Extending Algebraic Axiom Techniques to Handle Object-Oriented Specifications

Alyce Brady, Member, IEEE
David R. Musser, Member, IEEE Computer Society
David L. Spooner, Member, IEEE

August 2, 1999
Abstract

This paper describes AxSL, an Axiomatic Specification Language that extends algebraic axiom methods to support object-oriented concepts such as procedural operations, object state, and inheritance. To bridge the gap between procedural operations and functional axioms, we introduce auxiliary functions, each of which describes the effect of an operation on a single modifiable parameter or return value. We then demonstrate how auxiliary functions fit into the development of proof goals for verifying the partial correctness of object-oriented procedures.

To make specifications easier to read and write, AxSL uses natural language keywords and phrases and supports both post-conditions, common in mathematical model specifications, and axioms. AxSL uses the concept of sufficient-completeness to help guide the development and verification of specifications. AxSL also supports a number of features that are not universal among object-oriented languages, such as multiple inheritance, multiple interfaces to a class, and shared class attributes.

Index Terms: formal methods, specification languages, algebraic axioms, sufficient-completeness, components, object-oriented, procedural operations, auxiliary functions
1 Introduction

Specification languages have generally fallen into two categories: algebraic axiom specifications and mathematical models. Algebraic axioms have the advantage that they are representation-independent, defining the behavior of operations in terms of their effects on other operations. A major disadvantage, however, is that 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, but they require the specifier to define classes and the behavior of their operations in terms of an underlying abstract data representation.\(^1\) For some objects, especially real-world, complex objects, these mathematical models seem to be artificially contrived.

This paper describes AxSL, an Axiomatic Specification Language that extends algebraic axiom techniques to object-oriented classes. In particular, AxSL uses auxiliary functions to define the behavior of procedural operations axiomatically. The paper also shows how AxSL, like Larch, uses the concept of sufficient-completeness to help specifiers with the development and verification of specifications. Finally, the paper briefly describes how AxSL supports inheritance (detailed information can be found in [1] and [2]) and other object-oriented features.

2 Algebraic Axioms vs. Mathematical Models

Algebraic axiom specification languages allow a user to define an abstract data type by giving the signatures of the operations on the type and a set of equations that define the behavior of the operations in terms of one another. If the set of operations includes all those necessary to generate all possible instances of the abstract data type (the generating set of operations), 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. 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. Larch [3] is a well-known example of an algebraic axiom language; 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).

The primary advantage of the mathematical modeling technique is that post-conditions are generally easier to understand than axioms, both because they

\(^1\)Throughout this paper we use the general term operation to encompass pure functions, pure procedures, and procedures with return values.
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 of post-conditions is that they 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 of the mathematical modeling technique is that although the state variables that describe a mathematical model are usually understood to be abstract, the method can be used to specify the actual data representation if that is appropriate for the particular application.

The main advantage 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? Another 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 originally defined by Guttag and Horning [6] and described in more detail below). Finally, algebraic axiom specifications do not run the risk of influencing the actual data representation of an implementation in an inappropriate way.

3 AxSL

AxSL, an Axiomatic Specification Language developed at Rensselaer Polytechnic Institute for specifying object-oriented library components, is based on Larch. Like Larch, AxSL is a keyword-based notation rather than relying on a myriad of Greek letters and mathematical symbols. Figures 1 and 2 demonstrate the two levels of a Larch specification for a common Stack type: an abstract, language-independent specification written in the the Larch Shared Language (LSL) [3], and a language-specific interface specification (in this case using Larch/C++ [7]) that provides a bridge to a specific implementation of the LSL abstract data type. It is interesting to note that the LSL specification defines operation behavior using functional algebraic axioms (in the asserts section), while 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).

Figure 3 shows the AxSL specification of the Stack abstract data type. Although the AxSL specification also defines operation signatures and behavior (described in detail below), 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. In fact, one of AxSL’s most important features is the
STACK: trait

introduces

- NEW: \( \rightarrow \text{STK} \)
- PUSH: \( \text{STK} \times \text{ELT} \rightarrow \text{STK} \)
- POP: \( \text{STK} \rightarrow \text{STK} \)
- TOP: \( \text{STK} \rightarrow \text{ELT} \)
- SIZE: \( \text{STK} \rightarrow \text{INT} \)
- EMPTY: \( \text{STK} \rightarrow \text{BOOL} \)

asserts

STK generated by \([\text{NEW}, \text{PUSH}]\)

for all \( s \in \text{STK}, e \in \text{ELT} \)

- \( \text{POP}(\text{NEW}) = \text{NEW} \)
- \( \text{POP}(\text{PUSH}(s, e)) = s \)
- \( \text{TOP}(\text{PUSH}(s, e)) = e \)
- \( \text{SIZE}(\text{NEW}) = 0 \)
- \( \text{SIZE}(\text{PUSH}(s, e)) = \text{SIZE}(s) + 1 \)
- \( \text{EMPTY}(s) = (\text{SIZE}(s) = 0) \)

end STACK

Figure 1: Larch Shared Language Specification for Stack

way in which it blends the algebraic axiom nature of LSL with the object-oriented nature of Larch/C++, while remaining language independent.

As an object-oriented specification language, AxSL uses a procedural signature notation, supports the basic object-oriented concepts of inheritance, operation overloading, and dynamic binding, and allows the specification of data representations when appropriate. These topics are discussed briefly below and in more detail in [1] and [2]. The primary focus of this paper, however, is the way that AxSL extends functional algebraic axiom techniques to specify the behavior of object-oriented operations.

Operation Signatures in AxSL

In the Stack example, the object-oriented nature of AxSL is most obvious in its operation signatures. AxSL operations use a procedural signature notation, in which the object is an implicit parameter to all operations and an implicit return value from all modifying operations. AxSL operations may be pure functions, with a single return value, or may be procedures that modify one or more parameters (including the underlying object). They may also combine the two, as in the pop operation in Figure 3, which pops the top element from the stack and returns that element to the calling operation. The Larch Shared Language, on the other hand, is a functional language in which operations may construct a new, modified version of an object (as in POP in Figure 1), but may not have side effects and thus may not modify the object itself. A Larch interface language follows the same style for defining operation signatures as its corresponding implementation language; in particular, Larch/C++ is an object-oriented, procedural language, just like AxSL.
template <class T>
class Stack {
  public:
    // uses STACK (T for int, Stack for T)
    // spec int MAXSIZE;
    // spec int nhrElts;
    // spec T data[MaxSize];
    // invariant nhrElts >= 0;

    Stack O;
    // behavior {
    //   ensures nhrElts = 0;
    // } //

    void push (const T & e); // behavior {
    //   requires assigned(nhrElts, pre)
    //   \land assigned(data[e\ldots, pre])
    //   \land modifies nhrElts, data[e\ldots];
    //   ensures \forall i, 0 \leq i < nhrElts, \ldots
    //   \land data[e\ldots, [i] = data[e\ldots[[i]]];
    // } //

