Driver's Education and Formal Interface Specifications

Jason O. Hallstrom and David P. Jacobs

Technical Report RSRG-11-05
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

October 2011

Driver's Education and Formal Interface Specifications

July 15, 2011

Imagine yourself walking into a car dealership to buy a new car. Your stomach clenches. There are a lot of reasons to be nervous. Once you find a car you like, you'll need to haggle with the salesman over the price, the credit terms, and any other issues that might come up. After all of that, you'll need to figure out how to drive the darn thing home!

Wait a minute. If you've been driving for awhile, that last worry never entered your mind. Why not? After all, there are more makes and models of cars than you can shake a stick at. (Pun intended.) Each one has its own service manual because they are all different under the hood. Just take a look; one quick peek at the engine of a Mustang and a Prius will convince you. So why don't we worry about how we'll manage driving when we buy a new car or borrow one from a friend? The answer is so obvious the question seems silly: From the driver's perspective, every automatic behaves like an automatic, and every manual behaves like a manual. Every car, regardless of its internal implementation, provides the same interface to its users.

Providing a standard, well-understood interface that makes it easy to drive a car without understanding how it works is valuable. That's why, when we step into a new car, we immediately understand how it works without having to read the manual. It's also useful to the carmakers; they know exactly which functions the car must provide because every car must provide them. They're of course free to implement those functions however they want, but the gas pedal better behave like a gas pedal, and the brake pedal better behave like a brake pedal — or nobody is going to buy their car!

1 Interfaces and Implementations

When we develop object-oriented software, the most basic component of an application is a class. A class consists of two kinds of things. It contains state elements, sometimes referred to as "member variables" or "fields". These are variables associated with the objects created from the class. (The process of creating an object from a class is sometimes referred to as "construction" or "instantiation".) It also implements functions, sometimes referred to as "member functions" or "methods", again associated with the objects created from the

class. These functions are used to interact with individual objects, to access the state elements they contain.

Classes can be incredibly complex in terms of the logic they contain — arguably more complex than some cars. Equally important, some classes are used over and over again across many applications. This is especially true of classes that implement common data structures, such as queues, stacks, lists, and sets. If you consider any one of these data structures, you'll see that it can be implemented in many ways. In fact, a single class library might contain multiple implementations of the same data structure. Consider, for example, a queue. One implementation might be implemented using a static array. Another might be implemented using a dynamic array. Still another might use a linked-list. If each time we needed a queue in one of our programs, we had to review the queue's particular implementation, we wouldn't be very effective programmers. We would be so busy studying the implementations of the various classes we needed to use that we wouldn't be able to write any new code of our own. We'd spend our days reading source code but never writing any!

Of course, if you've ever used a class library, you know that this isn't how the development process works for most programmers. When you find a queue implementation in the library, you don't spend too much time worrying about how to use it. Why not? Again, the answer is so obvious, the question seems silly: From the using programmer's perspective (sometimes called the "client"), every bounded queue behaves like a bounded queue, and every unbounded queue behaves like an unbounded queue. Every queue component, regardless of its implementation, provides roughly the same interface to its users.

When we talk about the interface of a class, we're talking about two types of information. Suppose that you've been given a new class that you're expected to use as part of a programming project. To use the implementation correctly, you will first need to understand the *syntactic* requirements for interacting with instances of the class (i.e., objects). This means that you need to know the names of the functions the class provides, the types of arguments those functions require, and the types of values the functions return, if any. The name of a function, its argument types, and its return type are collectively referred to as the function's "signature". Understanding the signatures defined by the new class is only half the battle. To use the class correctly and impress your friends and neighbors, you must also understand the *semantics* of the functions.

The word "semantics" refers to meaning. When we talk about the semantics of a function, we're talking about what it means to use the function. Again, there are two parts to consider. First we need to understand what we're allowed to do as callers: When can the function be called[1] and what argument values can be passed[2]? These requirements are referred to as "pre-conditions"; they

[1]When we say "when" here, we're not referring to time; we're referring to the state of the object that the function is attached to. This object is sometimes referred to as the function's "target" object. So this question is more accurately phrased as, what values must the target object contain before the function may be called?

[2]Notice that this information is not captured in the function's signature. An argument's type gives you some information, but not enough. Consider, for example, a function that

are requirements that must be satisfied by the caller of a function before it is called. Once we have an understanding of our obligations as a caller, we need to understand the behavior of the function: What will the function do if it is called correctly? The promises that a function makes to its callers are referred to as "post-conditions"; they are guarantees that the function makes about its behavior. With a solid understanding of a function's pre-conditions and post-conditions, we're ready to get to work using the function in our program.

2 Design-by-Contract

This model of programming based on pre-conditions and post-conditions is sometimes referred to as "Design-by-Contract". Consider again your new car purchase. As a self-proclaimed audiophile, you'll want to install a top-of-the-line stereo; you'd like the subwoofer to break glass. When you call the car audio store to arrange for the work to be done, you enter into a contract, even if it's stated only implicitly. The store's representative asserts certain pre-conditions for doing the work: You are required to bring the car to the store, possibly remove your old stereo, and pay a certain fee. In return, the representative guarantees certain post-conditions: Your new stereo will be installed, and your car will be returned to you in perfect working order. If you (the caller) satisfies the pre-conditions, the store (the implementer) will satisfy the post-conditions. If, however, you fail to meet a pre-condition, the store (the implementer) is under no obligation to behave properly. To convince yourself of this, try to underpay the car audio store. You'll be able to read the rest of this on your long walk home!

