CSC 509 Lecture Notes Week 4

Using Formal Specs to Support Testing (Monday)

Class Project Proposals (Wednesday)

I. Quick Notes about "How to Read a Paper"

A. A useful little ditty.

B. Particularly like observation about literature search: "... if you are lucky, [you'll find] a pointer to a recent survey paper [and then] you are done."

II. Very common refrains about manual test case generation, as performed by humans:

A. It’s tedious.

B. It’s boring.

C. It’s error prone.

D. It may leave important things untested.

E. There’s got to be a better way.

III. Questions about the readings:

A. In the overall field of software testing, how significant is the subject matter in the survey paper?

B. How does the JML tools paper relate to the survey paper?

C. How does the jmlunitng paper relate to the general JML tools paper?

IV. **Answers, using a cosmologic metaphor:**

A. literature outlined in Lecture Notes 1 covers the ***galaxy*** of software testing.

B. Survey paper covers rather remote ***galactic neighborhood*** of auto test gen from specs.

Software testing cosmology, cont'd

C. The JML tools paper talks about ***one small solar system*** in the larger neighborhood.

1. Note that in 75 pages survey paper only mentions Java a few times in passing.
2. It never references JML specifically.
3. Survey authors evidently don't think much of the JML solar system.

Software testing cosmology, cont'd

4. The jmlunitng paper talks about ***one pretty small planet*** in the JML solar system, still in its formative stages.

V. My impressions of the survey paper.

A. I rather disagree with their statement at the outset that "Traditionally formal methods and software testing have been seen as rivals"

1. I think this invokes some rather old "traditions".

2. Authors go on to say that in a fact for some time "these approaches are seen as complementary"

Impressions of survey paper, cont'd

B. Otherwise, I very strongly agree with the authors statements in the paper introduction, including in particular these:

1. combined formal analysis of specification and test could provide very strong guarantees of correctness

2. information gathered by testing may assist when using a formal specification

Impressions of survey paper, cont'd

3. testing can be used in order to provide initial confidence in a system before effort is expended in attempting to prove correctness

4. Where it is not cost-effective to produce a proof of conformance, the developers may gain confidence in the SUT through systematic testing.

Impressions of survey paper, cont'd

5. this might be complemented by proofs that critical properties hold

6. a proof of correctness might also use information derived during testing

Impressions of survey paper, cont'd

7. Finally, a proof of correctness relies upon a model of the underlying system and dynamic testing might be used to indirectly check that this model holds.

8. An interesting challenge is to generate tests that are likely to be effective in detecting errors in the assumptions inherent in a proof.

Impressions of survey paper, cont'd

C. Overall, they're espousing one of my favorite themes in software development -- *multiple views of the same artifact can be very helpful indeed*

D. in this context we have these multiple views

1. the spec
2. the generated test cases
3. the code

VI. "Heavyweight" versus "Lightweight" methods.

A. *Heavyweight* methods are fully formal, based on fully mechanized logics such as

1. theorem provers
2. resolution-based model checkers
3. constraint solvers

Heavyweight versus Lightweight, cont'd

B. *Leightweigh* methods based on a formal spec, but lec they do not employ fully or at all the mecha-nized logics of the heavyweight methods;

1. instead, the lightweight approaches provide some form of implementation that uses the specifica-tion as a data structure from which a non-exhaustive but "good" set of tests are generated.

VII. On the weightiness of the three papers

A. The survey is pure heavyweight stuff.

B. The JML tools paper talks about some moderately heavy weight, medium weight, and lightweight tools.

C. The JMLUnitNG paper self describes its tool as "extremely lightweight".

VIII. **Snarky swipe at "noweight" informal testing.**

A. Based on an ad hoc "think clearly about it" test generation methodology.

B. Use an ad hoc oracle definition.

C. Use test coverage tools to mitigate ad hocness, often without full satisfaction.

IX. Some Highlights of the Survey Paper --

A. Centerpiece of the survey is the focus on five different styles of formal specification:

1. Model-based
2. State machines
3. Concurrency formalisms
4. Hybrid digital/analog
5. Algebraic

X. Model-Based, e.g., Z, JML

A. Most directly relevant to software in the systems and information processing domains typical for end-user applications.

