# ASLANTEST TESTPLAN
A Testplan for the
Symbolic Execution Tool
ASLANTEST

by

Jeffrey G. Douglas

UNIVERSITY OF CALIFORNIA
Santa Barbara

June 1993

# Contents

# List of Figures

# Chapter 1

# Introduction

# 1 Introduction

This is the system test plan for the symbolic execution tool Aslantest. At a high level, Aslantest consists of three distinct modules: the translator, the executor, and the graphical user interface. This test plan describes the testing approach used in testing the translator and the executor modules which make up the Aslantest tool.

The approach presented by this test plan is to integrate the two modules together after testing each of them separately. Detailed approaches to unit, integration, system, and regression testing are presented. Where appropriate, specific unit, integration, and system test cases are provided.

There are five sections to this document. The next section outlines the testing of each main module. Section three outlines the approach to integrating the three main modules. Section four explains the approach to system testing. Finally, section five outlines an approach to regression testing in the event of changes or enhancements to the system.

# Chapter 2

# Unit Testing

# 2 Unit Testing

This chapter outlines the approach to testing the two mandatory components of Aslantest: the translator and the executor.

## 2.1 The Translator

The translator is the simplest of the three main modules. The translator consists of two parts: a lex lexical analyzer and a yacc parser. Testing of this module consists of testing the lexical analyzer and then testing the lexical analyzer conjoined to the parser.

The translator serves two roles. First, it translates Aslan specifications producing a series of executor function calls that the executor uses to animate the specification. It is also called to translate user-provided Aslan predicates during the execution of a specification. In this role, the translator merely translates the predicate to a pre-order form and returns it to the executor.

## 2.2 Testing of The Translator

Testing of the translator is done in two phases. First, the lexical analyzer is tested in isolation. Then, the translator is tested with the lexical analyzer integrated with the parser. Testfiles containing specifications or predicates are used for testing in both phases. Results may be compared with corresponding result files to determine if the test was successful.

To set up the translator for testing, two executables must be created. This is done by typing "make testtool" in the source directory. The result of this will be two executables, entitled ldebug and ydebug. Ldebug is an executable of the isolated lexical analyzer that prints out messages corresponding to the constructs it recognizes. Ydebug is the complete translator compiled in a debug mode. In addition to echoing the translated form of its input, it also produces messages explaining what has been translated.

## 2.3 The Test Cases

The following are test cases for the translator module. To run a test, the specification or predicate should be placed in a file as described in the test file description. Then the expected results of the test case should be placed in a result file. The test file can then be run on the translator and compared with the result file. It is helpful to remember that the lexical analyzer's only responsibility is to recognize all keywords, key symbols, and identifiers in the given specification or predicate and send them to the parser. The lexical anaylzer makes no attempt to differentiate between a specification or a predicate; it simply recognizes keywords. Therefore, the only errors caught at the lexical analyzer level are unrecognized characters or illegal character combinations.

---

```
********** TEST #1 - Basic Spec ******************
DESCRIPTION OF TEST FILE:
   A basic specification. Tests various type forms, integer and
   boolean operations.
```

---

```
********** TEST #2 - Set Spec ******************
DESCRIPTION OF TEST FILE:
   Tests different representations of sets.
```

---

```
********** TEST #3 - List Spec ******************
DESCRIPTION OF TEST FILE:
   Tests different representations of lists.
```

---

```
********** TEST #4 - Quantified Spec ******************
DESCRIPTION OF TEST FILE:
   Tests various combinations of quantified expressions.
```

---

```
********** TEST #5 - Bound Spec ******************
DESCRIPTION OF TEST FILE:
   A specification containing an Aslantest representation of
   a bound variable. Should be caught as an error by parser
   but not by lexical analyzer.
```

```
********** TEST #6 - Reduces_To Spec *****************
DESCRIPTION OF TEST FILE:
   A specification containing an Aslantest ‘reduces_to’
   operator. Should be caught as an error by parser
   but not by lexical analyzer.
```

---

```
********** TEST #7 - Misplaced Spec *****************
DESCRIPTION OF TEST FILE:
   File begins with keyword ’symbolic_exec_call’ which
   indicates the file contains an aslantest predicate.
   Cannot enter entire specifications in this mode.
   Parser should catch error.
```

---

```
********** TEST #8 - Bound variable Pred *****************
DESCRIPTION OF TEST FILE:
   A user-entered predicate containing bound
   variable references.
```