Design-by-Contract establishes a clear set of responsibilities for clients and implementers of classes. Users understand that they must satisfy the pre-conditions of every function called within their program; the post-conditions can be assumed. Similarly, implementers understand that they may assume each function's pre-conditions; the post-conditions must be ensured. As we will see, this simple set of rules simplifies the development process considerably.

3 Informal Documentation

When you imagined yourself walking into the car dealership to purchase your new car, you didn't worry about how you'd drive it home. The car's interface was well-known to you, even if the implementation under the hood was mysterious. Now imagine yourself at the headquarters of Boeing, signing the contract on your new 787 Dreamliner. Thanks to a very large gift from your very rich uncle,

accepts two arguments, the first an array of characters, and the second an integer. From this information you immediately know that you can't pass a floating-point number as the second argument. But can you pass 11? Perhaps not. What if the integer is used as an index into the array? In this case, the set of allowable integer values that may be passed is limited to the set of allowable array indices. If the array contains 10 slots, indexing the 11th is sure to result in trouble.

you don't have to worry about the payments; they're covered. But you are responsible for flying the Dreamliner home. Feeling nervous? Everyone along your flight path is.

Unlike the car, which provides a well-known interface, the Dreamliner's interface is unfamiliar to most users, and far more complex. To use the aircraft effectively, you're going to need some documentation (and probably a license). For the sake of those on the ground and in the air, the documentation had better be as clear and as concise as possible.

This same type of scenario plays out routinely when you're writing software. When you encounter a class that provides an interface you understand well, you're able to use it immediately. When you encounter an interface that you haven't used before, you need to spend some time understanding the interface — both its syntactic and semantic requirements. The syntactic requirements are usually easy to understand; they're captured by the syntax of the programming language. The semantic requirements, however, are more complex. Remember that for each function, you need to understand what you must do to place a correct call (i.e., the pre-conditions), and what you should expect to happen as a result (i.e., the post-conditions). Without that information, you can't possibly use the implementation effectively. The information needs to be captured in some form of interface documentation.

While it may be a stretch to call Design-by-Contract a universal standard, the basic strategy is widely adopted by commercial library vendors. If you take a look at some of the online documentation available for the class libraries that come with Java, C#, Python, and other popular object-oriented languages, you'll see this reflected in the documentation's structure. Pick a class that implements a common data structure from your favorite class library. (I know, I know; it's hard to pick just one.) The documentation likely begins with an English narrative that explains how client programmers should think about instances of the class. Or stated another way, the narrative tries to capture the *state space* of the class, the set of values the corresponding objects may hold. The remainder of the documentation loosely characterizes the pre-conditions and post-conditions of the methods defined by the class.

Consider Java's `Stack` class as an example. The documentation begins, "The `Stack` class represents a last-in-first-out (LIFO) stack of objects." The remainder of the documentation describes the `Stack` methods. The post-condition of `push()`, for example, explains that `push()` "pushes an item onto the top of [the] stack." The `peek()` function throws an exception "if [the] stack is empty." This can be viewed as a (checked) pre-condition; you ought not call `peek()` on an empty stack.

But let's consider for a moment how useful this documentation is to developers. To recap what we've learned, a `stack` should be understood as a stack, and its `push()` method pushes. Clear? Probably not. Who was this documentation written for anyway? Imagine if your Dreamliner's documentation was written this way. "The control column is used for control, and the doohickey stick is used to doohickey." (So I've exhausted my knowledge of aeronautics. Sue me.) The point is that the informal documentation available for commercial

class libraries is written in such a way that it is most useful to those who already understand how the classes work! Newcomers can use the documentation to get a feeling for how the classes might be used, but must often resort to trial-and-error, placing various calls using various argument values. This approach is inefficient and error prone, as I suspect you may already know from personal experience. For class documentation to be most valuable, it must be written in a manner that is clear and concise; ambiguities must be avoided.

4 Formal Specifications

Where do we go from here? One approach is to try to be very careful in describing each class, to choose just the right words that capture its state space and function behaviors. As it turns out, this is surprisingly hard to do. To convince yourself, try to explain to a friend how a `stack` interface works, assuming your friend has never heard of a stack before. In particular, you're not allowed to describe any of the ideas in terms of themselves (e.g., "a `stack` is a stack, and `push()` pushes"). If you take this exercise seriously, your description is likely to suffer from two problems. First, it will be verbose. Even a simple concept like a stack is difficult to explain concisely. Second, despite your best efforts to craft a careful narrative, the description is likely to contain some points of ambiguity. But don't feel too badly about the outcome. After all, natural languages weren't designed for precision. They evolved over time to enable the expression of a very broad set of ideas, and the resulting sentences are subject to interpretation. This makes English, for example, a bad choice for documenting the semantics of something that has very precise usage requirements and associated behaviors.

So which languages were designed with precision and concision in mind? As it turns out, there's really just the one. (Wait for it...) The language of mathematics. Formal logics, in particular, were developed to enable the precise expression of mathematical properties and rigorous reasoning about the facts those properties imply[3]. Each logic involves a limited vocabulary and is totally devoid of ambiguity. And despite the anxiety you may feel about adopting formal logic as a documentation standard, we'll quickly see that many interesting program properties can be expressed using a small set of mathematical statements. In most cases, it's no harder than programming; I promise.

Let's again consider the `Stack` interface. To begin, we need to map the concept of a stack into mathematical terms. More precisely, we need to adopt a *mathematical model* of the stack that captures its state space in a way that is descriptive to developers and supports the documentation of pre-conditions and post-conditions. A mathematical *string* turns out to be a good choice. Before we begin documenting the interface in terms of this model, it's useful to review this mathematical concept.

