship the subclass shares with that superclass. AxSL provides keywords for the three inheritance types described above, and also provides the generic `IS A` keyword for inheritance relationships whose type is as yet unknown. The inheritance specifications should also show what operations will be redefined, rather than inherited, in the subclass, and what operations will be excluded altogether if the class is a selective subclass. Inheritance is discussed in more detail below.

**Identifying Operations**

All object-oriented design methods provide a way to specify at least the public operations that a class supports. A complete AxSL specification of an operation includes the name, return type, and the names and types of parameters. The order of the parameters is also assumed to be important. As each operation is added to the class specification it must be identified as a constructor, a modifier, or an observer. In most cases, this identification is straightforward: constructors create new objects, modifiers change the state of existing objects, and observers return information about existing objects. Constructors do not take an object as an implicit first parameter, while modifiers and observers do.

Some operations, however, are harder to classify. For example, the `pop` operation in Figure 3 modifies the Stack object and returns the element that had been on the top of the Stack. Should `pop` be considered a modifier, an observer, or both? There are several other cases in which what appears to be an observer actually qualifies as a modifier. An object may modify another object that is part of its own state, a parameter object, or a global object. An operation may also modify the state of the system, by generating output for example, or produce a non-deterministic return value, such as a result from a human user. Any operation that can be considered both a modifier and an observer should be classified as a modifier. Only “pure” or “passive”
observers should be identified as observers in a class definition.

It is also possible, though less common, for an operation to be both a constructor and a modifier. An object could have an operation that creates a new backup of itself before making modifications, for example. In this case, the operation would take the object as an implicit first parameter, and so would qualify as a modifier. Any operation that creates new objects, whether constructor or modifier, should list the objects it creates after the CONSTRUCTS keyword. The most common object for an operation to create is SELF, but parameter objects and return values may also be created by the operation.

Beginning to Specify Operation Behavior

An operation’s PRE-CONDITION is a predicate that asserts what must be true when the operation is invoked for it to behave as expected. The POST-CONDITION is a predicate that asserts what is true when the operation has completed, primarily with respect to its return value and side effects. Since the pre-condition describes what the operation can expect from the caller and the post-condition describes what the caller can expect from the operation, these two together are sometimes referred to as a contract [2] [8] [9]. The CONSTRUCTS list, described in the previous section, indicates what new objects an operation may create. The MODIFIES-AT-MOST list indicates which objects the operation may modify. The most common object for an operation to modify is SELF, but it may also modify its parameter objects. If state information is specified for the object (see below), the operation may modify that as well. The modifies-at-most list is actually a special kind of post-condition, because it specifies, by omission, which objects are guaranteed to be unchanged by the operation.

When an operation is added to a class specification, the operation pre-condition, if there is one, and its modifies-at-most list, if it is a modifier, should be specified. If the object-oriented design shows which other classes or class operations are used by the current operation, that information may also be included after the USES keyword.

Some post-conditions are obvious at this stage, and may be specified as well. Otherwise, they will be added later, using the notion of sufficient-completeness as a guide. Once the axioms and post-conditions needed for sufficient-completeness have been generated, extraneous post-conditions can be removed unless they add to the readability of the specification. Figure 4 shows an early stage of a Stack class definition under development. The pre-condition on the only operation that needs one has been specified, as have the modifies-at-most lists on the modifiers. Some post-condition expressions have been identified.