B. Most accessible to programmers.

XI. FSMs, e.g., State Charts

A. Used in communication systems and other forms of apps that can be aptly characterized using FSMs

B. I personally find this form of specification obtuse, tedious, and not relevant for many forms of end-user software.

XII. Concurrency formalisms, e.g., CSP

A. An essential formalism for modeling and testing concurrent systems.

B. A single-thread formalism such as Z and JML simply *must* have some additional mathematical representation to deal effectively with multi-threaded software architectures.

C. Another approach to this not mentioned in the survey is Lamport's *temporal logic*.

XIII. Hybrid math models, e.g., CHARON

A. As a practical mater, this style is more applicable to systems-level and embedded software than end-user software.

B. Testing approaches focus on the use of simulation.

C. The survey paper questions the practicality of this approach in general, and I agree particularly for end-user software.

XIV. Algebraic, e.g., OBJ, Maude

A. This is a powerful and elegant approach to specification.

B. Among other things, the specification is itself fully executable

1. an execution and a proof are the same thing
2. execution is performed by term reduction in essentially the same form as term reduction is used in mechanized algebraic proofs

Algebraic specs, cont'd

C. The specification is 100% "model free", in that are are no concrete data models defined.

D. Behavior is defined entirely in terms of an equality definition of operation behavior, with the only data model per se being that of a *term*.

Algebraic specs, cont'd

E. To test an algebraic specification, one can use the same form of inductive partition of inputs as for model-based specification

F. The fundamental problem with algebraic specs is that the mapping from the specification to a conventional sequentially-executing program is not at all straightforward.

1. With model-based specs, the specification is predicative annotation that is directly attached to the program.
2. With a algebraic specification, the specification is associated the the program at the class level.

Algebraic specs, cont'd

G. And alas, the mathematics involved in this form of specification is sufficiently dense and inaccessible to most software developers as to render this approach to specification and subsequently testing impractical.

H. This is a shame really, given the supreme elegance of algebraic specification.

-- Quick Highlights of JML Tools Paper --

A. It describes what's on offer from the JMl crowd in addition to automated testing.

B. The majority of high-end research is on verification and other forms of static and dynamic analysis.

-- Highlights of the JMLUnitNG Paper --

A. Describes a tool that generates JUnit-style tests from JML specs.

B. There are three aspects to such a tool, as outlined in the survey paper and embodied in this particular tool (among many others):

JMLUnitNG, cont'd

1. Choose a spec language -- JML in this case.
2. Choose a well-known test generation technique -- exhaustive range and input combination in this case
3. Implement in a existing spec language translation environment -- java4c in this case

-- Highlights of Three Different Spec Languages --

XV. There's clearly a "tower of babel" problem with the diversity of different specification languages, all of which do the same thing

A. We'll look at two model-based languages -- Z and JML

B. We'll also have a quick look at an algebraic language -- OBJ

Spec language tower of babel, cont'd

C. There's no question that the diversity and obtuse-ness of spec languages is at least one important factor in the lack of wide-spread adoption.

XVI. A sample Z spec

paper handout, online at classes/509/examples/lecture4/Stack.z.pdf

XVII. The equivalent JML spec

paper handout, online at classes/509/examples/lecture4/Stack.java

XVIII. JML & Z side-by-side comparison

paper handout, online at classes/509/examples/lecture4/StackZ.java

XIX. The equivalent OBJ spec

paper handout, online at classes/509/examples/lecture4/Stack.obj

XX. 509 projects in this area

A. I continue to be interested in having an auto-test generation tool that is usable CSC 307 and 309.

B. The jmlunitng tool is the closest I've seen to such a tool so far.

C. In order to be deployable in 309, in needs some key improvements:

Key improvements to jmlunitng, cont'd

1. Upgrade from Java 5 to Java 7 or 8.

2. Eliminate the combinatorially explosive number of test cases using well-known techniques introduced by Weyuker.

   Provide support for full oracle executability, adapting ideas from a Cal Poly MS thesis by Paul Corwin.