    T pop O;
    // behavior {
    //   requires assigned(nhrElts, pre)
    //   \land assigned(data[e\ldots, pre]) \land nhrElts > 0;
    //   modifies nhrElts, data[e\ldots];
    //   ensures result = data[e\ldots[[nhrElts-1]]
    //   \land nhrElts' = nhrElts - 1
    //   \land \forall i, 0 \leq i < nhrElts - 1 \implies
    //   \land data[e\ldots[[i] = data[e\ldots[[i]]];
    // } //

    T top O const;
    // behavior {
    //   requires assigned(nhrElts, pre)
    //   \land assigned(data[e\ldots, pre]) \land nhrElts > 0;
    //   modifies nothing;
    //   ensures result = data[e\ldots[[nhrElts-1]];
    // } //

    int size O const;
    // behavior {
    //   requires assigned(nhrElts, pre)
    //   \land assigned(data[e\ldots, pre]);
    //   modifies nothing;
    //   ensures result = nhrElts;
    // } //

    boolean isEpty O const;
    // behavior {
    //   requires assigned(nhrElts, pre)
    //   \land assigned(data[e\ldots, pre]);
    //   \land modifies nothing;
    //   ensures result = (nhrElts = 0);
    // } //
};

Figure 2: Larch/C++ Specification for Stack
CLASS Stack[T::TYPE]
EXTERNAL INTERFACE
HAS OPERATIONS:
CON: mkStack (T::TYPE) -> Stack[T]
   POST-CONDITION: size() = 0
MOD: push (elt: T) ->
   MODIFIERS-AT-MOST: SELF
   LET: s' = SELF
   POST-CONDITION: size() = (s'.size() + 1) AND top() = elt
   AND SELF = auxStackAfterPush (s')
MOD: pop () -> elt: T
   PRE-CONDITION: not empty()
   MODIFIERS-AT-MOST: SELF
   LET: s' = SELF
   POST-CONDITION: elt = s'.top()
   AND SELF = auxStackAfterPop (s')
OBS: top () -> elt: T
   PRE-CONDITION: not empty()
OBS: size () -> Integer
OBS: empty () -> b: Boolean
   POST-CONDITION: b = ( size() = 0 )
BASIC MODIFIERS: mkStack(TYPE), push(T)
BASIC OBSERVERS: top(), size()
HAS AUXILIARY FUNCTIONS:
auxStackAfterPush (Stack[T], T) -> Stack[T]
auxStackAfterPop (s:Stack[T]) -> Stack[T]
PRE-CONDITION: not s.empty()
BASIC MODIFIERS: mkStack(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]

Key: CON indicates constructor operations
     Mod indicates modifying operations
     Obs indicates observer operations

Figure 3: AxSL Stack Specification
Since AxSL is not a functional language, it must make a distinction between constructor operations, those that actually 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. Since operations such as pop may modify the underlying object and also the other operation arguments, AxSL specifications include a MODIFIES-AT-MOST clause to indicate exactly which arguments of a given operation may be changed. The Larch Shared Language, as a functional language, does not need to make these distinctions. A Larch interface language uses the conventions of its corresponding implementation language. Larch/C++ supports the same procedural style as AxSL; it denotes constructors by giving them the same name as the class, uses the const keyword to identify accessor operations (all others are modifiers), and uses C++'s pass-by-reference mechanism for identifying arguments that may be changed.

Specifying Behavior in AxSL

The ability to define procedural operations is a critical element in creating object-oriented specifications, but it makes it difficult to use algebraic axioms to define operation behavior. 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. This axiom does not work in the AxSL specification for Stack because the procedural push does not return anything and pop has no parameters and returns the popped element rather than the modified stack. AxSL introduces auxiliary functions to bridge the gap, allowing developers to define procedural operations in terms of functional axioms. Each result or side effect of an operation has a separate auxiliary function associated with it. Furthermore, the implicit first parameter of the object itself is made explicit as a new initial parameter to the auxiliary functions. For example, in the AxSL Stack specification, the axiom

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

defines the same behavior as the LSL axiom above, but uses auxiliary functions associated with the push and pop operations rather than the operations themselves. In addition, auxiliary functions allow developers to specify data representations similar to mathematical model specifications when that is desirable. Auxiliary functions are defined further in the next two sections.
Model-based languages, such as VDM, Z, and Larch/C++, specify operation behavior in post-conditions on the operation rather than in mathematical equations in a separate section of the specification, as axiomatic languages do. AxSL supports both styles. In general, an axiom whose left-hand side consists of one function applied to the result of another may be expressed as a post-condition clause on the latter operation. For example, the Larch Shared Language axiom

$$\text{size}(\text{push}(s,e)) = \text{size}(s) + 1$$

can be expressed in AxSL as a post-condition on the \text{push} operation:

$$\text{size}() = (s'.\text{size}() + 1)$$

where $s'$ refers to the original value of $s$. Similarly, an axiom whose left-hand side contains a call to only one operation can be expressed as a post-condition on that operation. For example, the LSL axiom

$$\text{empty}(s) = (\text{size}(s) = 0)$$

can be expressed in AxSL as a post-condition on the \text{empty} operation:

$$b = (\text{size}() = 0)$$

where $b$ refers to the return value of \text{empty}. In order to support the semantic definition of operations through post-conditions, AxSL uses \text{LET} expressions on modifying operations to introduce names, such as $s'$ in the example above, that can be used to refer to the initial values of modifiable objects. These include the implicit first parameter, called \texttt{SELF}, and any other modifiable parameters as specified in the \texttt{MODIFIES-AT-MOST} clause.

Generally, AxSL allows the specification developer to choose whether to represent the behavior of non-modifying functions in axioms, as in Larch, or in equivalent post-conditions. In the AxSL Stack specification, however, there is one axiom that is not specified in a post-condition. AxSL was designed to be part of a larger system in which expressions in the specification are associated with assertions in the implementation. Expressions that modify state should not appear in executable assertions, and so the examples in this paper do not include such expressions in operation post-conditions either. In the Stack specification, the LSL axiom

$$\text{pop}(\text{push}(s,e)) = s$$

does not become the post-condition

$$s'.\text{pop}() = s'$$
on the AxSL push operation, since verifying the push post-condition by invoking pop would undo the effect of the operation. Thus, the examples shown here only include non-modifying operations and auxiliary functions (which are functional by definition) in post-condition expressions. Invoking these expressions does not have an effect on the system.