Specifying Behavior More Completely

A class’s behavior is specified by a combination of operation pre-conditions, operation post-conditions, invariants, and axioms. We have already seen that axioms use functional equations. It is also a good idea to restrict post-conditions to functional expressions, since one could imagine adding post-conditions to the eventual implementation in the form of assertions, which should not modify the state of the
CLASS Stack[T: type]

EXTERNAL INTERFACE
HAS OPERATIONS:
  CON: mk-Stack (T: type) → s: Stack[T]
    CONSTRUCTS: self
    POST-CONDITION: (s = SELF) AND empty()
  MOD: push (elt: T) →
    MODIFIES-AT-MOST: self
    POST-CONDITION: not empty()
  MOD: pop () → elt: T
    PRE-CONDITION: not empty()
    MODIFIES-AT-MOST: self
  OBS: size () → i: Integer
  OBS: empty () → b: Boolean

END Stack[T: type]

Figure 4: Stack Under Construction: Specifying Operations

object. (Eiffel [9] supports assertions directly, C [10] and C++ [11] provide limited support, and C++ and Java [12] support exceptions, which can be used similarly.) Object-oriented operations, however, are inherently procedural; they may change the state of the object to which they are tied, return a value other than the modified object, and even modify their arguments. This means that they may have as many results as parameters in the MODIFIES-AT-MOST clause, including the implicit first argument of the object itself, as well as one result for the return value.

Auxiliary Functions

An AxSL specification defines a set of auxiliary functions to define the behavior of an operation, one function for each operation result. Each auxiliary function has the same argument types as the operation (with the operation’s implicit first parameter, the object, made explicit), has no side effects, and returns the single result (i.e., return value or side effect) with which it is associated. The auxiliary functions for constructor operations are exceptions; they do not require an extra first parameter representing the object. If the operation has a pre-condition, the auxiliary function has the same pre-condition with any calls to operations replaced by calls to the appropriate auxiliary functions.

Post-condition clauses associate each auxiliary function with its corresponding operation result. For modifying operations, let expressions provide a way to name the initial value of each object that the operation might modify. For example, Figure 5 shows the Stack class with its full complement of auxiliary functions and let-expressions. There are two auxiliary functions representing the two results of the pop operation: auxPop represents the return value (the top element being removed) and auxStackAfterPop represents the side-effect (the modified stack object). The post-condition on the pop operation contains the two expressions that establish this correspondence, with s' referring to the initial value of the object and SELF referring to the final value.

Given an operation signature, pre-condition, and modifies-at-most list, the
Class Stack[T: type]

External Interface

Has Operations:

Con: mk-Stack (T: type) → s: Stack[T]

Constructs: self

Post-Condition: (s = aux MkStack(T)) AND empty()

Mod: push (elt: T) →

Modifies-At-Most: self

Let: s' = self

Post-Condition: Not empty() AND self = auxStackAfterPush (s', elt)

Mod: pop () → elt: T

Pre-Condition: Not empty()

Modifies-At-Most: self

Let: s' = self

Post-Condition: elt = auxPop(s') AND self = auxStackAfterPop(s')

Obs: size () → i: Integer

Post-Condition: i = auxSize(self)

Obs: empty () → b: Boolean

Post-Condition: e = auxEmpty(self)


Auxiliary Functions:

aux MkStack (T: type) → s: Stack[T]

auxStackAfterPush (s: Stack[T], elt: T) → Stack[T]

auxStackAfterPop (s: Stack[T]) → Stack[T]

Pre-Condition: Not auxEmpty(s)

auxPop (s: Stack[T]) → elt: T

Pre-Condition: Not auxEmpty(s)

auxSize (s: Stack[T]) → i: Integer

auxEmpty (s: Stack[T]) → e: Boolean

.

End Stack[T: type]

Figure 5: Stack Under Construction: Adding Auxiliary Functions

generation of the operation’s auxiliary functions, let-expressions, and associated post-condition clauses is so formulaic that it can be done by software tools. The Pax tool for parsing AxSL specifications generates the appropriate auxiliary functions, let-expressions, and associated post-condition clauses for operations that are missing them.

Developing Axioms

Once we have the set of auxiliary functions for an operation, we can define their behavior using traditional algebraic axiom semantics. Intuitively, we can think of the behavior of the original operation as being the sum of the behaviors of the functions. (A formal justification of this, and a demonstration of how auxiliary functions can be used to develop proof goals for operations, appears in [13].)