---

```
********** TEST #9 - Reduces_to Pred *****************
DESCRIPTION OF TEST FILE:
   A user-entered predicate containing reduces_to operators.
```

---

```
********** TEST #10 - Parens tests *****************
DESCRIPTION OF TEST FILE:
   A predicate containing extra parentheses. The extra parens
   should not be reflected in the resulting translation.
```

---

```
********** TEST #11 - Precedence tests *****************
DESCRIPTION OF TEST FILE:
   Predicates should use parentheses to manipulate
   precedence. The translated form should reflect
   correct precedence.
```

```
********** TEST #12 - Misplaced Preds *****************
DESCRIPTION OF TEST FILE:
   A user-entered predicate which is not preceded by
   the keyword 'symbolic_exec_call'. Error that should
   be caught by the parser.
```

---

```
********** TEST #13 - ALT Spec *****************
DESCRIPTION OF TEST FILE:
   A specification containing ALT operators. Note that the
   parser performs the translation of this procedural
   operator to its non-procedural disjunctive form.
```

---

```
********** TEST #14 - Parameterized Variables *****************
DESCRIPTION OF TEST FILE:
   A specification containing parameterized
   variables in various forms.
```

---

```
********** TEST #15 - Structures *****************
DESCRIPTION OF TEST FILE:
   A specification focusing on structured types.
```

---

```
********** TEST #16 - Illegal Characters *****************
DESCRIPTION OF TEST FILE:
   Specifications and predicates containing
   illegal characters and constructs. Illegal
   characters should be caught by the lexical
   analyzer. Misspelled keywords or illegal
   constructs should be caught by the parser.
```

## 2.4 The Executor

The executor is a large module consisting of a series of sub-modules, each of which contain a collection of lisp functions. Figure 2.1 shows the breakdown of the executor into sub-modules. There are six more complex modules, each of which interact with a a collection of library routines. For testing purposes, the library routines can be broken down into four simple modules. In unit testing each of the ten modules, each function which is called from outside the module is tested. If a function uses other functions internal to the module to complete its task, then that function is tested in a bottom-up fashion. First the internal functions are tested, then the external function is integrated with the internal functions and tested. If a function being tested makes a function call outside of its module, a stub is used to model the function call; integration between modules is saved for later in the testing process.

## 2.5 The Sub-Modules

The following sections outline the testing of each sub-module. Each test number indicates the testing of a particular function. Test numbers like 1.1, 1.2, etc. indicate the testing of an internal function which is called from an external function.

The testing of each function is described using a test template. The test template contains the following fields:

- *Test Function Name:* The name of the function being tested.
- *Called By:* Modules that call this function. The keyword 'Internal' indicates that the function is not called outside the module.
- *Purpose:* A one-line description of the function's purpose.
- *Internal Functions Called:* A list of functions within the module called by the function being tested. the testing of each of these functions is described before the test description of higher level function.
- *Function Description:* A more detailed description of the role of the function.
- *Driver:* A description of the driver function used to test the function.