[3] A "formal logic" is a mathematical system that provides formal rules for writing mathematical statements and deriving new statements that are guaranteed to be true. There are many different logics, but they all share this same approach.

In mathematics, a string is an ordered sequence of elements. Syntactically, a string is represented as a comma-separated list contained within angle brackets. All of the elements within a given string must be of the same type. Let's consider a few examples:

- $< 4, 6 >$
- $< 7, 1, -2 >$
- $< 4.5, 3.2 >$
- $< 0.1 >$
- $< c, a, b, b >$
- $< e >$
- $<>$

The first two examples denote strings of integers containing 2 and 3 elements, respectively. The third and fourth denote strings of real numbers containing 2 elements and 1 element, respectively. The third and fourth denote strings of characters containing 4 elements and 1 element, respectively. Finally, the last example denotes a string containing zero elements. This one is special; we refer to the value as the "empty string". Want some more examples? (I didn't think so.)

Like any mathematical type, we need some operations to make the type interesting. (Just listing example after example could get boring, as you may have noticed.) There are only two operations defined on mathematical strings, *length* and *concatenation*. The length operator is *unary*; it takes one string as argument and produces an integer as a result. It's represented using vertical bars placed on either side of the argument. So again considering the examples above, we have the following equalities:

- $| < 4, 6 > | = 2$
- $| < 7, 1, -2 > | = 3$
- $| < 4.5, 3.2 > | = 2$
- $| < 0.1 > | = 1$
- $| < c, a, b, b > | = 4$
- $| < e > | = 1$
- $| <> | = 0$

Concatenation, the second operator, is *binary*; it takes two strings as arguments and produces a third. It's used to glue two strings together. The operator is represented using an asterisk between the arguments. If we wanted to apply some glue to the example strings above, we'd get the following:

- $< 4, 6 > * < 7, 1, -2 > = < 4, 6, 7, 1, -2 >$
- $< 4.5, 3.2 > * < 0.1 > = < 4.5, 3.2, 0.1 >$

- $< c, a, b, b > * < e >=< c, a, b, b, e >$
- $<> * < c, a, b, b > * <>=< c, a, b, b >$

Notice that the concatenation operator can only be applied to strings that are of the same type. You couldn't, for example, glue a string of integers to a string of characters. Also notice that the empty string has no effect when you glue it to another string. If you wanted to impress your friends and family, you could explain this property mathematically: The empty string is the *identity element* for concatenation.

That's it. There's nothing else you can do with strings, so stop asking. If you don't want to already, at some point you're going to feel an overwhelming urge to use array notation to access an element within some string. You can't. Not ever. Stop thinking about it. You get length and concatenation — that's it.

At this point, we're ready to begin documenting the `Stack` interface formally. The first step is to identify the interface's model and the initial value that instances of the interface must hold when they are first created. Our *interface specification* begins as follows[4]:

```
public interface Stack<E> {
   /*!
    ** modeled_by: string of E
    ** initial_value: <>
   !*/
```

Listing 1: `Stack` Interface Specification (partial; 1 of 5)

The `modeled_by` clause specifies the interface model, a mathematical string of `E`, where `E` is a generic type parameter; it could be anything[5]. The `initial_value` clause specifies the value that an instance of the interface will hold at the point of creation. In this case, the clause states that all `stack`s are initially empty.

Now let's consider the functions provided by the interface — `push()`, `pop()`, `length()`, and `clear()`. By now you should object to me writing things like, "the functions have their usual meaning." That's completely useless if you've never seen a stack before! Instead, we must define the semantics of these functions in terms of pre-conditions and post-conditions using the interface model. The basic intuition is that we will designate the left side of the mathematical string as the "top" of the stack, and the right side of the string as the "bottom". This makes it easy to specify each function precisely.

[4] We use Java syntax for our interfaces, but the specification ideas are not Java-specific; almost any programming language would work equally well.

[5] The idea here is that the generic parameter `E` will eventually be replaced with an actual programming type, say `Integer`. This type will also be specified mathematically. When its mathematical type is substituted for `E` in the `modeled_by` clause, a *fully instantiated* model will result, namely, a string of integers.

Let's begin with `push()`, which accepts a single argument of type `E`. This interface isn't designed for a size-limited stack, so `push()` may be called at any time; it doesn't define a real pre-condition. It does, however, provide behavioral guarantees that need to be captured in its post-condition. So what does this function do? That's easy; `push()` places its argument on the left side of the mathematical string. More formally, when `push()` returns, the `stack`'s value will be equal to the string containing the argument, concatenated with the `stack`'s value before `push()` was called. The value of the `stack` immediately before `push()` was called is referred to as its "pre-conditional value". The value of the `stack` immediately after `push()` returns is referred to as its "post-conditional value". Let's take a look at the specification:

```
void push(E e);
/*!
 ** requires:
 **    true
 ** ensures:
 **    self = <e> * #self and
 **    e = #e
!*/
```

Listing 2: `Stack` Interface Specification (partial; 2 of 5)

The `requires` and `ensures` clauses specify the function's pre-condition and post-condition, respectively. Each is represented by a mathematical *assertion*, a boolean expression defined in terms of the model. The pre-condition assertion must evaluate to *true* before the function is called. (This is the caller's responsibility.) The post-condition assertion must evaluate to *true* after the function returns. (This is the implementer's responsibility.) Since `push()` may be called at any time, the pre-condition is specified simply as `true`, indicating that the pre-condition is always satisfied; there is nothing special the caller needs to do[6]. The post-condition is more complex; it is a conjunctive statement involving two clauses[7]. The first clause states that the post-conditional value of the `stack`, denoted by `self`, will be equal to the string containing the argument `e`, concatenated with the pre-conditional value of the `stack`, denoted by `#self`. The second clause states that the post-conditional value of the argument `e` will be equal to its pre-conditional value, denoted by `#e`; that is, `e` will be preserved by the function.