Key improvements to jmlunitng, cont'd

D. Summary of the approach to full execution:

1. Refine jmlunitng notion of using constructor tests to create "worlds" of test fixture objects, specifically having worlds for each defined data type.

Key improvements to jmlunitng, cont'd

2. As in Corwin thesis, implement executability of unbounded quantifiers by substituting a finite number of values from type-specific worlds to bound the range of any unbounded quantifier

Key improvements to jmlunitng, cont'd

3. E.g., `forall (Stack s ...)` is bounded by the number Stack objects otherwise created for test generation

4. Another is program transformation of unbounded quantification to a bounded form, using patterns employed by humans

5. When such pattern-based transformation cannot be achieved, then fall back on "finite-world" mechanism.

XXI. Review of well-know test generation rules

A. The rules are surprisingly straightforward.

B. They're employed explicitly or implicitly by humans when generating tests.

C. Mechanizing them is straightforward as well, one you're into the details of the spec language translation environment.

D. An overview of the rules follows.

XXII. Black box testing rules

XXII. **Black box testing rules**

A. Provide inputs where the precondition is true, varying inputs to exercise precond logic.

XXII. **Black box testing rules**

A. Provide inputs where the precondition is true, varying inputs to exercise precond logic.

B. Provide inputs where the precond is false, ***if not a by-contract method.***

Black box rules, cont'd

B. For data ranges:

Black box rules, cont'd

B. For data ranges:

1. Provide inputs below, within, above each pre-cond range.

Black box rules, cont'd

B. For data ranges:

1. Provide inputs below, within, above each pre-cond range.

2. Provide inputs that produce outputs at bottom, within, at top of each postcond range.

Black box rules, cont’d

Black box rules, cont'd

C. With and/or logic, provide test cases that fully exercise logic.

Black box rules, cont'd

C. With and/or logic, provide test cases that fully exercise logic.

1. Provide an input that makes each clause both true and false.

Black box rules, cont'd

C. With and/or logic, provide test cases that fully exercise logic.

1. Provide an input that makes each clause both true and false.

2. This means 2^n test cases, where n is number of logical terms.

Black box rules, cont’d

Black box rules, cont'd

D. For collection classes:

Black box rules, cont'd

D. For collection classes:

1. Test empty collection.

Black box rules, cont'd

D. For collection classes:

1. Test empty collection.
2. Test with one, two elements.

Black box rules, cont'd

D. For collection classes:

1. Test empty collection.
2. Test with one, two elements.
3. Add substantial number of elements.

Black box rules, cont'd

D. For collection classes:

1. Test empty collection.
2. Test with one, two elements.
3. Add substantial number of elements.
4. Delete each element.

Black box rules, cont'd

D. For collection classes:

1. Test empty collection.
2. Test with one, two elements.
3. Add substantial number of elements.
4. Delete each element.
5. Repeat add/del sequence.

Black box rules, cont'd

D. For collection classes:

1. Test empty collection.
2. Test with one, two elements.
3. Add substantial number of elements.
4. Delete each element.
5. Repeat add/del sequence.
6. Stress test with order of magnitude greater than expected size.

XXIII. 509 Projects in Spec-Based Testing

A. Recap of improvements to JMLUnit[NG]:

1. Reduce combintorially explosive number of test cases by applying preceding rules.

2. Improve postcond executability (Corwin thesis).

3. Memoize test execution results (Bolef thesis).

B. Other ideas for a 509 projects in this area:

1. Provide *"syntactically sugared"* GUI.
2. Provide testing of both spec and code.
3. Generate tests for multiple languages.