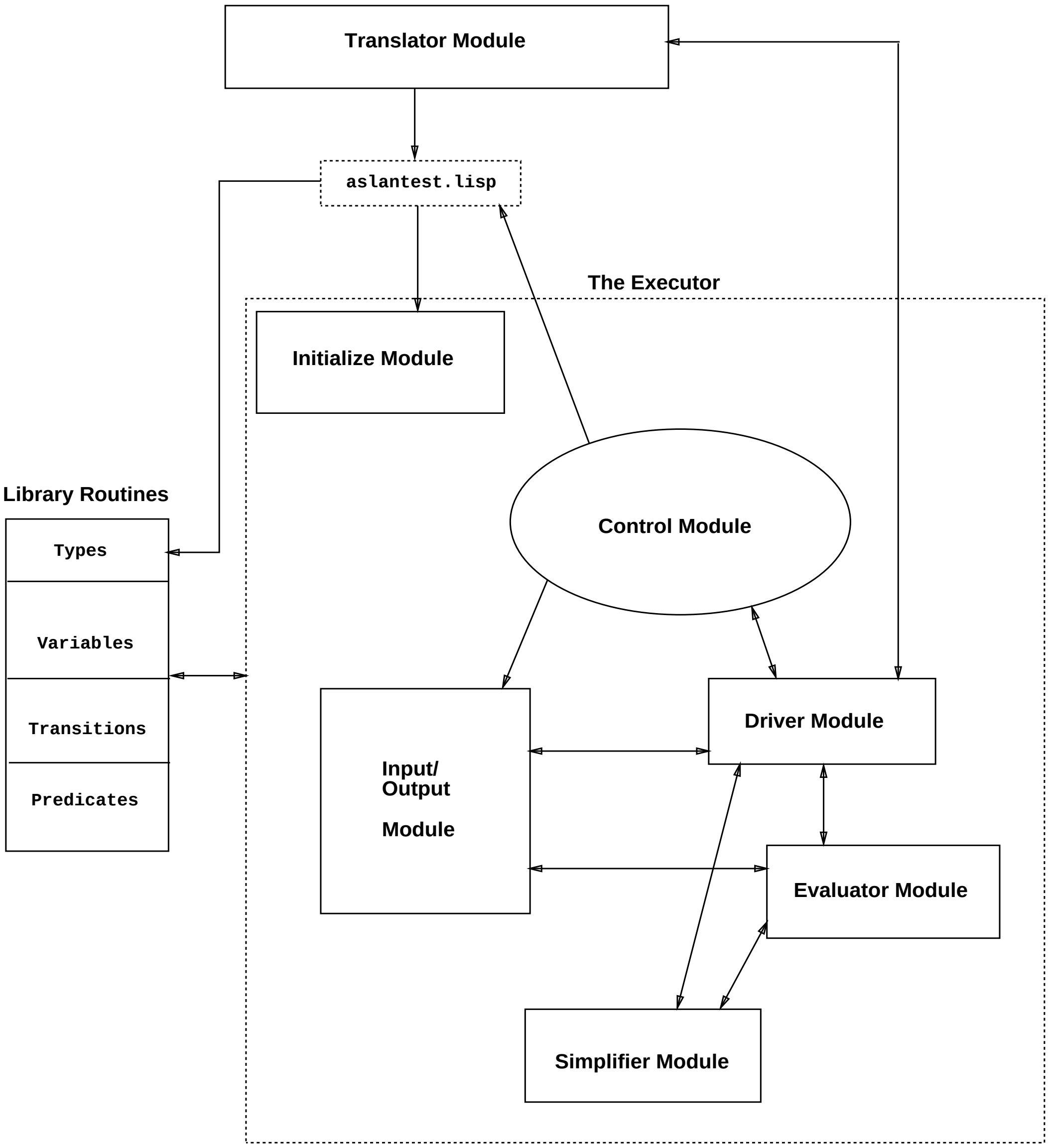


Figure 2.1 Aslantest Modular Layout

- *Stubs:* Any stubs needed to test this function are described here. Recall that stubs are used in place of calls to functions outside the module.
- *Test Cases:* Describes various test cases that should be run on the function.
- *Notes:* Provides notes on the function and on its test cases. Also lists other functions that are similar in design and role and thus can be tested in a similar fashion.

The function add_variable() below is a good example. It is called by the file generated from the translator to add a state variable to the current list of state variables. It calls one internal function, set_params(). Set_params() and its test cases are described in function 1.1. Then the remainder of the test template for add_variable() is displayed.

The use of the lisp programming environment makes isolation and testing of these functions quite simple. In fact, automation of the testing process for the executor module is possible. The Aslantest test suite contains an example file of test cases for the variables module. The file var_test.lsp contains test cases for six functions in the variables module. The file consists of a single function which is called with a single argument that tells the function which test cases to run. Other files could be developed for the other modules and a driver could be written that allows users to select tests to run.

### 2.5.1 The Variables Module

The variables module is part of the library routines. It consists of 16 external functions, some of which invoke other internal functions.

