Notes on Validation Invocations

Introduction

The purpose of a "validation invocation" is to support incremental testing of a specification, and subsequently an implementation. With a normal invocation, an operation is supplied input values, and the invocation produces output values. In a validation invocation, an operation is supplied *both* input and output values, and the invocation produces boolean values that indicate if the operation's preconditions and postconditions are satisfied.

Here's an example:

```
op Foo(i:integer, s:string)->(i':integer, s':string)
    pre: i > 0;
    post: (i' = 10) and (s' = "abc");
end;

op Test() =(
    Foo(1,"xyz")?->(10,"abc");    (* Produces value {true, true} *)
    Foo(1,"xyz")?->(9,"abc");     (* Produces value {true, false} *)
    Foo(0,"xyz")?->(10,"abc");    (* Produces value {false, nil} *)
);
```

In the discussion that follows, the term "validation" will be used as a shorthand for "validation invocation".

In general, a validation of the form

`op-name(value, ...)?->(value, ...)`

produces a tuple of type *`boolean and boolean`*. The first tuple element is the value of the precondition expression; the second tuple element is the value of the postcondition expression. When the value of the precondition is `false`, the value of the postcondition will always be `nil`.

The precondition and postcondition expressions are evaluated with the supplied values for the input and output parameters of the operation. Since preconditions and postconditions can only reference parameter variables, the evaluations will always be possible.

A validation is similar to a test case in unit test plan. Such test cases are defined as a list of input values and a list expected output values. Test plans are typically in a tabular form, such as the following:

Unit Test Plan for Operation Foo:

Case No.	Inputs	Expected Outputs	Remarks
1	`i=1, s="xyz"`	`i'=10, s'="abc"`	Simple test case that should succeed.
2	`i=0, s="xyz"`	`nil`	Output is `nil` because precondition fails.
	...		

The execution of a test case goes like this:

1. bind the supplied input values to the operation's input parameters
2. evaluate the body of the operation
3. compare the actual output values of the operation with the expected values

The test case succeeds if the actual output values match the expected values. It fails otherwise.

In comparison to a normal test cast, the execution of a validation goes like this:

1. bind the supplied input values to the operation's input parameters *(same as for the test case)*
2. evaluate the operation's precondition and postcondition *(as opposed to evaluating the operation body)*
3. return a boolean two-tuple, containing the values of the precondition and postcondition *(as opposed to performing a comparison of expected and actual outputs)*

The execution of a test case and a validation are *similar*, but have different intents. The purpose of a test case is to determine if the implementation of an operation produces the correct output. The purpose of a validation is to determine the validity of the preconditions and postconditions for an *unimplemented* operation.

Test cases can be thought of as a way to incrementally debug an implementation. When a test case fails, it means either the expected outputs weren't what they were supposed to be, or more often, that the expected outputs were correct but the implementation failed to produce them when it should have. In this case, one looks at the implementation to figure out what went wrong.

Validations can be thought of as away to incrementally debug a specification. In particular, when the boolean value of the validated postcondition is false, it means that either the given outputs weren't what they were supposed to be, or or more often, that the postcondition does not accurately define the correct output condition. In this case, one looks at the condition logic to figure out what's wrong.

Need to supply a couple examples to illustrate concretely how validations are used to debug postconditions. Also, we'll discuss the fact that verified validations can be 100% reused as implementation-level test cases.

Running Validations

Based on the preceding description, running a validation involves the evaluation of the precondition and postcondition expressions. Except for quantifiers, this evaluation is straightforward -- it's the same as normal boolean expression evaluation.

For conditions that contain quantifiers, the trick for evaluation is to determine the value set to use. This is still straightforward for closed-form quantifications, i.e., ones involving the '`in`' form in `fmsl`. For example, the evaluation of the following quantified expression

```
forall (x in l) x >= 0
```

entails the evaluation of `#l` sub-expressions, with `x` successively bound to each value of l, in any order. The value of the entire `forall` expression is the boolean `and` of all the subexpressions.

The only "trick" in quantifier evaluation is for the unbounded forms. Consider this example,

```
obj UserRecord = name:string and id:integer and ...
forall (ur: UserRecord) ur.name != nil
```

The question here is for what particular `UserRecord` values should the quantifier be evaluated? There are these approaches:

1. use some heuristics to determine a "reasonable" set of values to use; such heuristics are derived from well-known techniques for automated test case generation
2. use actual values of `UserRecord` variables that have been bound during preceding validations of the operation in which the quantified expression appears
3. use actual values of `UserRecord` variables that have been bound during an entire interactive session, for all validated operations
4. use actual values of `UserRecord` variables that have been bound during an entire interactive session, for all validated and invoked operations

These approaches are not entirely mutually exclusive.

What follows is a sketch of some further details, to be expanded upon in the coming weeks.

For your thesis, dealing with the first approach should involve a small subset of known heuristics, as a proof of concept. In lieu of detailed work with these heuristics, you could implement a GUI that allows developers to enter sample values manually, in the form of tabular test plans.

Dealing with approaches 2-4 uses the same basic mechanism -- caching values when operations are validated, and if necessary, invoked. Caching in this context simply means recording a set of input values for each operation, and using that set as the current means to bound any quantifier evaluation.