PAL UI Editor Test Plan

1. Overview

The test plan for the PAL UI consists of both automated and manual test cases. Priority will be given to the development and execution of manual test cases since we lack a good framework for automated UI testing and can leverage the intelligence of human testers to detect additional problems and derive additional test cases. Similarly, automated integration tests should be given higher priority than automated unit tests since they perform operations more akin to those triggered by an end-user and few elements in the UI lend themselves to effective unit testing. This should provide the highest level of confidence for release. If the development of the PAL UI continues beyond our October, 2011 release, the test framework should gradually become more automated as our confidence in the deliverable increases and the test cases are better understood.

Both the automated and manual test cases should be built upon a common test domain(s). A set of canned test procedures will be defined within the test domain for use by both manual and automated tests. Other procedures can be created as needed to support edge cases, mutation testing, etc but may be transient within our test cases.

1. Test Domain

The primary test domain will be an extension of the synthetic “Arda” test domain, which exposes the types in task learning directly. Arda currently lacks usage of all supported primitive types, type inheritance and enum types. We should consider adding these to the action model and test plan once the base test plan is in place.

We will also use Novo as our test domain for our end-to-end system tests, particularly for the Adept 1.1 release. While Novo’s action model does not thoroughly exercise the type system, it does provide a good means for system testing and testing of learning/execution.

An additional test domain/action model may be developed with the express purpose of stressing the UI with long strings and testing the editor’s custom icon functionality for namespaces and actions.

1. Canned Test Procedures
   1. Valid Procedures
      1. Core
         1. Empty Procedure
         2. SingleAction
         3. Linear procedure with support relationships. No iteration, inputs or results
         4. Procedure with single input
         5. Procedure with single output
         6. Procedure with multiple inputs/outputs
         7. Procedure with constants
         8. Procedure that includes other procedures as steps
         9. Procedure that utilizes NullValueModel
         10. Procedure that utilizes NoEvalTermModel
      2. Types
      3. Collections/Functions
         1. Lists
         2. Sets
         3. Bags
         4. Structs
         5. MapGet/Nth
         6. First/Last
         7. Nested function calls
      4. Iteration
         1. Procedure with iteration
         2. Procedure with multiple levels of nested iteration
         3. Procedure with an empty loop
      5. Stress
         1. A procedure with more than 100 steps
         2. A procedure with a large number of procedure inputs
         3. A procedure with a large number of procedure outputs
         4. Procedure with long variable names, constant values, step names, etc
         5. Procedures that uses 3.1.5.2 & 3.1.5.3 as steps to stress action step visualization
      6. Miscellaneous
   2. Procedures with Warnings/Errors
      1. Procedure that references a term before it is declared
      2. Procedure with duplicate variable name
      3. Procedure with actions with invalid functors
      4. Procedure with action outputs that are unused
      5. Procedure with procedure inputs that are unused
      6. <Flesh out the different validation errors from Lumen>
2. Mutations

The following is a list of procedure mutations that are supported by the editor. It is not a list of test cases but rather a list of operations from which both automated and manual test cases should be derived. There may be multiple ways to invoke the following mutations in the UI but that are not broken out here. Note that the list below does not include “application level” mutations such as import/export, save a copy, rename, etc.

* 1. Procedure-Level Mutations
     1. Change procedure name
     2. Edit description
     3. Configure procedure inputs (via terms)
     4. Change the order of procedure inputs
     5. Configure procedure outputs
     6. Remove unused procedure inputs
  2. Step-Level Mutations
     1. Add a step
        1. Add a procedure step
        2. Add a step from another namespace
        3. Add a loop step
     2. Move a step
        1. Move in to a loop
        2. Move out of a loop
        3. Move a loop
     3. Delete steps
        1. Delete from within a loop
        2. Delete a loop
     4. Configure a loop
        1. Change the collection being iterated over (currently buggy)
  3. Term-Level Mutations
     1. Rename variables
     2. Change constant values
     3. Replace variable with constant
     4. Replace constant with existing variable
     5. Create new procedure input
     6. Modify default values for procedure inputs
     7. Replace function calls with term of proper type
  4. Supported validation errors/warnings
     1. Unsupported variable
     2. Unused procedure input
     3. Unused action output

1. Automated Tests

The following is a list of integration and unit test ideas for the PAL UI. These should be automated using JUnit.