---

```
********** FUNCTION 1: add_variable() *****************
TEST FUNCTION NAME: add_variable
CALLED BY: aslantest.lisp initialization file
PURPOSE: add a state variable to the current list.
INTERNAL FUNCTIONS CALLED: set_params()

**** FUNCTION 1.1 - set_params() ****
```

```
FUNCTION DESCRIPTION:
   Given a variable id(id) and a list of parameter
   types(ptypes), produces a list of parameter ids
   and stores them in the attribute ‘params’.
DRIVER:
   Calls set_params() with a variable id and a
   list of param types.
TEST CASES:
   1. A single parameter variable is given as an argument.
      Should generate a single item parameter list.
   2. A two-parameter id is given. Should generate a two-item
      parameter list.
   3. A non-parameterized id is given. Should generate
      a null list.

**** FUNCTION 1 - add_variable() ****

FUNCTION DESCRIPTION:
   Given a variable id and a list of parameter types, correctly
   add the variable to the list, and initialize the following
   attributes: params and ptypes.
DRIVER:
   Initializes value of tmp_var_list and calls add_variable()
   with a variable id(var_id) and a list of param types(ptypes).
TEST CASES:
   Set tmp_var_list to (b, c), and call add_variable with the
   arguments given in each case of Function 1.1. Results
   should be:
      tmp_var_list = (a, b, c)
      attribute ptypes of a = type argument
      attribute params of a = as stated above.
NOTES:
   The same kind of tests should be run on the following
   functions:
   add_constants(), add_const(), add_tmp_const(),
   add_defined_var(), and add_internal_trans().
```

---

```
********** FUNCTION 2: set_var_type() *****************
```

```
TEST FUNCTION NAME: set_var_type
CALLED BY: aslantest.lisp initialization file
PURPOSE: Sets the type of all variables in the tmp_var_list
   system variable; xfers them to permanent variable list.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a type id, assigns all variable ids currently in
   tmp_var_list to that type, adds them to var_list, and
   reinitializes tmp_var_list to nil.
DRIVER:
   Calls set_var_type() with a type id(type_id).
TEST CASES:
   1. Call function with a type, and tmp_var_list containing
      some ids. Should assign each the given type, xfer them
      to permanent var list, and set tmp_var_list to nil.
```

---

```
********** FUNCTION 3: is_var() *****************
TEST FUNCTION NAME: is_var
CALLED BY: predicate and transitions modules
PURPOSE: Given an id, returns true if it is in the
   current variable list.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a id(id), return true if it is in var_list; else
   return nil.
DRIVER:
   Initializes var_list; calls is_var() with an id(id).
TEST CASES:
   1. id is not a declared variable. Should return nil.
   2. id is a declared variable. Should return true.
   3. var_list is nil. Should return nil.
NOTES:
   This same type of test should be run on the following
   functions: is_def(), is_int_trans(), is_variable(),
   and is_const().
```

---

```
********** FUNCTION 4: get_var() *****************
TEST FUNCTION NAME: get_var
CALLED BY: evaluator and predicate modules.
PURPOSE: Retrieves specified attribute values.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a variable id and an attribute do one of the following:
     - if the variable is a define or variable id and the
       attribute is a valid one, return the value;
     - if the attribute is not valid, return nil;
     - if the variable id is not found, respond with an
       error msg.
DRIVER:
   Initializes var_list, define_list, and intern_transitions
   list.
   Initializes attributes relevant to test.
   Calls get_var() with a variable id(var_id) and an
   attribute(prop).
TEST CASES:
   1. Var id and attr. given are valid; should return the value
      associated with the attribute for that variable.
   2. Var id is not a declared variable; should respond with
      an error message.
   3. Attribute provided is not valid; should return nil.
NOTES:
   Same kind of tests should be run using define ids. Also,
   same kind of tests should be run on the following functions,
   which are simply extensions of get_var(): get_value(),
   get_new_value(), get_variable_args(), get_variable_ptypes(),
   get_explicit_values(), get_definition(), get_params(),
   get_type(), and get_genid().
```

---

```
********** FUNCTION 5: set_var() *****************
TEST FUNCTION NAME: set_var
CALLED BY: predicate, control, and evaluator modules
PURPOSE: Sets specified id attribute to specified value.
```

```
INTERNAL FUNCTIONS CALLED: is_var()

FUNCTION DESCRIPTION:
   If the given id is a state variable, set the given
   attribute for that variable to the given value.
DRIVER:
   Initializes var_list; calls set_var() with an id(var_id),
   value(value), and attribute(prop).
TEST CASES:
   1. Variable is a declared variable. Should assign attribute
      of variable to value.
   2. Variable is undeclared; respond with an error message.
NOTES:
   This same type of test should be run on the following
   functions, which are simply extensions of set_var():
   set_value(), set_new_value(), set_type(), set_params().
```