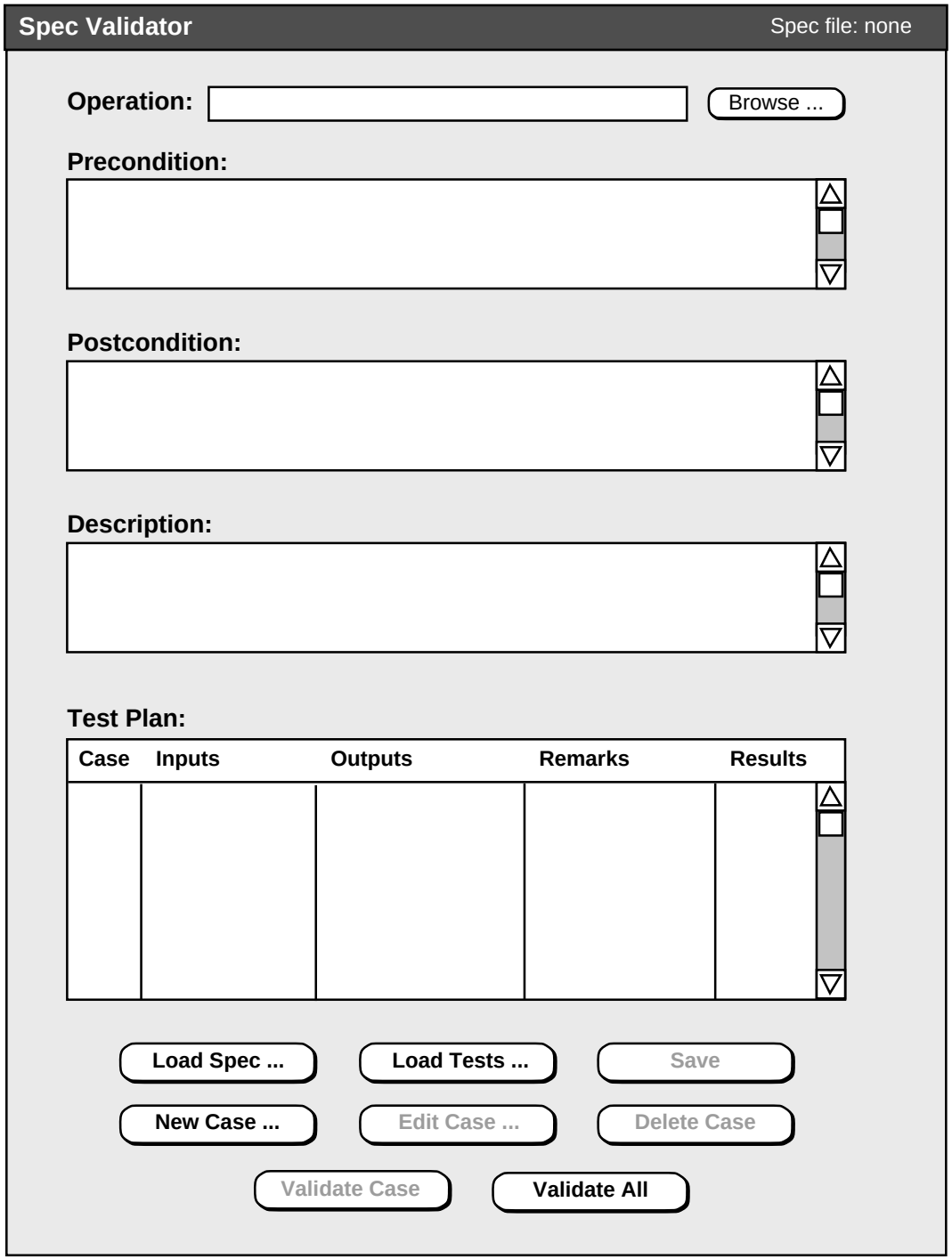
Other ideas, cont'd

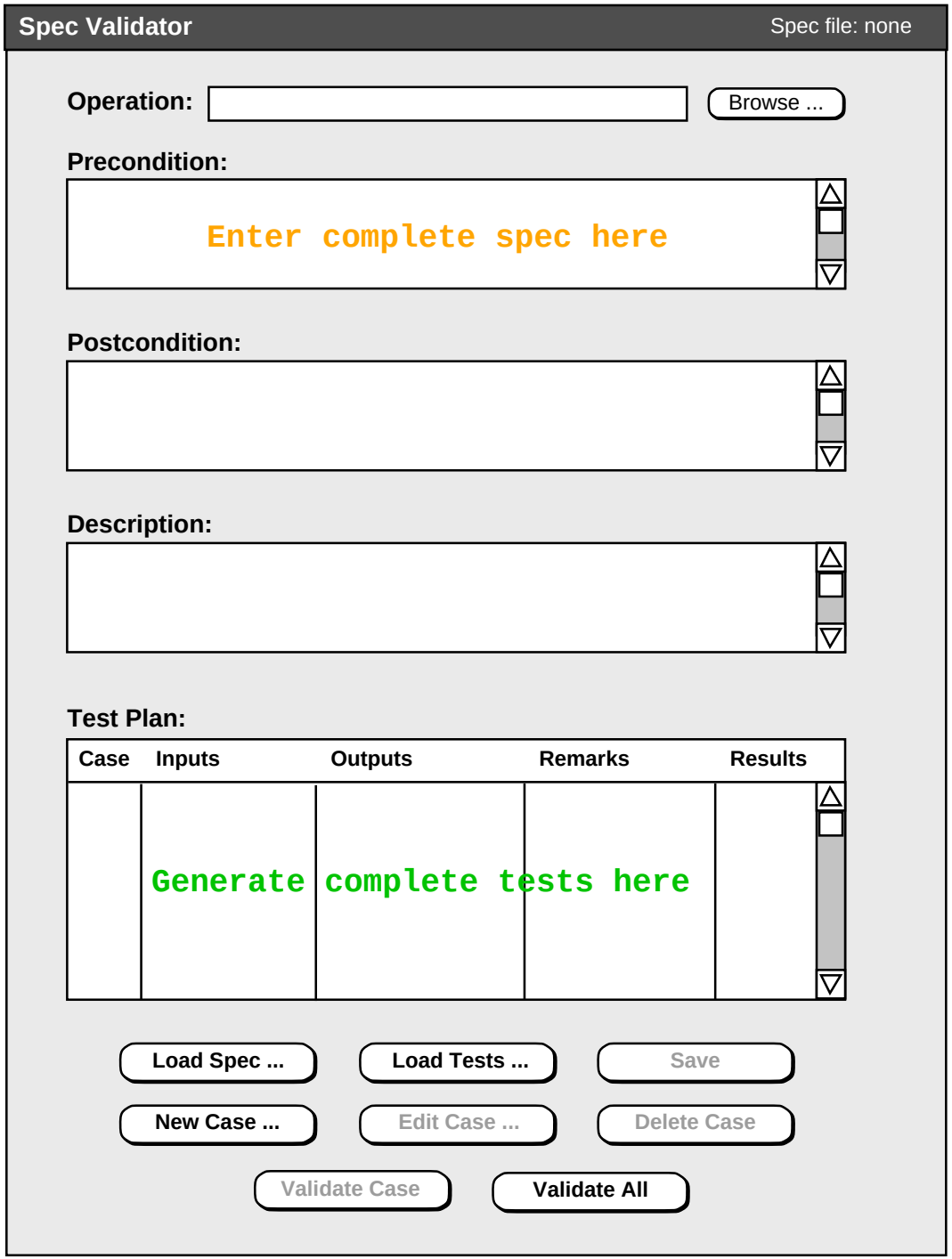
4. Translate pseudo-code test cases into compilable xUnit code (IBM Rex Tool)