This specification demonstrates two important syntactic mechanisms. First notice that the `self` keyword is similar to the `this` keyword used in popular languages like Java and C++. Like `this`, `self` refers to the "current" instance.

[6] In practice, a pre-condition of `true` is typically omitted; the `requires` clause is simply not included.

[7] A "conjunctive" statement is an assertion with multiple clauses, separated by logical "and"s. Each constituent clause must evaluate to *true* for the statement as a whole to evaluate to *true*.

The main difference is that `self` always refers to the mathematical value of the current instance. Second, this specification introduces the # notation used to refer to an object's pre-conditional value. We will use these notations extensively in our specifications.

Now let's consider the specification of `pop()`, which has a non-trivial pre-condition and returns a value:

```
E pop();
/*!
 ** requires:
 **    |self| > 0
 ** ensures:
 **    #self = <pop()> * self
!*/
```

Listing 3: `Stack` Interface Specification (partial; 3 of 5)

The pre-condition states that `pop()` may only be called if the length of the string is greater than zero; you can't call `pop()` on an empty `stack`. Alternatively, this could have been written as `self != <>`; the semantics are identical. Notice that in both cases, `self` refers to the pre-conditional value of the `stack`. After all, the pre-condition imposes requirements on callers, who only have control over pre-conditional values. It would be meaningless to ask a caller to satisfy state conditions defined in terms of values that occur upon completion of the call! For this reason, the # notation is never used in a pre-condition; the pre-condition *always* refers to pre-conditional values.

In the post-condition, `pop()` denotes the value returned by the function. (It does not represent a function call.) Hence, the post-condition states that the string containing the element returned by the function, concatenated with the post-conditional value of the `stack` will be equal to the `stack`'s pre-conditional value. It might take a minute or two to wrap your brain around this statement, but after a bit of thought, you'll see that it captures the desired behavior precisely.

All that's left are the easy ones. Let's take a look at the specification for `length()`:

```
int length();
/*!
 ** requires:
 **    true
 ** ensures:
 **    length() = |self| and
 **    self = #self
!*/
```

Listing 4: `Stack` Interface Specification (partial; 4 of 5)

The pre-condition states that `length()` may be called at any time. The post-condition consists of two clauses. The first states that the value returned by `length()` will be equal to the length of the post-conditional `stack`. The second clause states that the `stack` will be preserved. Taken together, the post-condition states that `length()` will return the number of elements contained within the `stack` and won't modify the `stack` in any way.

The types of preservation constraints that we've seen in the specifications of `push()` and `length()` are very common. We often want to state that a particular argument, say `arg`, is preserved by a given function. To avoid cluttering our specifications with clauses of the form `(arg = #arg)`, we'll introduce a short-hand notation. Specifically, we'll introduce an optional `preserves` clause in our specifications. The clause will consist of a comma-separated list of variables that are preserved by the corresponding function. So rather than writing `(arg = #arg)`, we'll simply write, `preserves: arg`. It's important to remember, however, that this isn't a new type of assertion. It's just a short-hand for expressing a type of assertion that occurs often.

Finally, we have the specification of `clear()`:

```
void clear();
/*!
 ** requires:
 **    true
 ** ensures:
 **    self = <>
!*/
```

Listing 5: `Stack` Interface Specification (partial; 5 of 5)

The pre-condition states that `clear()` may be called at any time. The post-condition states that `clear()` resets the `stack` to its initial value, the empty string.

Putting all of these specification fragments together and leveraging the short-hands we've introduced results in the complete specification of `Stack`, shown in Listing 6.

There you have it. You've been introduced to the world of formal specifications and documented your first interface using mathematical logic. You made it! I suspect that it didn't even hurt that much. In fact, with very little practice, you'll quickly be able to read formal specifications that capture a broad set of interesting program behaviors. With a little more practice, you'll be able to write them yourself. With even more practice, you'll be able to reason rigorously about the programs you write and maintain based on the specifications of the interfaces you use. The end result will be a dramatic improvement in your ability to develop high-quality software. (You might even get a job.) We have exciting work ahead, so let's keep going.

```
public interface Stack<E> {
   /*!
    ** modeled_by: string of E
    ** initial_value: <>
   !*/

   void push(E e);
   /*!
    ** preserves: e
    ** ensures: self = <e> * #self
   !*/

   E pop();
   /*!
    ** requires: |self| > 0
    ** ensures: #self = <pop()> * self
   !*/

   int length();
   /*!
    ** preserves: self
    ** ensures: length() = |self|
   !*/

   void clear();
   /*!
    ** ensures: self = <>
   !*/
}
```

Listing 6: `Stack` Interface Specification (complete)

5 Specification Examples

Before we move on to the next topic, it's useful to spend a bit of time looking at some more examples. We'll focus on common data structures and specify the corresponding interfaces formally. We'll need to introduce a few more mathematical models and a bit of additional syntax — but I know you're up to the challenge.