---

```
********** FUNCTION 6: variable_value() *****************
TEST FUNCTION NAME: variable_value
CALLED BY: predicate module
PURPOSE: Returns the current value of a state variable.
INTERNAL FUNCTIONS CALLED: get_value(), get_explicit_value(),
   get_list_value(), get_struct_value().

**** FUNCTION 6.1 - get_list_value() ****

FUNCTION DESCRIPTION:
   Given a list value structure and an index, return the
   indexed value if it exists. Otherwise, return nil. If
   the index is not an integer value, return the list
   indicating the non-integer reference.
DRIVER:
   Calls get_list_value() with a list(l) and an
   index(index).
TEST CASES:
   1. Index is within the range of the given list. Function
      should return the value corresponding to that index.
   2. List is empty. Return nil.
```

```
   3. Index is a non-numeric value. Return a reference
      structure using the list and index.
   4. Index is beyond the range of the given list. Return nil.

**** FUNCTION 6.2 - get_struct_value() ****

FUNCTION DESCRIPTION:
   Given a struct value structure and a field, return the
   field value if it exists. Otherwise, return nil.
DRIVER:
   Calls get_struct_value() with a structure(s) and
   a field(field).
TEST CASES:
   1. Field is a valid defined field in the structure. Return
      the value associated with that field.
   2. Field is not defined in the structure. Return nil.
   3. Structure value is nil. Return nil.

**** TEST 6.3 - get_explicit_value() ****

FUNCTION DESCRIPTION:
   Given a variable id and a list of actual params, retrieve the
   value associated with the actual params. If there is more than
   one such value, return a conditional expression as the value.
   If no value is found or the params are actually referencing a
   structure or list(indicated by the C_DOT keyword) return nil.
DRIVER:
   Initializes a variable and its relevant attributes.
   Calls get_explicit_value() with an id(id) and a
   param list(params).
STUBS:
   sub_params2() returns the expression as given to it.
   Simplify() acts as stated in each test case.
EXPECTED RESULT:
   Described above.
TEST CASES:
   1. Given parameter has a value in the list of values. Return
      a conditional with the value and the default value.
   2. Given parameter is not in the list. Return the default
      value.
```

```
   3. Given parameter is a list/struct reference. Return nil.

**** FUNCTION 6 - variable_value() ****

FUNCTION DESCRIPTION:
   Given a variable name and a possibly empty set of parameters,
   return its current value.
   If the name argument is not a variable, respond with an
   error message.
   If params is nil, return value of the 'value' attribute.
   If params contain bound vars, return original reference.
   Otherwise, use get_explicit_value(), get_list_value(), or
   get_struct_value() to determine value.
DRIVER:
   Initializes relevant variable attributes.
   Calls variable_value() with a variable id(name)
   and set of params(params).
STUBS:
   sub_params2() simply returns its argument;
   is-bound() returns true if the expression contains the
   keyword BOUND;
   If not mentioned, Simplify() returns true.
TEST CASES:
   Cases 1 - 3: Run 6.3 tests 1, 2, and 3. Expected results of
      cases 1 and 2 are the same. Expected result of case 3 is
      the original reference returned.
   4. Retrieve the value of a non-parameterized variable.
      Should return value.
   5. Attempt to retrieve the value of an undefined variable.
      Respond with an error message.
   6. Retrieve a value with a list variable using a valid index.
      Return the value corresponding to that index.
   7. Retrieve a value within a struct variable using a valid
      field. Return the value associate with the field.
   8. Retrieve a value from a parameterized variable using a
      bound parameter argument. Should return the original ref.
NOTES:
   Tests 6 and 7 can be run with the test cases used in tests
   6.1 and 6.2.
```

```
********** FUNCTION 7: make_value() *****************
TEST FUNCTION NAME: make_value
CALLED BY: predicate and evaluator modules
PURPOSE: Constructs a symbolic value representation.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a variable id,  construct the representation
   of its most current symbolic value.
DRIVER:
   Initializes a variable and its genid attribute.
   Calls make_value() with an id(id).
TEST CASES:
   1. Given id is in the variable list. Should return
      the symbolic value representation using the current
      value of the genid attribute.
```

---

```
********** FUNCTION 8: get_next_value() *****************
TEST FUNCTION NAME: get_next_value
CALLED BY: evaluator module
PURPOSE: Returns representation of next symbolic value.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a variable id,  construct the representation
   of its next symbolic value.
   If the new_genid attribute of the variable is nil, it is
   set to the value of genid plus 1. Then the representation
   using the value of new_genid is returned.
DRIVER:
   Initializes a variable and its genid attribute.
   Calls get_next_value() with an id (name).
TEST CASES:
   1. Variable id is declared, and its new_genid attribute is
      nil. New_genid should be set to one greater than genid,
      and the symbolic representation returned using new_genid.
```