* 1. Integration Tests
     1. Serialization/deserialization round-trip for all test procedures using the Core UI ATR implementation. Assert equivalence of serialized CTR-S at begin and end. (Does this apply to invalid procedures?)
     2. Execute each valid canned procedure using the Core UI ATR implementation and assert that Arda performed the actions as expected by string comparison
     3. Export/Import round-trip test for each canned procedure
     4. Mutations Tests
        1. For each mutation listed in section 4, load an appropriate procedure and use the edit controller or other appropriate mechanism to cause that mutation in the models. Assert necessary changes occurred in models, save the procedure, assert change was persisted appropriately/. …. Assert the presence of necessary Undo operations.
     5. Validate each of the Invalid canned procedures and assert the correct number and nature of errors and warnings.
        1. Assert that UI suggests the appropriate number and nature of repair suggestions.
        2. Assert that applying each repair suggestion has the expected result.
     6. Equals implementation tests:
        1. Create two copies of each canned test procedure and verify that equals() is true at each level of the parse tree
        2. Create two copies of each canned test procedure. On one of the copies, change the leaf term values and verify that equals is false between the two copies at all levels of the parse tree.
     7. Test suggested replacement terms (existing values) from lumen/VariableManager (could be part of VariableManager unit test).
  2. Unit Tests (descending priority)
     1. StorageAssistant/ActionModelAssistant
     2. VariableManager in conjunction with VariableModel.create
     3. SelectionManager
     4. UndoManager

1. Functional Tests

The functional tests for the editor are defined in the associated document “PAL UI Editor Functional Tests”. The test cases are organized in to five primary categories:

1. Application-Level Operations
2. Procedure-Level Operations
3. Step-Level Operations
4. Term-Level Operations
5. Miscellaneous

Each category is further broken out by pieces of related functionality. List items of the form “7.1.1.1” or “7.4.12.4” represent the actual test cases that should be tracked in a spreadsheet when executing a test plan for a given release. The bullet lists below a specific test case contain the assertions, expected behavior and things to check as part of that test case.

Note that for a specific release, it may not be necessary for all functional test cases to pass. This determination must be made based on the requirements of the release, the action models involved and development priorities. Some test cases already have corresponding issues in JIRA. The functional test cases as defined in the document reflect the ideal state of the system.

1. General Rules for Manual Testing

The following is a list of rules that should be kept in mind while performing manual UI testing. Instances where the UI does not adhere to these rules will usually constitute test failure.

* 1. Any exceptions or errors seen in the log file our application output should be considered bugs even if they do not appear to have a negative impact on program execution.
  2. Any possible action in the UI that results in a change that will be persisted upon saving must result in the creation of an appropriate undo step that is both undoable and redoable.
  3. The only way the user will be allowed to introduce errors in to a procedure is by moving or deleting steps. In all other instances the UI should prevent the user from invalidating the procedure. For example, while a user can delete a step whose output is referenced in subsequent steps, they should never be allowed to replace a term with a term of incorrect type.
  4. Any popup window, modal dialog or dialog step should lock all other application windows and UI components that are not participating in the dialog. Disabled components should appear disabled.
  5. Any errors generated by the bridge, lumen or any other underlying component should have an entry in the log file.
  6. All panes with dynamic content should support scrolling to view all content. Vertical scrolling should always be possible via the mouse wheel.
  7. All buttons should have a tooltip
  8. It should be possible to maximize, minimize and restore the application to its original size without causing display issues. Minimum supported application window size for the editor is 1000 x 500.

1. Performance Tests
   1. Application Speed
      1. Start-up should take less than 10 seconds (depends on # of types loaded)
      2. Performance should not degrade when there are more than 100 steps in a procedure
      3. Performance should not degrade when there are multiple levels of loop nesting
      4. Performance should not degrade when there are more than 100 procedure inputs.
   2. Memory Consumption
      1. Using a heap analysis tool, we should verify then when a user opens subsequent procedures that the memory for the previous ProcedureModel/ProcedureView is not leaked to insure that users can open many procedures in a single session.