5.1 The `Queue` Interface

We'll now develop a formal specification for a generic interface `Queue`. If you liked reading about the formal specification for `Stack`, you will like this section too, because queues have much in common with stacks. As you probably remember, a queue has a *front* and a *rear*. It operates like the line at a movie theater box office. The `enqueue()` operation adds a new element to the rear, and the `dequeue()` operation removes an element from the front. Other than that, there is not much difference between stacks and queues. So the first thing we need to do is decide what our mathematical model is. Since strings worked so well in our last example, we will use the same mathematical model:

```
public interface Queue<E> {
   /*!
    ** modeled_by: string of E
    ** initial_value: <>
   !*/
```

Note also that, like `Stack`, the initial value of a `Queue` is represented by the empty string. The only big decision we now need to make is which end of the string will be the front, and which end will be the rear. It doesn't matter, as long as we are consistent. We flip a coin and determine that the *left* side of the string corresponds to the front, and the *right* side of the string corresponds to the rear.

So here is the formal specification for method `enqueue()`.

```
   void enqueue(E e);
   /*!
    ** requires:
    **    true
    ** preserves: e
    ** ensures: self = #self * <e>
   !*/
```

Since we are modeling an *unbounded* queue, `enqueue()` can always be called. Said another way, its precondition is true. As noted earlier, a true precondition is sometimes omitted. The interesting part of the specification is the postcondition which says that `self`, the string representing the queue's new state, will equal the string representing the queue's old state, `#self`, concatenated with `<e>`, on the *right*. This describes the effect of `enqueue()`.

Next, let's look at method `dequeue()`:

```
E dequeue();
/*!
 ** requires: |self| > 0
 ** ensures: #self = <dequeue()> * self
!*/
```

Like the `pop()` operation for `Stack`, the `dequeue()` operation requires that the `Queue` be nonempty, in order to remove an element. Thus the precondition states that the length of `self` is greater than zero. The postcondition says that the string `<dequeue()>` of length one, concatenated with `self`, equals `#self`. That is, the new queue is formed by removing one element from the old queue.

There are two more operations, `length()` and `clear()`. They are identical to the their analogs for `Stack`, so the same comments apply. Our entire formal specification for `Queue` is shown in Listing 7.

```
public interface Queue<E> {
   /*!
    ** modeled_by: string of E
    ** initial_value: <>
   !*/

   void enqueue(E e);
   /*!
    ** preserves: e
    ** ensures: self = #self * <e>
   !*/

   E dequeue();
   /*!
    ** requires: |self| > 0
    ** ensures: #self = <dequeue()> * self
   !*/

   int length();
   /*!
    ** preserves: self
    ** ensures: length() = |self|
   !*/

   void clear();
   /*!
    ** ensures: self = <>
   !*/
}
```

Listing 7: `Queue` Interface Specification

```
public interface Sequence<E> {
   /*!
    ** modeled_by: string of E
    ** initial_value: <>
   !*/
```

Listing 8: `Sequence` Interface Specification (partial; 1 of 5)

5.2 The `Sequence` Interface

In our next example of a formal specification, we consider a generic interface called `Sequence`. This is intended to represent a finite list of elements, where an element may be added or removed at *any* position. The position of the elements in the list are numbered $0, \ldots, n-1$, where n is the length of the sequence. When instantiated, the Java classes `ArrayList` and `Vector` produce sequences. As you might guess, we will model our formal specification as a `string of E`, whose initial value is `<>`, shown in Listing 8.

That said, let us discuss method `add()` and its formal specification, shown in Listing 9.

```
   void add(E e, int pos);
   /*!
    ** requires: 0 <= pos <= |self|
    ** preserves: e, pos
    ** ensures:
    **    there exists l, r : string of E
    **    s.t.
    **    #self = l * r and
    **    |l| = pos and
    **    self = l * <e> * r
   !*/
```

Listing 9: `Sequence` Interface Specification (partial; 2 of 5)

First note that `add()` that takes two parameters, a element `e` of type `E`, and an integer `pos`, which will be the position of `e` *after* the `add()` operation. If there are currently n elements in the sequence, then after the `add()` operation there will be $n + 1$, and so the new element must have a position in the range $0, \ldots, n$. (For example, if you have three people standing in a line, and a new person buts in, the new person will end up in position zero, one, two, or three.) This explains the method's precondition. Note for example, when `pos` is zero, the new element will be inserted at the beginning. When `pos` equals `|self|`, the new element will get added at the end of the sequence.

Now we'll explain the postcondition for `add()`. First, looking back at the postcondition of `enqueue()`, we wrote `self = #self * <e>` which implied that the old string `#self` was preserved as a *prefix* in the new string `self`.

So how can we capture that same idea with operation `add()`? Simply that the substring to the *left* of the inserted element, and the substring to the *right* of the inserted element are unchanged. Since we do not have a name for these substrings, we call them `l` and `r`. Note that the left substring `l` will have length `pos`.

We call the logical construct `there exists` an *existential quantifier*[8]. Note that existential quantifiers are used extensively. For example, how could you say an integer $n > 1$ is not prime? Well, you might say *there exists* a factor k such that $1 < k < n$ and $k|n$.

So the postcondition says something like this: *there are* two strings `l` and `r`, *such that*

1. `#self = l * r`
2. `|l| = pos`
3. `self = l * <e> * r`

Note that Clause 1 and clause 3 together guarantee that the strings to the left and right of the newly inserted element `e` are unchanged. Clause 2 guarantees that `e` is placed in the correct position. Finally, observe that if `l` is the empty string, `e` will be placed at position zero, and if `r` is the empty string, `e` will be placed at the far right.