4 The Case for Auxiliary Functions

Traditionally, algebraic axioms have been used with functional operation signatures. All operations in the Larch Shared Language, for example, are functional. Object-oriented operations are inherently procedural rather than functional, since a modifying operation changes the state of the object to which the operation is tied, rather than creating a new instance with a different state. Since object-oriented operations need not return the modified object, they frequently return some other value. This means that these operations may have more than one result, or effect. In fact procedural object-oriented operations may also modify their arguments, so they may have as many results as arguments, including the implicit first argument of the object itself, and one result for the return value. In the case of the pop operation, one common object-oriented implementation would have the two results shown in Figure 3, namely, a return value (the top element being removed) and a side-effect (the modified stack object).

This procedural nature of object-oriented operations means that they cannot be used in functional-style algebraic axioms. One solution to this problem would be to create a new, procedural syntax for object-oriented axioms, including a mechanism, such as let expressions, for attaching names to initial and intermediate results. This would amount to developing a limited programming language, with a formally defined semantics.

Another solution is to create a set of auxiliary functions for each operation, one function for each operation result. The number of results is the number of parameters listed in the Modifies-At-Most clause, including the implicit first parameter of the object itself, plus the return value if there is one. Each auxiliary function has the same arguments and pre-condition as the operation, with the operation’s implicit first parameter made explicit. Being pure functions, auxiliary functions have no side effects and return the single result with which they are associated. Constructor operations, those that create new instances of a class, are exceptions; although auxiliary functions may be defined for them, they do not require an extra first parameter since the object to be created is not passed to them.

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. Section 5 demonstrates more formally how auxiliary functions can be used to specify the behavior of operations.
Figure 4 shows the full set of auxiliary functions that define the Stack class. The 
pop operation, for example, has two auxiliary functions, one for each result: 
auxStackAfterPop, which returns the “changed” stack (actually a new stack, since this 
is a true function), and auxPop, which returns the top element.

The behavior of the auxiliary functions is defined in the Axioms section of the 
specification using algebraic axioms equivalent to those found in the Larch Shared 
Language specification of Figure 1. Axioms consist primarily of functional composition 
and the equality operator. The equality operator is an equality of function rather than 
an equality of object identity. Two objects are “equal” if they have the same observable 
behavior. Object identity is only a consideration if the object identifier is an observable 
characteristic, for example, if there is an observer operation that returns the object 
identifier.

The axioms define the behavior of the auxiliary functions, but do not directly 
define the behavior of the original operations. To do that, we define each operation in 
terms of its auxiliary functions in its post-condition. Specifically, for each operation 
result—return value or modifiable parameter in the MODIFIES-AT-MOST clause—there 
is an expression in the operation’s post-condition defining the final value of the result as 
the value of the associated auxiliary function. The post-condition on the pop operation 
in Figure 4, for example, contains two expressions defining the final values of its results 
in terms of its auxiliary functions.

Note that the AxSL Stack specification in Figure 3 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. One reason to use auxiliary functions for functional 
operations, though, would be to support the traditional syntax of nested function calls 
in complex axiomatic expressions, as opposed to the object-oriented dot notation of 
class operations used by AxSL (similar to that found in C++ [8], Java [9], or Eiffel [10]). 
For example, the pre-condition on auxStackAfterPop in Figure 4 is NOT auxEmpty(s) 
rather than NOT s.empty(), as in Figure 3.

Another difference between Figures 3 and 4 is that the former specification uses 
the functional top operation as the auxiliary function associated with pop’s return 
value. If the first Stack specification had not included the top operation (which 
Figure 4 does not), the auxPop auxiliary function would have been necessary. A final 
difference is that Figure 4 does not make use of the ability to define observer functions 
in post-conditions, as Figure 3 does. 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, since assertions should not include 
expressions with side-effects. Furthermore, specifying observer behavior in
CLASS Stack[T: TYPE]

EXTERNAL INTERFACE
HAS OPERATIONS:
  CON: mkStack (T: TYPE) → s: Stack[T]
  CONSTRUCTS: self
  POST-CONDITION: s = auxMsStack(T)
  MOD: push (elt: T) →
    LET: s' = self
    MODIFIES: At-MOST: self
    POST-CONDITION: self = auxStackAfterPush(s', elt)
  MOD: pop () → elt: T
    PRE-CONDITION: NOT empty()
    LET: s' = self
    MODIFIES: At-MOST: self
    POST-CONDITION: elt = auxPop(s')
      AND self = auxStackAfterPop(s')
  OBS: size () → i: Int
    POST-CONDITION: i = auxSize(self)
  OBS: empty () → e: Bool
    POST-CONDITION: e = auxEmpty(self)
BASIC MODIFIERS: mkStack(TYPE), push(T)
BASIC OBSERVERS: pop(), size()

AUXILIARY FUNCTIONS:
  auxMsStack(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: auxMsStack(TYPE), auxStackAfterPush(Stack[T], T)
BASIC OBSERVERS: auxPop(Stack[T]), auxSize(Stack[T])

AXIOMS:
  OBJECT INVARIANT: auxSize(self) ≥ 0
  FORALL T: TYPE, s: Stack[T], elt: T
  auxStackAfterPop(auxStackAfterPush(s, elt)) = s
  auxPop(auxStackAfterPush(s, elt)) = elt
  auxSize(auxMsStack(T)) = 0
  auxSize(auxStackAfterPush(s, elt)) = auxSize(s) + 1
  auxEmpty(s) = ( auxSize(s) = 0 )

END Stack[T: TYPE]

Figure 4: Full Set of Stack Auxiliary Functions

12
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.

Auxiliary functions can also be used to specify data representations, as in mathematical model specifications. A developer may add a **Has State** section to the class specification and then, for each state variable, introduce two auxiliary functions corresponding to the implicit operations of accessing and modifying the state variable. The signatures of these auxiliary functions would look like

\[
\begin{align*}
\text{accessStateVar} & \quad (c: \text{ClassType}) \rightarrow (\text{var: \text{StateVarType}}) \\
\text{modifyStateVar} & \quad (c: \text{ClassType}, \text{newVal: \text{StateVarType}}) \rightarrow \.cleanup{c: \text{ClassType}}
\end{align*}
\]

For example, if we were to add a mathematical sequence as an abstract data representation for the Stack class, the auxiliary functions would be

\[
\begin{align*}
\text{accessSeq} & \quad (s: \text{Stack}) \rightarrow (\text{val: \text{Sequence[T]}}) \\
\text{modifySeq} & \quad (s: \text{Stack}, \text{newVal: \text{Sequence[T]}}) \rightarrow s: \text{Stack}
\end{align*}
\]

In this case, `modifySeq` would be a basic modifier rather than `push`, and the relationship between them could be expressed with the following axiom:

\[
\begin{align*}
\text{FORALL } T: \text{type}, s: \text{Stack}, e: T \\
\text{auxStackAfterPush} (s, e) &= \.cleanup{\text{modifySeq} (s, \text{concat}([e], \text{accessSeq} (s)))}
\end{align*}
\]

where `concat` is the concatenation operation defined on mathematical sequences and `[e]` is the sequence containing only `e`.