The first step is to identify the list of BASIC MODIFIERS. This is a subset of the class’s operations such that all possible objects of the class can be created by applying (perhaps repeatedly) operations in the subset. For example, for any Stack object that has been created by some combination of the mk-Stack, push, and pop operations, an equivalent object could be created using just mk-Stack and push. Modifiers that are
not basic modifiers are called extra modifiers. Actually, to develop axioms we need
basic modifying auxiliary functions, not basic modifying operations. For each basic
modifying operation, the auxiliary function that corresponds to the modification of SELF
becomes a basic modifier function. In addition to identifying basic and extra modifiers,
it is also convenient to divide the observers into basic observers (the smallest set of
observers that produce all observable information about the class) and extra observers.

The primary guiding principle in determining which axioms are needed in a class
specification is to achieve sufficient-completeness [1]. A class is sufficiently complete if:

- a minimal set of generating auxiliary functions, the basic modifying functions, has
  been identified, and
- the axioms and post-conditions for the class provide rewrite rules such that:
  1. All variable-free expressions of the class's type that contain auxiliary
     functions that are not basic modifying functions can be rewritten as
     expressions whose only functions of the class's type are basic modifying
     functions.
  2. All variable-free expressions not of the class's type can be rewritten as
     expressions that do not contain any functions of the class's type.

(A more formal version of these rules can be found in [13] and [14].)

In order to ensure the first of the sufficiency clauses, an extra modifier function,
when applied to any valid object of the class, must be capable of being rewritten in
terms of basic modifier functions. To ensure the second sufficiency clause, any auxiliary
function not of the class's type, i.e., the observer functions and auxiliary functions that
represent modifications to objects other than SELF, when applied to any valid object of
the class, must be capable of being rewritten in terms that contain no calls to the class's
basic or extra modifier functions. The algebraic axioms of the class specification provide
the necessary re-write rules. If the specification developer can show that, using the
algebraic axioms, the rules above are satisfied, then the set of axioms, and, therefore,
the specification of the behavior of the class, is considered sufficiently complete.

The notion of sufficient-completeness can be used not only to verify the
completeness of a set of axioms, but also to drive the generation of the left-hand sides of
axioms. To show that any given auxiliary function that is not a basic modifier satisfies
the first or second sufficiency rule “when applied to any valid object of the class,” that
auxiliary function should have axioms that relate it to each of the basic modifiers. In
other words, each extra modifying function should have as many axioms associated
with it as there are basic modifying functions; each axiom defines the behavior of the
extra modifying function when applied to a different basic modifying function. A basic
observer function should also have an axiom relating it to each of the basic modifier
functions. Extra observers are the exceptional case; they may be defined in terms of
one or more basic observers in a single equation. Figure 6 shows the suggested axiom

---

2As pure functions, the basic modifying auxiliary functions are actually constructors.
forall T: type, s: Stack[T], elt: T
auxStackAfterPop (auxMkStack (T)) =
auxStackAfterPop (auxStackAfterPush (s, elt)) =
auxPop (auxMkStack (T)) =
auxPop (auxStackAfterPush (s, elt)) =
auxSize (auxMkStack (T)) =
auxSize (auxStackAfterPush (s, elt)) =
auxEmpty (auxMkStack (T)) =
auxEmpty (auxStackAfterPush (s, elt)) =

Key: s is a universal quantifier for Stack objects
elt is a universal quantifier for objects of type T

Figure 6: Suggested Axiom Left-Hand Sides for Stack Example

left-hand sides for the Stack in Figure 5. The class specification developer must then
generate right-hand sides that accurately specify the behavior of the operations.

