Requirements for the Spec Validator
an Incremental Specification Validation Tool

Version 0.1
19 February 2009

Contents

1. Introduction

These requirements describe a specification validation tool, called the "Spec Validator". The purpose of the tool is to allow the developer of a specification to test the specification incrementally, as it is being developed. The specification is written as an object/operation model, with formal behavior defined as preconditions and postconditions on the operations.

The testing of the specification entails providing sample inputs and outputs to the specified operations. The test inputs are "plugged in" to the precondition, which is evaluated to produce its boolean result. If the precondition is true, the test outputs are plugged into the postcondition to produce its boolean result. The user then examines the evaluated results to confirm that the precondition and postcondition evaluate as expected.

For a test case of correct behavior, the precondition and postcondition should both evaluate to true. For a test case with invalid inputs, the precondition should be false, and the postcondition nil. A test case can reveal a flaw in the specification if the user expects the results to be true, but the evaluation says otherwise. For example, a postcondition error is detected when a test case has outputs that are known to be correct, but the postcondition evaluates to false.

The functional requirements of the Spec Validator present examples of typical usage. The examples illustrate how the Validator can be a productive part of specification development.

2. Functional Requirements

The Spec Validator provides functionality to load a specification and define test cases for each operation. When one or more test cases have been defined, the tool allows the cases to be individually validated, or validated altogether. The tool also provides an editing interface for the cases themselves, as well as for the preconditions and postconditions being tested.

Following an overview of the Spec Validator user interface, details of tool functionality are presented in the following usage scenarios:

- an introductory example of specification validation
- details of plan editing
- details of specification validation
- details of file load and save functions

2.1. User Interface Overview

When the user launches the Spec Validator, the tool displays its initial interface, as shown in Figure 1. The user may provide the name of a specification to be tested when the tool is invoked, e.g., by command-line argument, or other appropriate platform-specific means to supply an initial working file.

Once a specification is loaded, the user enters the name of an operation to be tested in the ’Operation’ text field. To do so, the user types the full name of an operation, or an unambiguous leading prefix for an operation name. When the user presses the TAB or ENTER key, the tool locates the operation in the current specification, and completes the entry in the ’Operation’ text field. The completion is the full spelling of the operation name, and the operation input/output signature. If an operation name is overloaded in the specification, the user can disambiguate by typing one or more input argument types.

As an alternative to typing the operation name, the user may press the ’Browse’ button next to the text field. When the user does so, the tool displays an alphabetic list of operation names, from which the user can select.

When an operation is selected, the tool displays its precondition and postcondition expressions in the two text areas immediately below the ’Operation’ field. If the operation specification has a description, the tool displays it in the so-labeled text area.

The ’Operation’ field, and the three text areas below it are all user editable. If the user discovers that changes need to be made to any of the information in these texts, the user may perform the edits and save the changes.