5 Specifying Operation Semantics Using Auxiliary Functions

How do we know that the behavior of an operation as a whole can be defined as the sum of the behaviors of its auxiliary functions, as claimed above? In this section we will demonstrate how to model operations as procedures and then show how auxiliary functions correspond to the definition of assignment semantics for procedures. Finally, we will generalize the use of auxiliary functions for specifying the behavior of operations beyond the special case of procedures that satisfy assignment semantics.
5.1 Formally Defining the Behavior of Operations

As noted earlier, an AxSL operation may be a pure function if it does not modify any arguments but returns a value, it may be a procedure if it has no return value but modifies the underlying object or one or more arguments, or it may be a procedure with a return value.

Let

\[ op(x_1 : T_1, \ldots, x_n : T_n) \rightarrow x_{n+1} : T_{n+1} \]

**PRE-CONDITION:** PRE

**MODIFIES-At-MOST:** \( y_1, \ldots, y_k \)

**POST-CONDITION:** POST

be an operation declaration for objects of class \( C \), with implicit parameter \( x_0 \) (also known as **self**) of type \( C \), explicit formal parameters \( x_1, \ldots, x_n \) which are distinct, simple variable names of type \( T_1, \ldots, T_n \), and with optional formal result \( x_{n+1} \) of type \( T_{n+1} \). (If \( x_{n+1} \) does not exist, \( op \) is a pure procedure.) Assume for now that \( op \) is deterministic and that it does not reference or modify any global variables, nor any local variables whose values are maintained between calls (static variables). Assume also that there is no aliasing among the arguments to \( op \) when it is called, i.e., that all of the explicit and implicit actual parameters to \( op \) are distinct. In a later section we will see how to extend operation specifications to allow non-determinism and the use of global and static variables, and how the aliasing requirement may be relaxed under some conditions.

The pre-condition, **PRE**, is a predicate that specifies a condition that must be met before the operation is called in order for the operation to behave as specified, **POST** is a predicate that must be met after the operation has been called. The only variables that **PRE** may refer to are the implicit and explicit formal parameters, \( x_0, \ldots, x_n \). If there is an object invariant for the class (as in both Figures 3 and 4), then that is an implicit part of the pre-condition:

\[ \text{PRE}' = \text{PRE} \land I \]

where \( I \) is the object invariant. The complete pre-condition, **PRE'** is a predicate (boolean function) with arguments \( x_0, \ldots, x_n \), which we can call **op-pre**:

\[ \text{PRE}' = op_{\text{pre}}(\bar{x}) \]

where \( \bar{x} \) represents the vector \( (x_0, \ldots, x_n) \). For example, the complete pre-condition for the **pop** operation in Figure 3 is

\[ \text{PRE}'_{\text{pop}} = \text{pop}_{\text{pre}}(\text{SELF, elt}) \]

\[ = \text{not SELF.empty}() \land \text{SELF.size}() \geq 0 = \text{SELF.size}() > 0 \]

14
The modifies-at-most list for an operation is the set of names of all formal
parameters potentially altered by the body of \( op \), i.e.,
\[
\forall j: 1 \leq j \leq k: y_j \in \{ x_0, x_1, \ldots, x_n \}
\]
If \( op \) is a modifying operation, then \( x_0 \) (i.e., \( \text{SELF} \)) must be included in the
modifies-at-most list. For convenience, \( Y \) is used to refer to the complete
modifies-at-most list, i.e.,
\[
Y = \{ y_1, \ldots, y_k \}
\]
(If \( Y \) is empty, \( op \) is a pure function.)

The post-condition, \( \text{POST} \), is a predicate that specifies the state after the call to
\( op \) has taken place. It may refer not only to the final values of the formal parameters
after the call to the operation, but also to the formal parameters' initial values and the
operation's return value. The initial values of the parameters are represented by primed
variables, e.g., \( x'_i \), \( 1 \leq i \leq n \). The modifies-at-most list also forms part of the complete
post-condition, since it specifies not only which parameters may be altered, but also, by
omission, which parameters may not. (Again, assume that no aliasing is allowed; see
Section 5.5.) Furthermore, if there is an object invariant for the class, then that is an
implicit part of the post-condition also. The complete post-condition, taking the
modifies-at-most list and object invariant into account, is
\[
\text{POST}' = \text{POST} \land (\forall i: 0 \leq i \leq n: (x_i \notin Y \Rightarrow x_i = x'_i)) \land I
\]
where \( I \) is the object invariant. Since the complete post-condition ensures that
parameters not in the modifies-at-most list will be unchanged, the predicate \( \text{POST}' \)
may be expressed in terms of the initial values of the formal parameters, the final
values of those parameters in \( Y \), and the return value:
\[
\text{POST}' = op\text{-post} (\vec{x}', \vec{y})
\]
where \( \vec{x}' \) represents the vector of initial values \( (x'_0, \ldots, x'_n) \) and \( \vec{y} \) represents the vector
of final values \( (y_1, \ldots, y_k, x_{n+1}) \). If \( op \) does not have a return value, i.e., \( x_{n+1} \) does not
exist, then \( \vec{y} \) is the vector \( (y_1, \ldots, y_k) \). Continuing with our \( \text{pop} \) example from Figure 3,
the complete post-condition for \( \text{pop} \) is
\[
\text{POST}'_{\text{pop}} = \text{pop\text{-post} (SELF', \text{SELF}, \text{elt})}
\quad = (\text{elt} = \text{SELF}'.\text{top()} \land \text{SELF} = \text{auxStackAfterPop} (\text{SELF'}))
\quad \land \text{SELF}.\text{size()} \geq 0)
\]

\( \text{PRE}' \) and \( \text{POST}' \) together form a contract [11] [10]. Ensuring that the
pre-condition \( \text{PRE}' \) is satisfied is the obligation of the caller, while ensuring that the
post-condition \( \text{POST}' \) is satisfied is the obligation of the operation implementer. This
corresponds to Hoare’s notation for verifying the partial correctness of a statement:

\[ \{ Q \} S \{ R \} \]

which states that if a program state satisfies \( Q \) and if statement \( S \) terminates, then the predicate \( R \) is true after \( S \) terminates [12]. Since an AxSL operation may have a return value, however, it constitutes an expression with side effects rather than a statement.

### 5.2 Modeling Operations as Procedures

Since operations may have multiple results, they cannot be modeled as functions. They can, however, be modeled as pure procedures if the return value of the operation is represented as an extra modifiable parameter.\(^2\) (If an operation does not have a return value, it is already a procedure.)