In the completed Stack example in Figure 7, however, the two axioms relating the
pop auxiliary functions to the auxMkStack function have not been provided, because
the pre-condition on pop precludes the possibility of the operation being called on a
newly created Stack object (the behavior of pop on an empty stack is undefined). There
is also only a single axiom for the auxEmpty function; it follows the exception noted
above that some observer functions, applied directly to a universal quantifier, may be
declared in terms of one or more other observers in a single equation. Finally, the extra
post-condition expressions introduced in Figure 4, which specified that a newly
constructed Stack is empty while a Stack after a push operation is not, have been
removed. They are not required to achieve sufficient-completeness, and, therefore, are
redundant. Such expressions can be kept to improve readability; but they make it
harder to maintain logical consistency. (In this example, the object invariant is also
redundant because it can be proven inductively; it was retained to improve readability
and to provide an example of an invariant.)

What makes the set of axioms in Figure 7 sufficiently complete, however, is not
just the choice of left-hand sides, but also the contents of the right-hand sides. Axioms
(and post-conditions) consist primarily of functional composition and the equality
operator. In this case, the left-hand side expressions whose types are not the class's
type, i.e., the last four axioms, reduce to expressions that do not contain any functions
of the class's type (rule 2). The one left-hand side expression of the class's type, the
first axiom, reduces to an expression whose only functions of the class's type are basic
modifiers (rule 1). The right-hand side of the first axiom can be shown to reduce to an
expression containing only basic modifier functions by an inductive proof. Since the
pre-condition on pop does not allow it to be called on empty stacks, the base case for
the inductive proof is

auxStackAfterPop (auxStackAfterPush (auxMkStack(T), elt)) = auxMkStack(T)

Algebraic axioms are frequently recursive. Imagine, for example, a Queue class
with a similar syntax to that for a Stack, except that Queue has enqueue and dequeue
CLASS Stack[T: TYPE]
  
EXTERNAL INTERFACE

HAS OPERATIONS:
  CON: mk-Stack (T: TYPE) → s: Stack[T]
    CONSTRUCTS: self
    POST-CONDITION: s = aux%Stack(T)

MOD: push (elt: T) →
    MODIFIES-AT-MOST: self
    LET: s' = self
    POST-CONDITION: self = auxStackAfterPush (s', elt)

MOD: pop () → elt: T
    PRE-CONDITION: not empty()
    MODIFIES-AT-MOST: self
    LET: s' = self
    POST-CONDITION: elt = auxPop(s') AND self = auxStackAfterPop(s')

OBS: size () → i: Integer
    POST-CONDITION: i = auxSize(self)

OBS: empty () → b: Boolean
    POST-CONDITION: e = auxEmpty(self)

BASIC MODIFIERS: mk-Stack(TYPE), push(T)

BASIC OBSERVERS: pop(), size()

AUXILIARY FUNCTIONS:
  aux%Stack (T: TYPE) → s: Stack[T]
  auxStackAfterPush (s: Stack[T], elt: T) → Stack[T]
  auxStackAfterPop (s: Stack[T]) → Stack[T]
    PRE-CONDITION: not auxEmpty(s)
  auxPop (s: Stack[T]) → elt: T
    PRE-CONDITION: not auxEmpty(s)
  auxSize (s: Stack[T]) → i: Integer
  auxEmpty (s: Stack[T]) → e: Boolean

BASIC MODIFIERS: aux%Stack(TYPE), auxStackAfterPush (Stack[T], T)

BASIC OBSERVERS: auxPop(Stack[T]), auxSize(Stack[T])

AXIOMS:

OBJECT INVARIANT: auxSize(self) ≥ 0

FOR ALL T: TYPE, s: Stack[T], elt: T
  auxStackAfterPop (auxStackAfterPush (s, elt)) = s
  auxPop (auxStackAfterPush (s, elt)) = elt
  auxSize (aux%Stack (T)) = 0
  auxSize (auxStackAfterPush (s, elt)) = auxSize(s) + 1
  auxEmpty(s) = (auxSize(s) = 0)

END Stack[T: TYPE]

Figure 7: Complete Stack Example
Class Queue[T: type]

::

AXIOMS:

OBJECT INVARIANT: auxSize(self) ≥ 0

FORALL T: type, q: Queue[T], elt: T

auxQAfterDequeue (auxQAfterEnqueue (auxQQueue(T), elt)) = auxQQueue(T)

auxQAfterDequeue (auxQAfterEnqueue (auxQAfterEnqueue (q, elt1), elt2))

= auxQAfterEnqueue (auxQAfterEnqueue (auxQAfterEnqueue (q, elt1)), elt2)

auxDequeue (auxQAfterEnqueue (auxQQueue(T), elt)) = elt

auxDequeue (auxQAfterEnqueue (auxQAfterEnqueue (q, elt1), elt2))

= auxQAfterEnqueue (auxDequeue (auxQAfterEnqueue (q, elt1)), elt2)

::

END Queue[T: type]

Figure 8: Axioms for a Queue Class

operations. Like pop, the dequeue operation has a pre-condition requiring that the Queue object not be empty. The axioms defining the behavior of dequeue are shown in Figure 8. The second and fourth axioms in this example are analogous to the first and second axioms in the Stack class. Since these axioms are recursive, though, the left-hand sides have an extra enqueue, i.e., they define the dequeue operation when there are at least two items on the queue. Otherwise, they might refer to the behavior of a dequeue operation when there are no items on the queue, a situation that is disallowed by the pre-condition on dequeue.

On the surface, the Queue axioms do not seem to provide re-write rules that guarantee sufficient-completeness. After all, the right-hand side of the second axiom contains a call to an extra modifier, and the right-hand side of the fourth axiom, although not of type Queue, contains functions of type Queue. As with the Stack class, the specification developer must use inductive proofs to show that these axioms satisfy the rules of sufficient-completeness. In this case, the base cases for the induction are provided explicitly as the first and third axioms.

Invariants

AxSL supports invariants, as in Figure 3 and Figure 7. An OBJECT INVARIANT describes any requirements or constraints that each object of the class must satisfy individually to be in a valid, static state between operation calls. A CLASS INVARIANT describes requirements or constraints that all objects of the class must satisfy collectively. For example, if an application has an upper bound on the number of instances of a class it can handle, then the class invariant would state that the number of instances is not greater than the upper bound. An invariant is an implicit expression in the pre- and post-conditions of every public operation of the class, and helps to define the behavior of those operations.
Simplifying Specifications

The completed Stack in Figure 7 is not exactly the same as the original AxSL Stack example in Figure 3. The original specification employed several techniques to make specifications easier to read and write. First of all, it did not define auxiliary functions for the constructor and observer operations. Operations that are themselves functions, such as mk-Stack, size, and empty, may replace their corresponding auxiliary functions in axioms and post-conditions, resulting in specifications that are shorter and easier to understand. In fact, the specification in Figure 3 used the functional `top` operation as the auxiliary function associated with `pop`'s return value. The disadvantage of doing this, though, is that complex expressions using the dot notation of class operations may be harder to read than nested function calls.

Another difference between the Stack specifications in Figures 3 and 7 is that the original example defined the behavior of the `size` and `top` observer operations in the post-conditions for `mk-Stack` and `push` and the behavior of the `empty` operation in its own post-condition, rather than in axioms. Figure 7 does not include a `top` operation, but the post-condition on `push` could have been

\[
\text{Post-Condition: } \text{size}() = (s'.\text{size}() + 1) \text{ and auxPop}(s') = \text{elt}
\]

\[
\text{and self = auxStackAfterPush}(s', \text{elt})
\]

Generally speaking, post-conditions are easier to understand than axioms because they describe what is true immediately after the call to the operation rather than what would, hypothetically, be true if one function were applied to the result of another. On the other hand, modifying operations should be defined in axioms rather than post-conditions if the specification expressions will correspond to assertions in the final implementation. Even specifying observer behavior in post-conditions leads to scattering semantic information throughout the class rather than concentrating it in a single place – the Axioms section. Thus, deciding which style to use for observer operations is a question of readability rather than of substance.