Next, we'll examine method `remove()`, and its specification, in the `Sequence` interface, shown in Listing 10. Method `remove()` may be thought of as the analog of `pop()` or `dequeue()`.

```
E remove(int pos);
/*!
 ** requires: 0 <= pos < |self|
 ** preserves: pos
 ** ensures:
 **    there exists x : E; l, r : string of E
 **    s.t.
 **    #self = l * <x> * r and
 **    |l| = pos and
 **    self = l * r and
 **    remove() = x
!*/
```

Listing 10: `Sequence` Interface Specification (partial; 3 of 5)

This method has one parameter, `pos`, which specifies the position of the element that is to be removed. Note the difference between the precondition for `add()` and the precondition for `remove()`. If there are n elements in the sequence, the position p of the element to be removed must satisfy $0 \leq p \leq n - 1$, or

[8]The existential quantifier is usually written as $\exists$, but keyboards lack this symbol. The universal quantifier is written $\forall$, but we shall not have occasion to use it in this tutorial.

$0 \leq p < n$. Finally, note that this precondition *forces* the `Sequence` object to be nonempty, because were n zero, no integer p satisfies $0 \leq p \leq n - 1$.

The postcondition of `remove()` is similar in spirit to that of `add()`. There are two substrings `l` and `r` which represent the list to the left and right of the element `x` to be removed. They will satisfy

1. `#self = l * <x> * r`
2. `|l| = pos`
3. `self = l * r`
4. remove() = x

Clause 1 and Clause 3 collectively assure us that the elements to the left and right of the removed element are unchanged. Clause 2 assures us that the correct element was removed. Clause 4 assures us that the returned element was the one removed.

Method `elementAt()` and its formal specification are shown below. Unlike `add()` and `remove()`, this method preserves `self`. The method has one parameter, `pos`. Its purpose is to return the value `x` at position `pos`. Given the previous two methods, it should be easy to understand this formal specification.

```
E elementAt(int pos);
/*!
** requires: 0 <= pos < |self|
** preserves: self, pos
** ensures:
**    there exists x : E; l, r : string of E
**    s.t.
**    #self = l * <x> * r and
**    |l| = pos and
**    elementAt() = x
```

Listing 11: `Sequence` Interface Specification (partial; 4 of 5)

The two remaining methods, `length()` and `clear()` are trivial, and shown below.

```
int length();
/*!
 ** preserves: self
 ** ensures: length() = |self|
!*/

void clear();
/*!
 ** ensures: self = <>
!*/
```

Listing 12: `Sequence` Interface Specification (partial; 5 of 5)

5.3 The `Set` Interface

We now wish to develop an interface called `Set` which is meant to represent finite sets. Before doing so, let us review some concepts about sets. A set may be thought of as a collection. In mathematics, we often use curly braces to denote sets. Here are some examples of sets:

- $\{4, 6\}$
- $\{10\}$
- $\{7, 1, -2\}$
- $\{c, a, b\}$
- $\emptyset$

Of course, we can have sets of integers or sets of anything. The symbol $\emptyset$ denotes the *empty set*, the special set containing no elements. It is sometimes denoted $\{\}$.

If A is a set, we write $x \in A$ to mean that x *is a member of* A. Of the following four, only the first two are true.

- $4 \in \{4, 6\}$
- $6 \in \{6\}$
- $8 \in \{4, 6\}$ (not true)
- $0 \in \emptyset$ (not true)

It is important to understand two properties about a set: First, the elements in a set *do not have multiplicity*. So it makes no sense for us to write $\{4, 6, 6\}$ when talking about a set. Second , elements in a set are *not ordered*. Thus, the the sets $\{4, 6\}$ and $\{6, 4\}$ are equal. If A and B are two sets, we write $A = B$ to mean the sets are equal [9]. The following are true:

- $\{4, 6\} = \{6, 4\}$
- $\{4, 6\} = \{4, 6\}$

The cardinality of a set is the number of elements, and written using vertical bars:

- $|\{4, 6\}| = 2$
- $|\{10\}| = 1$
- $|\{7, 1, -2\}| = 3$
- $|\{c, a, b\}| = 3$
- $|\emptyset| = 0$

[9] In set theory, one defines $A \subseteq B$ if $x \in B$ whenever $x \in A$. Then $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

An important binary operator is *union*, denoted $\cup$. If A and B are sets, $A \cup B$ is the set of elements that are in A or B or possibly both [10]. The following equations are all true:

- $\{4,6\} \cup \{8\} = \{4,6,8\}$
- $\{8\} \cup \{4,6\} = \{4,6,8\}$
- $\{4,6\} \cup \{6,8\} = \{4,6,8\}$
- $\{4,6\} \cup \{4,6\} = \{4,6\}$
- $\{4,6\} \cup \emptyset = \{4,6\}$
- $\emptyset \cup \{4,6\} = \{4,6\}$
- $\emptyset \cup \emptyset = \emptyset$

We will need to define one more binary operator, *set difference*. If A and B are sets, $A - B$ is the set of elements in A but not in B. The following are true:

- $\{4,6,8\} - \{8\} = \{4,6\}$
- $\{4,6,8\} - \{2,3,8\} = \{4,6\}$
- $\{6,8\} - \{4,6,8\} = \emptyset$
- $\{4,6,8\} - \emptyset = \{4,6,8\}$
- $\emptyset - \{4,6\} = \emptyset$.

There are other set operations we could discuss, for example *intersection*. However, the only set operations we will use in our are interface `Set` are *equality*, *membership*, *cardinality*, *union*, and *difference*. These will be denoted in our functional specification as `=` , `in`, `||`, `union`, `-` .