Let

\[
\text{opProc } (x_0 : C, \ x_1 : T_1, \ldots, \ x_a : T_a) \\
\text{PRE-CONDITION: } \text{PRE} \\
\text{MODIFIES-AT-MOST: } y_1, \ldots, y_b \\
\text{POST-CONDITION: } \text{POST}
\]

be a procedure declaration corresponding to the operation declaration \( \text{op} \) defined at the beginning of Section 5.1, where \( a = n + 1, b = k + 1 \), and \( y_{k+1} = x_{n+1} \) if \( \text{op} \) has a return value, and \( a = n, b = k \) otherwise. The pre- and post-condition predicates for \( \text{opProc} \) are the same as those for \( \text{op} \):

\[
\text{PRE'} = \text{PRE} \land I \\
= \text{op.pre} (\bar{x})
\]

\[
\text{POST'} = \text{POST} \land (\forall i: 0 \leq i \leq n : (x_i \notin Y \Rightarrow x_i = x'_i)) \land I \\
= \text{op.post} (\bar{x'}, \bar{y})
\]

where \( \bar{x} \) is the vector \((x_0, \ldots, x_n)\), \( \bar{x'} \) is the vector of initial values \((x'_0, \ldots, x'_n)\), and \( \bar{y} \) is the vector of modifiable parameters \((y_1, \ldots, y_b)\). Notice that there is no initial value for the optional parameter representing the return value, \( x_{n+1} \); in other words, \( x'_{n+1} \) is undefined.

Using this pure procedural version of the original operation \( \text{op} \), we have

\(^2\)The actual notation in [12] is \( P(Q)R \), where \( P \) is the pre-condition, \( Q \) is the statement, and \( R \) is the post-condition.

\(^3\)The semantics of a call to an operation with a return value and a call to a procedure with an extra parameter are not exactly the same, however. The former, as an expression, may be embedded in complex expressions whereas the latter can only appear as an independent statement. Section 5.5 addresses this difference.
\{ \text{PRE} \land (\vec{x} = \vec{x}') \} \ \text{opBody} \ \{ \text{POST} \}

where \text{opBody} is the body of operation \text{opProc}. This captures the procedure developer’s side of the construct.

We can express the semantics of \text{opProc} to be verified by the procedure developer as the predicate \text{semantics}(\text{opProc}, \text{PRE}', \text{POST}') , which is true if

\((\text{PRE} \land (\vec{x} = \vec{x}')) \Rightarrow \text{wlp(\text{opBody}, \text{POST}')}

where \text{wlp} is the \textit{weakest liberal precondition}, (i.e., weakest precondition except that termination is not required [13, p. 21]).

In the special case that a procedure satisfies \textit{assignment semantics}, its formal semantics can be defined as a function from its input parameters to its output (modifiable) parameters [14]. Informally, a procedure satisfies assignment semantics if the only effect of the procedure is to assign new values to its modifiable arguments (no side effects) and if the new values can be expressed as a tuple which is a function of the procedure’s parameters. When this condition is satisfied, the procedure can be modeled as an assignment statement and, with the assumption that there is no aliasing, its formal semantics derived from assignment semantics.

Bernstein and Lewis provide a formal definition of assignment semantics for procedures that have no explicit post-condition. We use the predicate \text{assign}(\text{opProc}, f, \text{PRE}') to indicate that procedure \text{opProc} satisfies assignment semantics when pre-condition \text{PRE}' is met and that \( f \) is the function mapping the input parameters of \text{opProc} to its modifiable arguments. Predicate \text{assign}(\text{opProc}, f, \text{PRE}') is true if

\((\text{PRE} \land (\vec{x} = \vec{x}')) \Rightarrow \text{wlp(\text{opBody}), (\forall i : 0 \leq i \leq n : (x_i \notin Y \Rightarrow x_i = x_i') } \land (\vec{y} = f(\vec{x}'))

where \( \vec{x} \) is the vector \((x_0, \ldots, x_n)\), \( \vec{x}' \) is the vector \((x'_0, \ldots, x'_n)\), and \( \vec{y} \) is the vector \((y_1, \ldots, y_b)\), as before [14, pp. 79-82].\(^4\) Again, \( \vec{x} \) and \( \vec{x}' \) do not include \( x_{n+1} \) because there is no initial value for \( x_{n+1} \). When a procedure with an explicit post-condition, \text{POST}', satisfies assignment semantics, we have

\text{semantics}(\text{opProc}, \text{PRE}', \text{POST}') \Rightarrow \text{assign}(\text{opProc}, f, \text{PRE}')

or

\(( (\text{PRE} \land (\vec{x} = \vec{x}')) \land \text{wlp(\text{opBody}, \text{POST}')) } \Rightarrow \text{wlp(\text{opBody), (\forall i : 0 \leq i \leq n : (x_i \notin Y \Rightarrow x_i = x_i') } \land (\vec{y} = f(\vec{x}'))

\(^4\)We have modified Bernstein and Lewis’s notation slightly to correspond more closely to AxSL’s specification of operations.
5.3 The Semantics of Procedure Calls

Let

\[ \text{opProc} \left( u_0, \ldots, u_a \right) \text{ alters } v_1, \ldots, v_b; \]

be a call to procedure \( \text{opProc} \), where \( u_0, \ldots, u_a \) are the actual parameters corresponding to \( x_0, \ldots, x_a \), and \( v_1, \ldots, v_b \) are those members of \( u_0, \ldots, u_a \) corresponding to the formal parameters \( y_1, \ldots, y_b \) in the modifies-at-most list for \( \text{opProc} \). (The \( v_i \)'s must be variables, not general expressions.) The proof goal for verifying the conditional correctness of this procedure call is

\[ \{ U \} \text{ opProc} \left( u_0, \ldots, u_a \right) \text{ alters } v_1, \ldots, v_b; \{ V \} \]

where \( U \) and \( V \) are the actual pre- and post-conditions.