Finally, both specifications are missing the `pop` axiom for empty stacks. Since the corresponding pre-conditions are defined, however, the specification should be considered sufficiently-complete. Thus, testing for sufficient-completeness in an AxSL specification must take into account axioms, pre-conditions, post-conditions, and invariants.

Identifying Other Interfaces

In addition to external interfaces that specify public operations, AxSL also supports other types of class interfaces. A subclass-visible interface contains operations available to subclass objects, a private interface contains operations available only to the object itself, and friend interfaces contain operations visible to specified classes or class operations. Figure 9 shows a Stack class with the `size` operation available only to subclass objects and `self`, while the operations in the external interface are available to all objects.
Class Stack[T: type]
Has external interface:
Has operations:
Con: mk-Stack (T: type) → s: Stack[T]
  Constructors: self
  Post-condition: (s = self) AND empty()
Mod: push (elt: T) →
  Modifiers->At-Most: self
  Post-condition: NOT empty()
Mod: pop () → elt: T
  Pre-condition: NOT empty()
  Modifiers->At-Most: self
Obs: empty () → b: Boolean

Has subclass-visible interface:
Has operations:
Obs: size () → Integer

End Stack[T: type]

Figure 9: An Example of Multiple Interfaces

Friend interfaces are an unusual feature in AxSL, because they do not correspond exactly to any commonly implemented access level in object-oriented programming languages. A class may have several friend interfaces, each of which provides a specific set of operations to a specific set of classes or class operations. (This is different from C++ friends, which have access to all operations in the class.) AxSL supports friend interfaces because they are a useful concept and because some design methods use them. Many class specifications include only external, subclass-visible, and friend interfaces, leaving the internal interface for each implementation to determine separately.

Each interface should include the operation signatures and auxiliary functions specific to that interface. The lists of basic modifiers and observers and the axioms that define their behavior should define the entire set of operations visible at that interface; for example, the set of basic modifiers for the subclass-visible interface might include external operations. Operations that are basic modifiers or basic observers for a high-level interface, such as the external interface, may be extra modifiers or extra observers in the context of a lower-level interface.

Specifying State
AxSL specifications can be used to specify data representations, as in mathematical model specifications, when that is appropriate. State specifications are usually found in the internal interface. Each state variable should have two auxiliary functions corresponding to the implicit operations of accessing and modifying the state variable. The signatures of these auxiliary functions look like

accessStateVar (c: ClassType) → var: StateVarType
modifyStateVar (c: ClassType, newVal: StateVarType) → c: ClassType

For example, if we were to add a mathematical sequence as a state variable in the internal interface for the Stack class, the auxiliary functions would be
accessSeq (s: Stack) → val: Sequence[T]
modifySeq (s: Stack, newVal: Sequence[T]) → s: Stack

In the context of the internal interface, modifySeq would be a basic modifier rather than push, and the relationship between them could be expressed with the following axiom:

\[
\text{FORALL } T: \text{TYPE}, s: \text{Stack}, e: T
\text{auxStackAfterPush}(s, e) = \text{modifySeq}(s, \text{concat}(e, \text{accessSeq}(s)))
\]

where concat is the sequence concatenation operation and [e] is the sequence containing only e.

**Inheritance**

The Stack examples in this paper have not used inheritance. Nevertheless, AxiSL supports the specification of inheritance relationships. Unfortunately, there is not just one semantic definition of inheritance in the object-oriented world; instead, class hierarchies and inheritance are used in various languages to represent a number of different kinds of class relationships. All of these, however, fall into three distinct categories:

- **subtype inheritance** (or behavioral inheritance [15]): allows type-safe dynamic binding of subclasses to superclasses
- **parameterized type inheritance**: supports specialization of homogeneous collection classes such as Lists, Sets, Stacks, and Queues
- **selective inheritance**: allows a subclass to selectively inherit any subset of properties from the superclass [6]

For example, C++ subclasses of public base classes with virtual operations are subtypes. They selectively inherit from private base classes. Java interfaces are abstract classes whose subclasses are subtypes of the interface class.

