Chapter 3, Introductory Pargraphs

Suggested new material for introduction to the chapter:

FMSL specifications consist primarily of object and operation definitions. The following is a simple illustrative example.

```
object PersonList
    components: Person*;
    description: (*
        A PersonList contains zero or more Person records.
    *);
end PersonList;

object Person
    components: first:Name and last:Name and age:Age;
    description: (*
        A Person has a first name, last name, and age.
    *);
end Person;

object Name = string;
object Age = integer;

operation Add
    inputs: p:Person, pl:PersonList;
    outputs: pl':PersonList;
    precondition: not (p in pl);
    postcondition: p in pl';
    description: (*
        Add a person to a list, if that person is not already in the list.
    *);
end Add;
```

This example illustrates the two primary forms of definition in FMSL: objects and operations. Objects have components, which are defined in terms of other objects. Object definitions "bottom out" in one of the built-in primitive types of `integer`, `real`, `string`, or `boolean`.

Operations have inputs, outputs, preconditions, and postconditions. The types of inputs and outputs are the names of defined objects. Preconditions and postconditions are boolean expressions.

Other notational features worthy of explanation are the following:

- '`(*`' and '`*)`' are used to enclose comments
- `Name` and `Age` use an optional short form of object definition; it can be useful for objects of simple scalar types, with no description
- the `in` operator is built-in; it tests for list membership
- any identifier can have an apostrophe character as a suffix; this is purely a lexical form, in that a trailing apostrophe is a legal character in an identifier; it is used most often in operation outputs when the type of an input and output object are the same; e.g., the `Add` input list is named `pl` and the output list is `pl'`, read "*pl prime*"

A complete discussion of FMSL syntax and semantics is given in its reference manual [45]. This thesis will only use a subset of its features, specifically those features that are germane to the topic of specification validation.

Given a specification such as the example above, a basic question is this: "How does one validate that it is correct?" Firstly, static correctness can be validated using the FMSL type checker, which performs syntactic and semantic analysis comparable to that performed by a programming language compiler. A particularly useful part of static analysis is completeness checking. For example, if the specifier left out the definitions of the `Name` and `Age` objects, the checker would flag the error in the definition of the `Person` object that uses `Name` and `Age`.

The focus of this thesis is determining the dynamic correctness of a specification. For an operation, this fundamentally requires some means of evaluation. In the example at hand, the Add operation could be evaluated in the following manner:

```
value p:Person = {"Arnold", "Schwarzenegger", 61};      -- a sample person
value pl:PersonList = [];                               -- an empty person list
value pl':PersonList = [p];                             -- a one-person list

> Add(p, pl);                                           -- invoke Add operation
```

The following aspects of notation warrant brief explanation:

- a `value` declaration defines a constant value of some type of object
- tuple values are enclosed in curly braces; a tuple is an object defined with `and`ed components
- list values are enclosed in square brackets; a list is an object defined with * components
- point-to-end-of-line comments are defined with '`--`'
- expression evaluations are preceded with the prompt character '`>`'; these are typically entered in the top-level of a conversational interpreter, but may be included within a specification file; the important point is that the '`>`' prompting character distinguishes an expression to be evaluated from a specification declaration, in this and all subsequent examples.
- an operation is invoked in the way standard to most programming languages, with the operation name followed by a parenthesized list of actual parameters

So, the question at hand is *"What value does the invocation of `Add(p, pl)` produce?*" Since the `Add` operation has no defining expression, the value of invoking `Add(p, pl)` is `nil`, where `nil` is the empty value for any type of object. `Nil` is in fact is result of evaluating `Add` for any inputs, given that `Add` is defined only with a precondition and postcondition.

The precondition and postcondition for `Add` define a behavior. However, they do so in a declarative and analytic form, not a constructive form. It is possible to define FMSL operations constructively, but that is not the point here. What is desired is a way to validate `Add`'s precondition and postcondition, given a particular set of inputs and expected outputs.

One way to do this is to extract the precondition and postcondition expression, and evaluate them individually. For example, given the preceding `value` declarations, the precondition expression could be tested with logic expressions such as this:

```
> p in pl;                      -- should be false
> not (p in pl);                -- should be true
> not (p in pl');               -- should be false
```

The postcondition expression could be tested like this:

```
> p in pl';                     -- should be true
> not (p in pl');               -- should be false
```

These are clearly rudimentary expressions. The point is that the logic of preconditions and postconditions can be dynamically validated by plugging in various values and examining the results. The work of this thesis has included the implementation of this form of expression evaluation in FMSL. This form of evaluation supports the notion cited earlier from Myers [6]: "if you run simple claims early, ... then you have a basis for understanding both the model and the system".

While isolated evaluation of boolean expressions can be helpful, it would be even handier to invoke an operation with sample input and output values directly. This kind of *validation invocation* can be characterized as follows for the `Add` precondition:

> *Given inputs p and pl, what is the value of the Add precondition?*

A more complete validating invocation is this:

> *Given inputs p and pl, expected output pl', what are the values of the Add precondition and postcondition?*

The syntax for such a validation invocation looks like this:

```
> Add(p, pl) ?-> pl';
```

The output of this validating invocation is a boolean two-tuple, that looks like this:

```
{ true, true }
```

The notational particulars are these:

- the first part of a validation invocation looks like a regular operation call, e.g., `Add(p, pl)`
- the '`?->`' is the validation operator[†]; per the preceding characterization, it means the following in this example: *Given inputs p and pl, is the Add precondition true, and given pl', is its postcondition true?*
- the output value of `{ true, true }` is the standard curly brace notation for a boolean two-tuple

A validation counter example can be tested, such as

```
> Add(p, pl) ?-> pl;
```

which produces the result `{ true, false }`

The preceding introduction to Chapter 3 has presented a simple motivating example. The remainder of this chapter will cover the details of specification evaluation, including in particular the evaluation of conditions with quantifiers. The coverage will feature the validation of a long-standing pedagogic example, in which the use of validating evaluations revealed a heretofore undiscovered flaw. This is a particularly good result, and demonstrates well the utility of dynamic specification validation.

[†] The somewhat curious syntax of the validation operator is derived from the FMSL syntax for operation signatures; i.e., the signature of the Add operation is `(Person, PersonList) -> PersonList`, where the `->` notation has been used in other specification languages in the denotation of input/output signatures.