If procedure \( \text{opProc} \) satisfies assignment semantics but has no other explicit post-condition and no object invariant, the following inference rule specifies the formal semantics of its procedure call [14]:

\[
\frac{\text{assign}(\text{opProc}, f, \text{PRE}', U \Rightarrow (V_{j(n)}^p \land \text{PRE}'_{n}))}}{\{ U \} \text{ opProc}(u_0, \ldots, u_a) \text{ alters } v_1, \ldots, v_b; \{ V \}}
\]

where \( n \) is the vector of the initial values of the actual arguments \( (u_0, \ldots, u_n) \), not including \( u_{n+1} \) even if it exists, and \( \bar{v} \) is the vector \( (v_1, \ldots, v_b) \). Once such a procedure has been shown to satisfy assignment semantics (which should be the responsibility of the procedure developer), we can express the procedure caller's verification sub-goals as:

\[ U \Rightarrow \text{PRE}'_{\bar{v}} \]

and

\[ U \Rightarrow V_{j(n)}^p \]

The post-condition, \( \text{POST}' \), for \( \text{opProc} \), may, however, be a stronger condition than that the procedure satisfies assignment semantics. If a procedure with post-condition \( \text{POST}' \) also satisfies assignment semantics, \( \text{i.e., if the procedure developer has shown that} \)

\[ \text{semantics}(\text{opProc}, \text{PRE}', \text{POST}') \Rightarrow \text{assign}(\text{opProc}, f, \text{PRE}') \]

then the procedure caller's verification sub-goals become

\[ U \Rightarrow \text{PRE}'_{\bar{v}} \]
and

\[(U \land \text{POST}^I_{x,y}) \Rightarrow V^I_{f(n)}\]

We can express these in terms of the predicates \(\text{op}_{\text{pre}}\) and \(\text{op}_{\text{post}}\) as

\[U \Rightarrow \text{op}_{\text{pre}}(\tilde{u})\]

and

\[(U \land \text{op}_{\text{post}}(\tilde{u}, f(\tilde{u}))) \Rightarrow V^I_{f(n)}\]

### 5.4 The Role of Auxiliary Functions

The \textit{assign} predicate for procedural operations that satisfy assignment semantics includes the following expression which specifies the new values of the procedure's modifiable parameters:

\[
\tilde{y} = f(\tilde{x})
\]

Each individual modified parameter is the result of composing the appropriate projection function with the function \(f\), \textit{i.e.},

\[
\forall j : 1 \leq j \leq b : y_j = g_j(f(\tilde{x}'))
\]

where \(g_j\) is the projection function from \(\tilde{y}\) to \(y_j\). The auxiliary functions introduced in Section 4 correspond to the application of these various projection functions to \(f\):

\[
\forall j : 1 \leq j \leq b : y_j = \text{aux}_{\text{op}_{\text{y}_{j}}}(\tilde{x}') = g_j(f(\tilde{x}'))
\]

where \(\text{aux}_{\text{op}_{y_j}}\) is the auxiliary function defining the new value for parameter \(y_j\) of procedure \(\text{op}\). For readability, the examples in Figures 3 and 4 use the following naming conventions:

- \(\text{auxClassAfterP} = \text{aux}_{\text{op}_{\text{y}_{0}}} \quad (\text{e.g., auxStackAfterPop})\)
- \(\text{auxP} = \text{aux}_{\text{op}_{\text{y}_{k+1}}} \quad (\text{e.g., auxPop})\)
5.5 Generalizing the Use of Auxiliary Functions

So far, our justification for the use of auxiliary functions to define the behavior of operations has been based on three main assumptions: that the operations we are interested in can be modeled as procedures, that those procedures satisfy assignment semantics, and that procedure calls do not include aliasing. In this section we will generalize our argument to cover a number of cases in which procedures do not satisfy assignment semantics, discuss the problem of aliases, and then finally address the fact that operations are not always procedures.

Procedures that do not satisfy assignment semantics include non-deterministic procedures, those that rely on state information other than what is passed in as parameters, and those that generate side effects other than what is reflected in the modifiable arguments. Procedures that do I/O also do not satisfy assignment semantics, but these are covered by the cases above if we consider input as either an extra source of state information or a source of non-determinism and output as an extra side effect.

**Non-determinism:** Non-deterministic procedures clearly do not satisfy the assignment predicate, which states that the vector of modified arguments is a function of the vector of initial parameter values. Non-deterministic procedures, such as a procedure that retrieves input from a user or some other outside source, have at least one result (i.e., modified argument) that is not a function of the input parameters. In other words, the actual value of the result depends on some outside information; given only the information provided by the input parameters, more than one value is possible.

If the operation is, in fact, deterministic when more information (such as the value input by the user) is available, then the operation can be specified using auxiliary functions. In this case, the extra information becomes an extra input parameter to the auxiliary functions, or at least to those auxiliary functions that need the information in order to be deterministic. This is similar to how the input stream is handled in denotational semantics, where it is treated as a parameter to functions that need it. (See, for example, Gordon’s *The Denotational Description of Programming Languages* [15].)

Consider a `getchar` operation, with no input parameters, that retrieves a single character from a user and returns that character:

\[
\text{getchar} \ () \rightarrow c: \text{Character}
\]

The return value, `c`, of the operation is non-deterministic based on the (non-existent) input parameters, because the specification depends on input from the user. The auxiliary function associated with `c` is

\[
\text{auxgetchar} \ (s: \text{Stream}) \rightarrow c: \text{Character}
\]

which is a deterministic function. The contract for `getchar` becomes

20
getchar () → c: Character  

**POST-CONDITION:**  
\[ c = \text{auxgetchar} \text{ (input)} \]

where `input` is a variable of type `Stream` known to the class with which `getchar` is associated (possibly a part of the object’s state, or a global variable), although not necessarily known to the class user. This solution moves the non-determinism from the auxiliary functions to the equations in the post-condition establishing the relationship between modified parameters and auxiliary functions. Having deterministic auxiliary functions, the specification developer can use axioms to define class behavior in the usual way.

**Global and Static Variables:** Another basic assumption underlying the use of assignment semantics for procedures is the assumption that operations do not reference or modify any global variables, nor local variables whose values are maintained between calls (other than the state of the object, the implicit first parameter). As with non-deterministic procedures, if a procedure were to use global or static variables, its result would not be a function of its input parameters.

If an object-oriented operation does use global variables, they should be considered as extra implicit parameters to the operation and should be specified as extra explicit parameters to the auxiliary functions. This is analogous to how non-determinism is handled. For any global variable that is modified by the operation, there should also be an auxiliary function defining the operation’s effect on that global variable. Local static variables are a bit different. In general, local variables whose values are maintained between operation calls (e.g., C static variables) are unobservable to the specification and therefore of interest only to the implementation. The exception is when such local variables have an observable effect on the behavior of the operation. If, for example, the behavior of an operation is different the first time it is called from subsequent calls, then that behavior should be defined in the specification. In such cases, static local variables can be specified similarly to global variables.

**Aliases:** Reasoning about the behavior of procedures based on assignment semantics is based on the assumption that there is no aliasing among the arguments in a procedure call. If, for example, a parameter `a` not in the `modifies-at-most list` (i.e., a pure input parameter) were an alias for a parameter `b` in the `modifies-at-most list` (a modifiable parameter), the claim in the `assign` predicate that the value of `a` is unchanged by the procedure would not be justified. If two modifiable parameters were aliases, clauses in the post-condition, `POST\text{1}`, might even be violated. Aliases among pure input parameters, on the other hand, are always safe.