AxiSL supports all three types of inheritance and also an informal, unspecified inheritance. AxiSL also supports multiple inheritance. A subclass may, for example, have a subtype inheritance relationship with one superclass and a selective inheritance relationship with another. An AxiSL specification developer can specify the inheritance type explicitly, as in Is A SUBTYPE OF, in which case it is possible to use inheritance rules about axioms, pre-conditions, post-conditions, and invariants to infer knowledge about some of the behavior of subclass operations from the superclass. Figure 10 provides an informal description of the inheritance rules for subtype inheritance as an example. On the other hand, a specification developer who has not carefully analyzed the inheritance type of a subclass may use the generic Is A specification. If the specification developer has included all of the relevant operation signatures, pre- and post-conditions, and axioms, it is possible to use the inheritance rules to logically deduce the inheritance relationship between subclass and superclass.

---

3These rules, described more fully in [6] and [14], are modifications of those by Liskov and Wing [16].
For any parent class $P$ and subclass $S$ of $P$,

1. Every public operation of $P$ is either inherited by $S$ or redefined in $S$ as a public operation.
2. The axioms for $P$ must include and be consistent with ($\triangleleft$ not introduce contradictions to) the axioms for $S$.
3. Any object of class $S$ must satisfy the object invariant of class $P$; therefore, the subclass object invariant may be the same or stronger, but not weaker.
4. For every public operation in $P$ that is redefined in $S$,
   - The redefined name must be the same as the original.
   - The number of formal parameters must be the same.
   - Any parameter that was valid for the original operation must be valid for the redefined operation (contravariance of argument types).
   - The return type must be the same as, or a subtype of, the return type of the original operation (covariance of return types).
   - Subclass pre-conditions may be the same or weaker, but not stronger (except as guaranteed by the subclass object invariant): $\text{pre}(op) \land \text{Imag}_S \Rightarrow \text{pre}(op_S)$.
   - When the superclass pre-condition holds, subclass post-conditions may be the same or stronger, but not weaker; $\triangleleft$, a subclass post-condition may be weaker only when the superclass pre-condition is not met: $\text{pre}(op) \Rightarrow (\text{post}(op_S) \Rightarrow \text{post}(op))$.

Figure 10: Rules for Subtype Inheritance

Inheritance plays a significant role in determining sufficient-completeness. In subtype inheritance relationships, inherited operations need not be repeated and post-conditions of redefined operations need not explicitly state the conditions of the superclass post-condition. Furthermore, the axioms of the superclass need not be repeated in the subclass. The pre-conditions of redefined operations, on the other hand, must always be fully specified, since redefined operations need not require all of the conditions of the superclass's pre-conditions.

Tool Support

A prototype parsing tool called Pax has been developed to parse and validate AxSL specifications. Pax reads classes written in the AxSL specification language, flagging syntactic and some semantic errors. It does type checking, checks for redundancies among operations or between operations and auxiliary functions, verifies the visibility of used and inherited operations, and verifies that all auxiliary functions are bound to an operation or instance variable and vice versa. One of the most useful features of Pax is that it can be used to automatically generate auxiliary functions and the left-hand sides of axioms and post-conditions required to achieve sufficient-completeness. Pax also generates the accessed-by and modified-by functions for state variables.

There are still some verification tasks that Pax is not yet able to perform. Pax is not yet able to verify that a specification conforms to its declared inheritance type or to generate the inheritance type if the signatures, pre-conditions, and post-conditions are fully specified. Pax is also unable to verify a class specification for consistency,
sufficient-completeness, or the validity of its invariants. These tasks require a logic engine, such as that provided by verification systems, to test the predicate expressions that determine subtype and parameterized type inheritance, consistency, and sufficient-completeness.