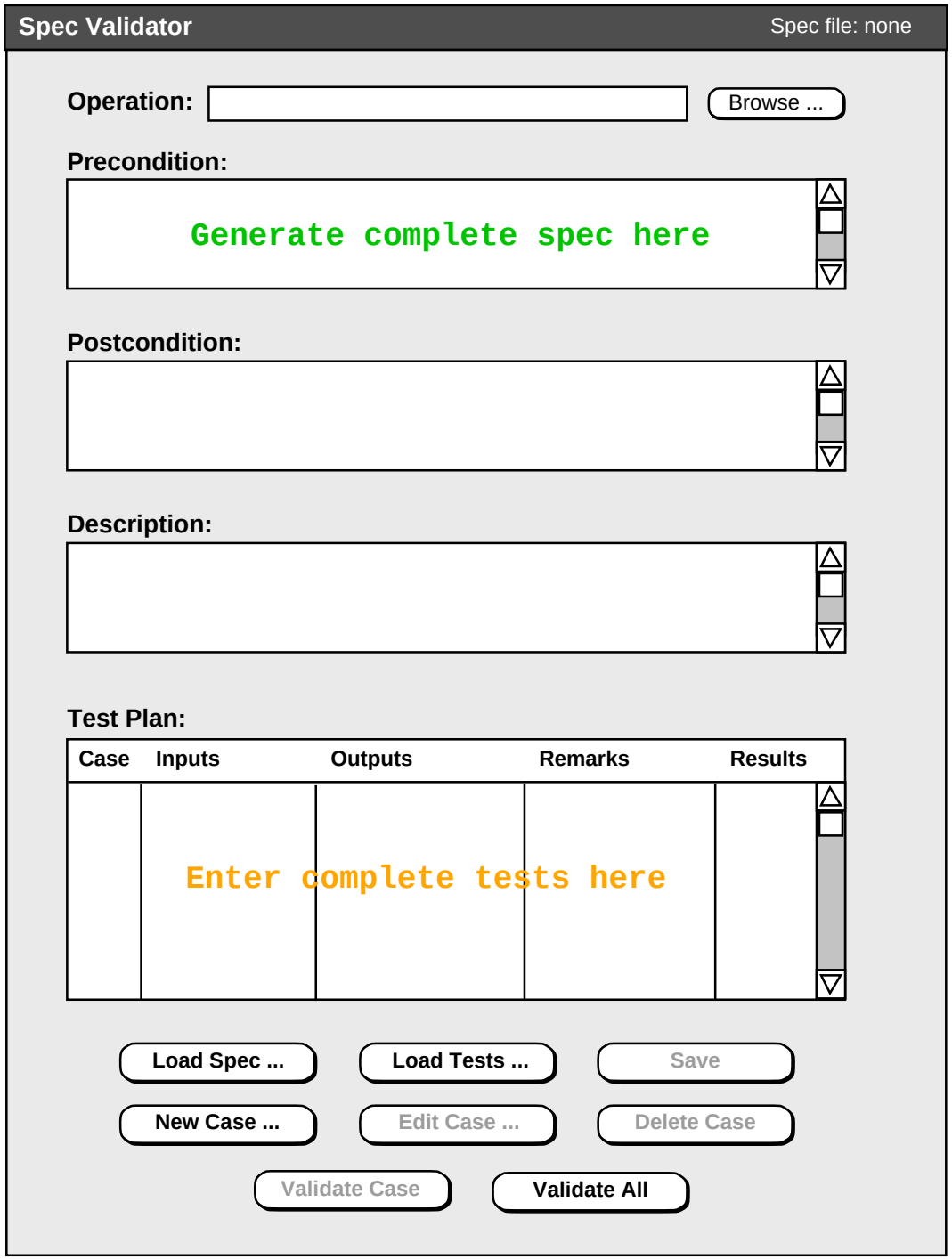
5. Generate specs from test cases (requires AI).