We are now ready to discuss interface `Set`. The first question is what mathematical model should we use? It should be no surprise that we will use *mathematical sets* as our model.

```
public interface Set<E> {
   /*!
    ** modeled_by: finite set of E
    ** initial_value: {}
   !*/
```

Listing 13: `Set` Interface Specification (partial; 1 of 5)

The initial value is the empty set.

Now consider method `add()` and its formal specification, shown in Listing 14. The intent of method `add()` is place a new element `e`, passed as the parameter into the set. The precondition insists that `e` not already be a member of `self`. This is done to make implementation easier, since elements in sets can not occur with multiplicity. The postcondition, `self = #self union {e}`

[10] The union operator is *associative*, that is, $(A \cup B) \cup C = A \cup (B \cup C)$. It is also *commutative*: $A \cup B = B \cup A$.

```
void add(E e);
/*!
 ** requires: e not in self
 ** preserves: e
 ** ensures: self = #self union {e}
!*/
```

Listing 14: `Set` Interface Specification (partial; 2 of 5)

says that `self` has all the old elements, together with `e`. It is a common mistake to write: `self = #self union e` . This is incorrect since the union operator is applied to sets. Again, we stress that if we removed the precondition, the postcondition would still be mathematically correct.

Next, consider method `remove`, which removes an element `e`, passed as a parameter, from the set. Again, for purposes of implementation, the precondition insists that `e` not be in the set. The postcondition can be interpreted to mean that `e` is not in `self`, but all other elements remain.

```
void remove(E e);
/*!
 ** requires: e in self
 ** preserves: e
 ** ensures: self = #self - {e}
!*/
```

Listing 15: `Set` Interface Specification (partial; 3 of 5)

Next, we consider method `removeAny()`, whose purpose is to remove an *arbitrary* element from the set, and return it. It is obvious that `self` be nonempty, thus the precondition. The postcondition must use an existential quantifier to name the element `x` to be removed. The three clauses in the postcondition assert that `x` was in the set, `x` was removed from the set, and the element returned was the element removed.

```
E removeAny();
 /*!
 ** requires: self != {}
 ** ensures:
 **    there exists x : E
 **    s.t.
 **    x in #self and
 **    self = #self - {x} and
 **    removeAny() = x
!*/
```

Listing 16: `Set` Interface Specification (partial; 4 of 5)

The remaing three methods are straightforward, and are shown below. Method `isIn()` returns a boolean indicating if parameter `e` is a member of the set. Method `size()` returns set's cardinality. Method `clear()` empties the set.

```
public boolean isIn(E e);
  /*!
   ** preserves: self, e
   ** ensures: isIn() = (e in self)
  !*/

 public int size();
 /*!
  ** preserves: self
  ** ensures: size() = |self|
 !*/

 public void clear();
 /*!
  ** ensures: self = {}
 !*/
```

Listing 17: `Set` Interface Specification (partial; 5 of 5)

6 Formal Verification

While software testing can reveal errors, software testing cannot *prove* that the software is correct. The reason is that there can be infinitely many inputs to any program.

What we can do however, is *reason* about a program. But understand that the simplest of programs can have behavior that is very complex. Consider the program which reads a single positive integer n. It then repeats the following if statement while $n \neq 1$: If n is even, it executes $n = n/2$, else $n = 3n + 1$. This program is roughly 7 lines of code, yet it is presently not known if this program halts for all positive n. [11]

In this section, we will show how some simple programs *can* be verified. To *verify*, means to mathematically prove the program's postcondition will hold, if the precondition holds. The example programs will not contain loops or if statements. Our proofs will use formal specifications, as well as properties of their mathematical models.

Now consider the following simple program which has a single argument `s` which is a `Stack`. We will demonstrate how this program can be formally verified.

```
public static void doNothing(Stack s) {
   /*!
    ** requires: |s| > 0
    ** ensures :  s = #s
   !*/

    Object x = s.pop();
    s.push(x);
}
```

Listing 18: Simple program

A main tool in program verification is to first identify and number the *states* of the program. There are three states in the program which we identify with comments:

```
    // state 0
    Object x = s.pop();
    //  state 1
    s.push(x);
    //  state 2
```

The next step is to construct a *reasoning table*, such as the one in Table 1. In general, the table has one row for each state. The goal is to prove the postcondition `s = #s`, starting with the precondition `|s| > 0`. For each argument or variable v in the program, we let v_i denote its value at state i. For example,

[11] The Collatz conjecture says that every positive n will eventually reach one.

in the above program, x_0, x_1, and x_2 denote the values of x at states 0, 1, and 2. Note that x_0 is undefined since the declaration of x has not occurred until state 1. Similarly, the states of the `Stack` s are denoted with s_0, s_1, and s_2.

State #	Facts	Obligations
`0`	$\|s_0\| > 0$	$\|s_0\| > 0$
`1`	$s_0 =< x_1 > *s_1$	`true`
`2`	$x_2 = x1$ $s_2 =< x_2 > *s_1$	$s_2 =< x_2 > *s_1$ $=< x_1 > *s_1$ $= s_0$

Table 1: Reasoning Table for Listing 18

In our reasoning table, the column labeled **Facts** gives information that we know to be true. The column labeled **Obligations** contain preconditions that must be met prior to the next state. Obligations must be proved using previously established facts. The proof starts by placing the program's precondition in the **Facts** cell for state 0, in this case $|s_0| > 0$. The table is filled, row-by-row, left-to-right. The goal is to derive the postcondition that $s = \#s$. Note this is exactly $s_2 = s_0$. In deriving facts, we can also make use of the *frame property* which says that if a variable is not referenced, its value is unchanged.