**Summary**

AxSL, an Axiomatic Specification Language, extends traditional algebraic axiom methods to support basic concepts of object-oriented languages, such as procedural operations, object state, and inheritance. As an axiomatic specification language, AxSL supports the specification of classes for which there are no obvious mathematical models, but it also supports model-based specification, using abstract state information, when appropriate. It supports the specification of operation behavior using both post-conditions, common in mathematical model specifications, and axioms.

To bridge the gap between procedural operations and functional axioms, AxSL introduces auxiliary functions, each of which describes the effect of an operation on a single modifiable parameter or return value. The behavior of each auxiliary function is defined axiomatically.

AxSL uses natural language keywords and phrases to make specifications easier to read and write. Since post-conditions are usually easier to read than axioms, AxSL encourages the use of post-conditions to represent operation behavior when the post-condition expression has no side effects. The **MODIFIES-AT-MOST** clause allows specification writers to leave out post-condition expressions that define parameters not modified by the operation. The ability to use functional operations directly in post-conditions and axioms, rather than creating auxiliary functions for them, makes specifications shorter, and therefore easier to write. Some hints for developing specifications that are easier to write and also easier to read are:

- Use post-conditions when possible, axioms when necessary.
- Use functional operations when possible, auxiliary functions when necessary.
- Pre-conditions eliminate the need for some axioms.
- Do not repeat inherited information.

The most difficult aspect of composing specifications is deciding which axioms and post-conditions need to be defined. Sufficient-completeness is an important concept that can be used as a guideline in developing axioms and post-conditions or in determining whether a specification contains all the axioms and post-conditions necessary to specify its behavior. Pax is a prototype parsing tool that can be used to automatically generate auxiliary functions and the left-hand sides of axioms and post-conditions required to achieve sufficient-completeness.
References


Appendix: AxSL Template

Abstract class: CLASS
CLASS CLASS

Is A:
Is A Subtype Of:
Is A Parameterized Subclass Of:
Is A Selective Subclass Of:
Overrides:
Excludes:

Uses externally defined types:

Has external interface:
Has operations:
  con mk-class () \rightarrow r: CLASS
Pre-condition:
Constructs: self
Post-condition: r = self
Modifies: basic-mod (p1: Type1, \ldots, pJ: TypeJ) \rightarrow r: TypeR
  (* Comment *)
Pre-condition:
Modifies-At-Most: self
Let: \textit{r}' = self
Uses operations:
Post-condition:
Obs: basic-obs (pX: TypeX, \ldots, pY: TypeY) \rightarrow o: TypeO
Pre-condition:
Post-condition:
Basic modifiers: mk-class(), basic-mod (Type1, \ldots, TypeJ)
Basic observers: basic-obs (TypeX, \ldots, TypeY)

Has auxiliary functions:
aux-class-after-basic-mod (CLASS, Type1, \ldots, TypeJ) \rightarrow CLASS
Basic modifiers: mk-class(), aux-class-after-basic-mod (CLASS, Type1, \ldots, TypeJ)
Basic observers: basic-obs (TypeX, \ldots, TypeY)
Axioms:
Object invariant:
Class invariant:
For all \textit{r} \in CLASS
LHS = RHS

Has subclass-visible interface:

Has friend interface:
Has friends:

Has internal interface:
Has state:
InstanceVar: TypeZ
Accessed by: access-instanceVar (CLASS) \rightarrow TypeZ
Modified by: modify-instanceVar (CLASS, TypeZ) \rightarrow CLASS
Basic modifiers:
Basic observers:

End CLASS

Other keywords and operators:
- TYPE: BOOLEAN, INTEGER, CHARACTER, REAL, STRING
- AND, OR, NOT, IF, THEN, ELSE, ENDIF
- +, -, *, /, <, >, <=, >=, ==, => (or ⇒)
- TRUE, FALSE, SELF
- integer, floating point, character, and string constants