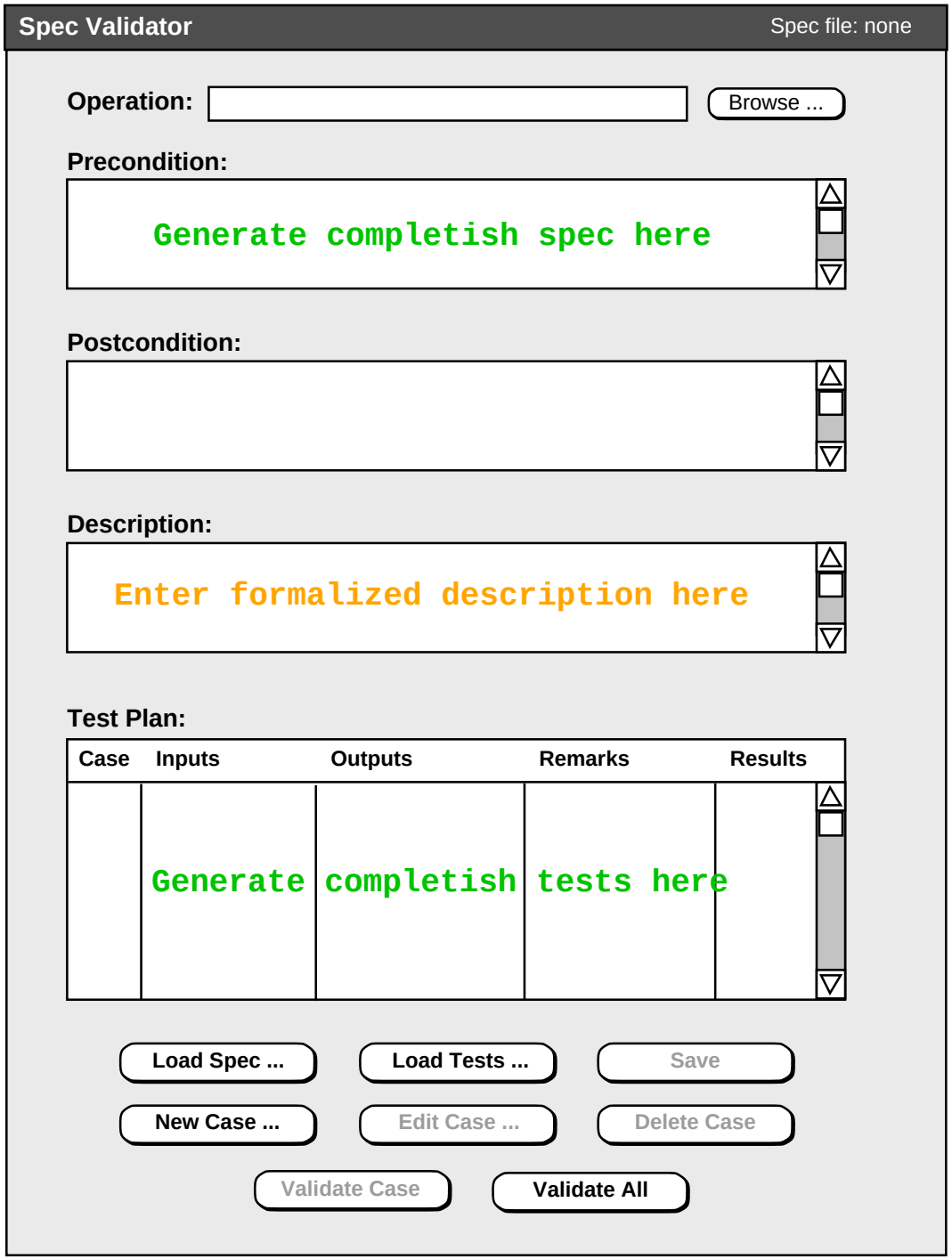
6. Generate specs from descriptions (requires mucho AI, NLP).

A sample UI for a spec-based testing tool


Spec Validator
Spec file: none
Operation:
Browse ...
Precondition:
Postcondition:
Description:
Test Plan:
Case
Inputs
Outputs
Remarks
Results
Load Spec ...
Load Tests ...
Save
New Case ...
Edit Case ...
Delete Case
Validate Case
Validate All


Spec Validator
Spec file: none
Operation:
Browse ...
Precondition:
Enter complete spec here
Postcondition:
Description:
Test Plan:
Case
Inputs
Outputs
Remarks
Results
Generate complete tests here
Load Spec ...
Load Tests ...
Save
New Case ...
Edit Case ...
Delete Case
Validate Case
Validate All


Spec Validator
Spec file: none
Operation:
Browse ...
Precondition:
Generate complete spec here
Postcondition:
Description:
Test Plan:
Case
Inputs
Outputs
Remarks
Results
Enter complete tests here
Load Spec ...
Load Tests ...
Save
New Case ...
Edit Case ...
Delete Case
Validate Case
Validate All


Spec Validator
Spec file: none
Operation:
Browse ...
Precondition:
Generate completish spec here
Postcondition:
Description:
Enter formalized description here
Test Plan:
Case
Inputs
Outputs
Remarks
Results
Generate
completish
tests here
Load Spec ...
Load Tests ...
Save
New Case ...
Edit Case ...
Delete Case
Validate Case
Validate All

XXIV. Additional reading for 509 project.

A. Weyruker paper

B. Corwin thesis

C. Korat paper

D. Executable JML specs paper