If a procedure call may include aliases between modifiable parameters or between modifiable parameters and pure input parameters, then that fact must be taken into consideration, along with knowledge of the procedure’s implementation, when verifying that the result of a procedure invocation satisfies the specifications of the auxiliary functions and, therefore, the specification of the operation. For example, aliasing could
be allowed among the parameters of a swap \((a, b)\) command if the implementation were

\[
    t := a; a := b; b := t;
\]

When \(a\) and \(b\) are aliases, the swap command would have no effect.

**Operations Are Not Procedures:** The final basic assumption in our argument for using auxiliary functions to define the behavior of AxSL operations is that operations can be modeled as procedures. This allowed us to make use of the definition of assignment semantics for procedures (and other code fragments). In Section 5.1, however, operations were defined more generally as supporting modified parameters (as in procedures) and return values (as in functions). There is an important semantic difference between the operation, \(op\), and its procedural representation, \(opProc\). The former, as an expression, can be embedded in complex expressions, whereas the latter cannot.

To understand the semantics of an embedded operation call in terms of a procedure, one would need to introduce a temporary variable, call the procedure with the temporary variable as the extra parameter representing the return value, and then use the temporary variable in place of the operation in the complex expression. A complex expression may, in fact, have multiple sub-expressions that must be broken out, evaluated, and stored in temporary variables, in which case the order of execution of these sub-expressions must be maintained. Thus, the exact semantics of an operation with a return value in the context of a complex expression, particularly the ordering of the evaluation of intermediate results, depends on the semantics of the complex expression, which is implementation-specific. AxSL does not define the semantics of complex expressions containing procedures with return values (or functions with side effects) since axioms contain calls to auxiliary functions, not operations. This issue is not a new one, however. It is a common problem, faced by compilers when generating low-level code for many types of complex expressions, even when procedures with return values are not present. (See, for example, Aho, Sethi, and Ullman [16].) In an implementation of the specification, where the semantics of the complex expression in question is known, the semantics of the operation can be understood by introducing a temporary variable and using the procedure call rule, as described above.

6 Sufficient-Completeness

One of the advantages of algebraic axioms is that a developer can use them to prove whether a specification is consistent and sufficiently complete, and to do other proofs by induction. Testing the internal consistency of a class merely consists of proving (or disproving) that the theory generated by the class's equational axioms and the rules of inference of first-order predicate calculus with equality do not include
TRUE = FALSE

The usual mathematical definitions of completeness are not, however, appropriate for abstract data types or classes [11]. Sufficient-completeness was first defined by Guttag and Horning [6], and can be proved or disproved for any class that includes a basic modifers declaration (or a generated by assertion in Larch). Informally, a set of class axioms are sufficiently complete if one can use them to eliminate non-basic modifier operations from any variable-free expression.

More formally, to prove that a theory for abstract type (or sort) $S$ generated by functions $f_1, \ldots, f_n$ is sufficient-complete, one must show that:

- All variable-free terms of sort $S$ that contain functions with range $S$ are provably equal to terms whose only functions with range $S$ are $f_1, \ldots, f_n$.
- All variable-free terms that are not of sort $S$ are provably equal to terms that contain no function with range $S$ [11, p. 202].

This definition of sufficient-completeness can be translated into the terminology of object-oriented specifications as: A class $C$ with basic modifying auxiliary functions $a_1, \ldots, a_n$ is sufficiently complete if:

- All variable-free expressions of type $C$ that contain functions with range $C$ are provably equal to expressions whose only functions with range $C$ are $a_1, \ldots, a_n$.
- All variable-free expressions that are not of type $C$ are provably equal to expressions that contain no function with range $C$.

For example, according to the axioms for Stack in Figure 4, the expression

\[
\text{sAfterPop(sAfterPush(sAfterPush(auxMkStack(),16),43))}
\]

(where sAfterPop and sAfterPush are abbreviations for auxStackAfterPop and auxStackAfterPush) can be rewritten as

\[
\text{sAfterPush(auxMkStack(),16)}
\]

which is an expression containing only the basic modifiers auxMkStack and sAfterPush. The expression

\[
\text{auxSize(sAfterPush(sAfterPush(mk-Stack(),16),43))}
\]
can be rewritten as 2, which is an expression that contains no Stack operations. By the
rules above, the Larch and AxSL specifications for the Stack class in Figures 1 - 4 are
not sufficiently complete because their axioms do not define the effect of calling pop on
a newly created stack. Figure 4, for example, does not show how to rewrite
sAfterPop(auxMkStack()) and auxPop(auxMkStack()). In this case the
"incompleteness" is intentional, as these functions (and corresponding operations) are
undefined on empty stacks. In fact, the pre-conditions on the pop operation in
Figures 3 and 4 are a substitute for the missing axioms. Once we include consideration
of pre-conditions, we see that the Larch Shared Language specification in Figure 1,
which does not have a pre-condition on POP or TOP, is not by itself sufficiently
complete, whereas the Larch/C++ specification in Figure 2 and the AxSL specifications
in Figures 3 and 4 are.

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 those axioms in the
first place. By the rules of sufficient-completeness, each auxiliary function (or functional
operation) that is not a basic modifier should have axioms that relate it to each of the
basic modifiers. An exception is that some observer functions may be defined in terms
of one or more other observers in a single equation. In Figure 4, the rules of
sufficient-completeness identify the following axioms as necessary:

- auxStackAfterPop should have two axioms, one relating the function
to auxMkStack and one relating it to auxStackAfterPush.
- The observer functions auxPop, auxSize, and auxEmpty should each
  have two axioms, one relating the function to auxMkStack and one
  relating it to auxStackAfterPush, or should have one axiom relating
  the function to other observer functions that have been fully defined.

If axioms are missing, as two of the Stack axioms are, then those cases may correspond
to exceptions or unmet pre-conditions [11, p. 204].

Since sufficient-completeness is defined in AxSL in terms of auxiliary functions, the
developer of a specification must identify which auxiliary functions corresponding to the
basic modifier operations are themselves basic modifier functions (i.e., in the generating
set). For any basic modifier operation with only a single result, determining the basic
modifier function is trivial; it is the single auxiliary function associated with the
operation. For an operation with multiple results and multiple associated auxiliary
functions, the auxiliary function corresponding to the implicit first parameter, SELF,
should be the basic modifier function corresponding to the basic modifier operation.

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. A basic observer function
should have an axiom relating it to each of the basic modifier functions. An extra
observer, on the other hand, may be defined in terms of basic observers. For any basic
observer operation with only one result, the corresponding basic observer function is the single auxiliary function associated with the operation. For a basic observer operation with multiple results, the auxiliary function associated with the return value is usually the appropriate corresponding basic observer function. Sometimes, however, it is one or more of the other auxiliary functions. The only way to determine which auxiliary functions should be listed as basic observer functions is to refer to the definition of basic observers.