To perform the `pop()` operation in the first line of the program, its precondition must be met, namely that the stack `s` is nonempty. Establishing this obligation can be be done by merely copying the fact $|s_0| > 0$ to the obligation cell.

Since the precondition was met, the postcondition of `pop()` is guaranteed: Thus, in the **Facts** cell for state 1, we write $s_0 =< x_1 > *s_1$, making use of the postcondition for `pop()` in the `Stack` interface. Since `push()` requires no postcondition, so we simply write `true` in the obligation box of row 1.

In state 2, we produce the facts: $x_2 = x_1$, which follows from the fact that `push()` preserves its argument. We also produce $s_2 =< x_2 > *s_1$, which is the postcondition of `push()`.

In the last obligation box, we use any of the preceeding facts to prove the program's postcondition $s_2 = s_2$. Note that all three previously derived facts are used. It is just a matter of manipulating strings.

When filling in a cell in the **Facts** column it is not necessary to include every possible fact, only those which will be used in the proof. However, it does not hurt to include all the facts, and then later remove the unused ones.

Listing 19 gives another, slightly more interesting program that we will use to illustrate verifcation. The following program also uses a `Stack`. Note there are six states which we have already numbered. We will prove the program

```
public static void mystery(Stack s) {
   /*!
   ** requires:  |s| > 1
   ** ensures :  there exists x,y : Object, r : string of Object
               such that    #s = <x> * <y> * r   and
                             s = <y> * <x> * r
   !*/
   //  state 0
   Object a, b;
   //  state 1
   a = s.pop();
   //  state 2
   b = s.pop();
   //  state 3
   s.push(a);
   //  state 4
   s.push(b);
   //  state 5
}
```

Listing 19: Another simple program

is correct using a reasoning table. The program's precondition says that the `Stack` object must have *more* than two elements. The program's postcondition can be interpreted to say that the top two elements have been interchanged.

We next construct the reasoning table, shown in Table 2, as follows. In row zero, the program's precondition was placed in the facts column. No obligation was necessary for the declarations of variables `a` and `b`.

In Row 1 of Table 2 the variables have NULL values, and the frame property applies to the parameter `s`. Since the next operation is a `pop()`, we are obligated to prove that the size of `s` is greater than zero. In fact, it is greater than one.

In Row 2, the postcondition of `pop()` is used to ascertain the fact

$$s_1 =< a_2 > *s_2.$$

In the obligation cell, since the next operation is a `pop()`, we prove that the size of s_2 is positive, which follows from the fact that $|s_2| = |s_1| - 1$.

In Row 3 of the table, we ascertain two facts. One follows from the postcondition of `pop()`, the other follows from the frame property. There is no obligation, since the next operation is a `push()`.

In Row 4 of the table, there are three facts. The first follows from the postcondition of `push()`. The second fact, namely $a_4 = a_3$, follows because the `push()` operation preserves its argument. The third fact, $b_4 = b_3$, follows from the frame property. There is no obligation, since the next operation is a `push()`.

State #	Facts	Obligations
`0`	$\|s_0\| > 1$	`true`
`1`	$a_1 = b_1 = \mathrm{N}ULL$ $s_1 = s_0$	$\|s_1\| > 0$ since $\|s_1\| = \|s_0\| > 1$
`2`	$s_1 =< a_2 > *s_2$	$\|s_2\| = \|s_1\| - 1 = \|s_0\| - 1 > 0$
`3`	$s_2 =< b_3 > *s_3$ $a_3 = a_2$	`true`
`4`	$s_4 =< a_4 > *s_3$ $a_4 = a_3$ $b_4 = b_3$	`true`
`5`	$s_5 =< b_5 > *s_4$ $b_5 = b_4$ $a_5 = a_4$	

Table 2: Reasoning Table for Listing 19

Finally, in Row 5 of the table, the first fact follows from the postcondition of `push()`, the second fact follows because the `push()` operation preserves its argument, and the third fact follows from the frame property.

We now ready to complete the last step of the proof, namely to establish the postcondition: there exists objects x and y and a string r such that:

$$\begin{aligned} \#s &= < x > * < y > *r \\ s &= < y > * < x > *r \end{aligned}$$

Using facts from Table 2, we have

$$\begin{aligned} s_5 &= < b_5 > *s_4 \\ &= < b_5 > * < a_4 > *s_3 \\ &= < b_4 > * < a_4 > *s_3 \end{aligned}$$

Similarly, we derive:

$$\begin{aligned} s_0 = s_1 &= < a_2 > *s_2 \\ &= < a_2 > * < b_3 > *s_3 \\ &= < a_3 > * < b_3 > *s_3 \\ &= < a_4 > * < b_3 > *s_3 \\ &= < a_4 > * < b_4 > *s_3 \end{aligned}$$

We now have:

$$s_0 = < a_4 > * < b_4 > * s_3 \quad (1)$$
$$s_5 = < b_4 > * < a_4 > * s_3 \quad (2)$$

By letting $x = a_4$, $y = b_4$, and $r = s_3$, we see that Equation 1 and Equation 2 imply the program's postcondition.

7 Conclusions

Formal specifications are made with a precise underlying mathematical model. We have seen that formal specifications help us both as an implementer and user of an interface. We have also seen that formal specifications can help us formally verify the correctness of certain programs. Formal specifications also permit verification to be *automated*. For example, some researchers have built more general purpose program verifiers, that can take, as *input* a program P, its precondition and postcondition, and verify the correctness. [12]

[12] There are theoretical limits to how far program verification can be taken. A classic result in computability theory says that the Halting Problem is undecidable, thus implying that it is not possible to create a program verifier general enough for *all* programs P.