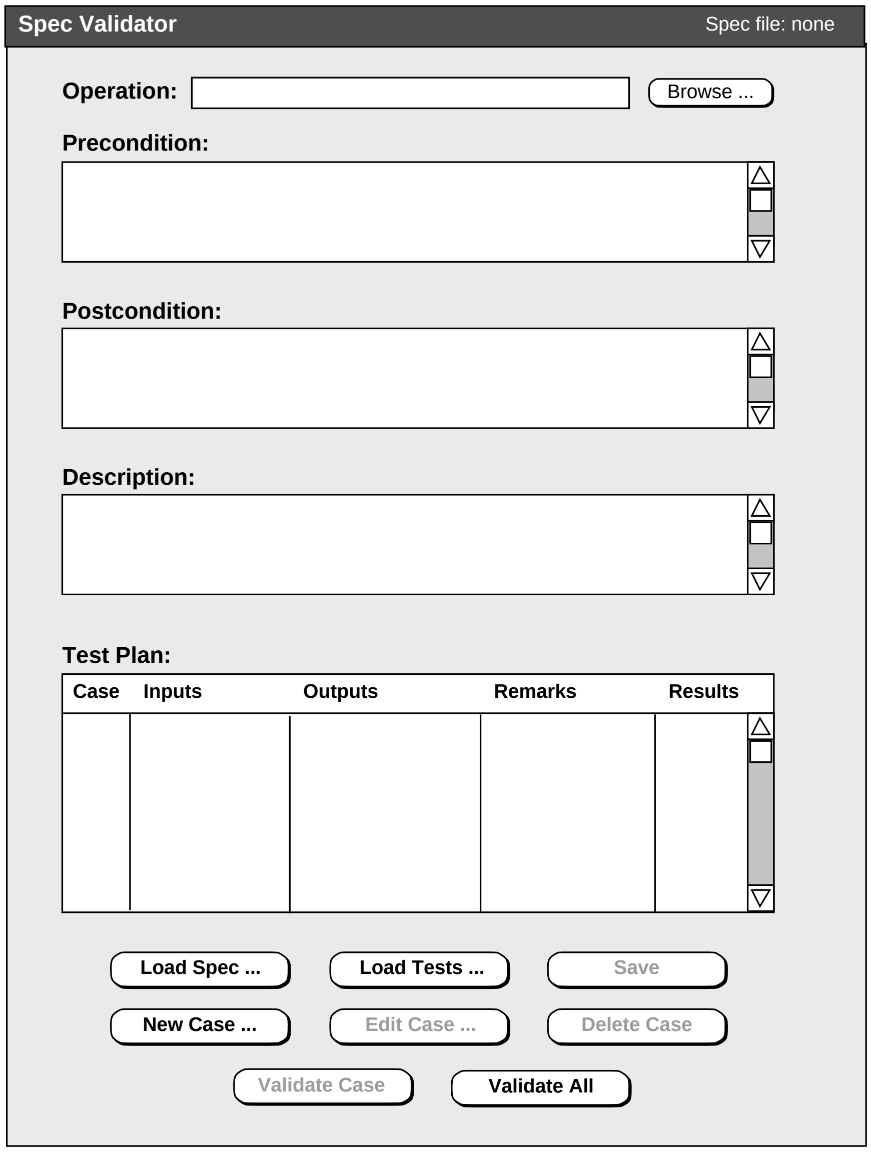

Figure 1: Initial User Interface.

The ’Test Plan’ edit area is the primary focus of the interface. It is where the user enters test cases to be used to validate the logic of the precondition and postcondition. Upcoming examples illustrate the details of defining these test cases.

The eight buttons on the bottom of the display perform the following actions:

- ’Load Spec’ provides a file chooser dialog in which the user selects a specification file to load.
- ’Load Tests’ provides a file chooser dialog in which the user selects a previously-saved testing file.
- The ’Save’ button saves any changes the user has made, to the specification and/or test plan; specification changes are saved in the most-recently loaded specification file; test plan changes are saved in the loaded test file, or in a file of the user’s choice if no test file has yet been loaded.
- ’New Case’ opens a test-case editing dialog in which the user enters the details of a test case

- The ’Edit Case’ button is enabled when the user selects an existing test case in one of the rows of the Test Plan; when the user makes such a selection, pressing New Case opens the test-case editing dialog on the selected test case.
- The ’Delete Case’ button is enabled when the user selects an existing test case in one of the rows of the Test Plan; when the user makes such a selection, pressing Delete Case removes that test case from the plan.
- The ’Validate Case’ button is enabled when the user selects an existing test case in one of the rows of the Test Plan; when the user makes such a selection, pressing Validate Case performs the validating evaluations on that one case
- ’Validate All’ performs the validating evaluation on all cases of the test plan.

Details of all of these operations are presented in the usage scenarios in the upcoming sections of the requirements.

2.2. A Basic Example of Specification Validation

Pick an operation with three or four inputs, one or two outputs, and interesting but not too-terribly detailed preconditions and postconditions. Show a couple examples of true and false validation cases. Have an initial bug in the postcondition, show how it's discovered, and then fixed.

2.3. Details of Plan Editing

Cover details of plan editing that weren't fully covered in Section 2.2. Since we're doing requirements for a simple proof-of-concept tool, there may not be anything to cover in this section, if 2.2 can present enough of what we're after.

2.4. Details of Specification Validation

Cover details of spec validation that weren't fully covered in Section 2.2. As with Section 2.3, there may not be anything to cover in this section.

Whether or not this section expands further, here's the current thinking on the detection of specification problems:

Validity of Inputs and Outputs	**Precondition Value**	**Postcondition Value**	**Problem**
One or more input values is known to be invalid.	`true`	*don't care*	The precondition is too weak, or has flawed logic.
All input and output values are known to be valid.	`true`	`false`	The postcondition logic is flawed.
All input values are known to be valid, but one or more output values are known to be invalid. This condition is provided as a challenge to the postcondition logic.	`true`	`true`	The postcondition logic is flawed.

The third entry in the table seems like it might be kind of interesting, in that it I don't recall it having been discussed in the testing literature I've read. The idea is that we provide test cases that are a deliberate challenge to the output of the function under test. If it is an interesting idea, it's doubtless been discussed somewhere, I just don't remember seeing it.

2.5. Details of Loading and Saving Files

Cover details of loading and saving specs and tests, that weren't fully covered in Section 2.2. As with 2.3 and 2.4, there may not be anything to cover here.