It is interesting to note that, since object-oriented operations may have both return values and side effects, some operations may be both modifiers and observers. The pop operation in Figure 4, for example, is not only an extra modifier, but also a basic observer operation; it provides the only way to observe elements in the stack, since the Stack class in this figure does not include a top operation.

We have already noted that operation pre-conditions, such as that on the pop operation in Figure 3 and Figure 4, may obviate the need for particular axioms. Post-conditions, also, may substitute for axiom expressions. For example, an axiom that relates a basic observer function to a basic modifier function may be represented in AxSL as a post-condition expression on the associated basic modifier operation. The post-condition on the push operation in Figure 3, for example, includes the “axiom” relating top to push. Similarly, an axiom that relates an extra observer auxiliary function to a basic observer function may be missing from the AXIOMS section of the specification, but present in the post-condition for the extra observer. The post-condition on empty in Figure 3, for example, contains the “axiom” relating the extra observer empty to the basic observer size. Thus, testing for sufficient-completeness in an AxSL specification must take into account axioms, pre-conditions, and post-conditions. The object invariant, which acts as an implicit pre-condition for all operations other than constructors and as an implicit post-condition for all operations (other than destructors), also plays a role.

7 Other Features of AxSL

Inheritance Semantics

A principle difference between the Larch Shared Language and AxSL is in their handling of inheritance. LSL supports the general concept of trait theory reuse through the assume, import, and include operations, but it does not explicitly support object-oriented inheritance. (Larch interface languages support whatever language-specific reuse concepts are built into their corresponding implementation languages.) Since inheritance is one of the fundamental concepts of the object-oriented paradigm, AxSL provides more direct support for it.

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. The intuitive and functional meanings that have been attached to object-oriented inheritance fall into three distinct categories:

- subtype inheritance (or behavioral inheritance [17]), which allows type-safe dynamic binding of subclasses to superclasses,
- parameterized type inheritance, which supports specialization of homogeneous collection classes such as Lists, Sets, Stacks, and Queues, and
- selective inheritance, which allows a subclass to selectively inherit any subset of properties from the superclass [1].

AxSL supports all three types of inheritance and an informal, unspecified inheritance with the is A Subtype Of, is A Parameterized Subclass Of, is A Selective Subclass Of, and Is A keyword phrases, respectively. An AxSL specification developer can specify the inheritance type explicitly, as in Is A Subtype Of, for example, 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 [1] [2]. 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 in reverse to logically deduce the inheritance relationship between subclass and superclass. AxSL also supports multiple inheritance. A subclass may, for example, have a subtype inheritance relationship with one superclass and a selective inheritance relationship with another.

Inheritance plays a significant role in determining sufficient-completeness. In subtype inheritance relationships, 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 [1] [2].

Multiple Views

Another extension that AxSL adds to Larch is the ability to specify different interfaces to the object through different visibility levels. Unfortunately, existing object-oriented languages have widely differing types of visibility interfaces—external, friend, subclass-visible, and internal. AxSL supports the external, or public, interface (containing operations available to objects of other classes), the subclass-visible interface (containing operations available to subclass objects), and the internal interface (containing operations available only to the object itself). These interfaces are
CLASS Stack[T: TYPE]
HAS 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: empty () → b: Boolean
HAS SUBCLASS-VISIBLE INTERFACE:
HAS OPERATIONS:
OBS: size () → Integer
END Stack[T: TYPE]

Figure 5: An Example of Multiple Interfaces

analogous to the three types of object clients identified by Wirfs-Brock and Wilkerson in [18]: external clients, subclass clients, and self clients. Figure 5 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. AxSL also supports the specification of state information in any interface, when that is appropriate for the application.

In some circumstances, it may be appropriate to have special external interfaces, called friend interfaces, instead of, or in addition to, a general external interface. Each friend interface 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 the specification of friend interfaces in spite of the fact that no existing object-oriented languages support them, because they are a useful concept and because some design methods, such as the one developed by Wirfs-Brock et al. [19], use them.

Invariants

AxSL supports invariants, as in Figure 3 and Figure 4. 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 subclass object invariant must hold for its inherited operations as well as for the new and redefined operations specified in the subclass. 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 of the class is not greater than the upper bound.
8 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 \textit{vice versa}.

Pax produces both a plain text version and a formatted \LaTeX\ source version of
each class it parses. 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. After the developer has
filled in the right-hand sides of the axioms and post-conditions, the plain text version of
Pax's output may be sent through the tool again as an input file to parse and validate
the modified specification.

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. These features remain in the plans for future work.

AxSL was first designed as part of a change management system for maintaining
object-oriented library components (classes) implemented in multiple languages [20] [2].
In this system, AxSL serves as the specification language in which to capture the
language-independent requirements for the reusable components. Synch is a prototype
development environment for creating and maintaining AxSL specifications and
multiple language implementations in a synchronized manner. Synch allows the user to
create links between operation signatures, axioms, pre-conditions, post-conditions, and
invariants in the specification and the corresponding assertions (or even whole code
segments) in each of the implementations. Developers can then use these links to
quickly find code in each language version that implements a given requirement of the
specification. In addition, if any changes are made to the specification, or if
language-independent changes are made to one of the implementations, Synch will use
the links to indicate to the developer which code in other files must be updated as a
result. Synch currently supports basic editing of specifications and implementations,
provides the ability to create, edit, and browse links between specifications and language
versions, and allows developers to call Pax from the specification editing window
(especially important since Pax can automatically generate auxiliary functions and the
left-hand sides of required axioms and post-conditions). Still being developed are the
ability to recognize changes that should be pushed through the system and the ability to provide visual indications of code that is no longer “in synch” with the specification.

9 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. It also supports a number of features that are not universal among object-oriented languages, such as multiple inheritance, multiple interfaces to a class, and shared class attributes. 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 there is an obvious mathematical model for a class. 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. An operation’s post-condition includes expressions that establish the relationship between the operation and its auxiliary functions, while 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 in those cases where a post-condition representation is equivalent to an axiomatic representation, i.e., when the post-condition expression has no side effects. Using post-conditions in this way also reduces the number of extra auxiliary functions that must be introduced into a class specification.

Sufficient-completeness is an important concept used to determine whether a specification contains all the necessary axioms and post-conditions to specify its behavior. The set of axioms for a class is sufficiently complete if it is possible to reduce any variable-free expression containing the class’s auxiliary functions and functional operations to an expression that contains only the class’s basic modifier functions or functions from other classes. Sufficient-completeness can also be used as a guide in developing specifications, since it indicates what axioms and post-conditions are necessary.
References