2. Variable id is declared, and its new_genid attribute is a numeric value. Return representation with new_genid.

---

```
********** FUNCTION 9: constant_value() *****************
TEST FUNCTION NAME: constant_value
CALLED BY: simplify and predicate modules
PURPOSE: Returns value of the specified constant.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   If a parameter list is provided, the id is returned
   with a reference to the given parameters. Otherwise,
   the id is returned.
DRIVER:
   Calls constant_value() with an id(id) and a
   parameter list(params).
TEST CASES:
   1. Parameter list is nil. Return id.
   2. Parameter list is not nil. Return reference structure
      with id and param list.
NOTES:
   const_type() is another simple function that returns the
   type of the given constant. It can be
   tested in a similar fashion.
```

---

```
********** FUNCTION 10: fix_int_trans() *****************
TEST FUNCTION NAME: fix_int_trans
CALLED BY: transitions module
PURPOSE: Expands define bodies.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a define id, stores expanded version in attribute
   def.
DRIVER:
   Calls fix_int_trans() with a define id(tr).
STUBS:
```

```
   expand_wff(), fix_BOUND(), expand_NOCHANGE() always return
   the keyword NOT_CHANGED.
TEST CASES:
   1. After execution, def attr. of tr should be NOT_CHANGED.
```

---

```
********** FUNCTION 11: init_value() *****************
TEST FUNCTION NAME: init_value
CALLED BY: evaluator module.
PURPOSE: Sets a variable's value to a symbolic value.
INTERNAL FUNCTIONS CALLED: set_var(), make_value().

FUNCTION DESCRIPTION:
   Sets a variable's value or explicit_values attribute to
   its most recent symbolic value.
   If the id has params, set the explicit_values attribute
   to the current symbolic value (according to the genid
   attribute) for all instances.
   If the id has no params, set the value attribute to the
   current symbolic value.
DRIVER:
   Initializes a variable and its relevant attributes.
   Calls init_value() with an id(id).
TEST CASES:
   1. Non-parameterized id is in variable list. Should increase
      genid attr. of id by one, and set value attr. of id
      to symbolic representation using genid.
   2. Parameterized id is in variable list. Should increase
      genid attr. of id by one, and set explicit_values attr.
      of id to symbolic representation using genid.
NOTES:
   init_state_vars() simply calls init_value() for each
   variable currently in var_list. It can be tested in
   a similar manner.
   Also, implicit_update() simply invokes init_value().
```

---

```
********** FUNCTION 12: flush_attributes() *****************
TEST FUNCTION NAME: flush_attributes
```

```
CALLED BY: control module
PURPOSE: Effectively empties the variable and constant
   lists upon reinitialization of the system.
INTERNAL FUNCTIONS CALLED: None.

FUNCTION DESCRIPTION:
   Takes as an argument a flag. If the flag is set to true,
   indicates a total flush, and the type field of all
   variables, constants, and temp. constants is set to nil.
   If flag set to nil, only temp. constants are reset.
DRIVER:
   Initializes variable and constant lists.
   Calls flush_attributes(), with flag argument(total).
TEST CASES:
   1. Call function with total = true. All variables and
      constants should have nil as type attribute value.
   2. Call function with total = nil. Temporary constants
      type attribute set to nil; all variables and
      specification constants retain type attribute values.
```

---

```
********** FUNCTION 13: update_value() *****************
TEST FUNCTION NAME: update_value
CALLED BY: evaluator module
PURPOSE: Performs the update of state variables during
   a state change.
INTERNAL FUNCTIONS CALLED:
   add_mem(), update_list_value(), update_struct_value(),
   make_free_constraint(), set_explicit(), flatten(),
   make_bound_constraint(), sub_old_bound_values(),
   has_Cur_Ref().

**** FUNCTION 13.1 - update_list_value() ****

FUNCTION DESCRIPTION:
   Given a list and an index, if the position indicated
   by the index exists within the list, replace the
   old value with the value given as an argument. If not,
   return the previous list.
```

```
DRIVER:
   Calls update_list_value() with the list(curval),
   the index(index), and a value(val).
STUBS:
   Simplify() - for all test cases, echos its argument
   as its return value.
TEST CASES:
   1. Index is within range of list. Return list with
      value is position indicated by index.
   2. Index is beyond range of list. Return list.
   3. List is nil. Return nil.

**** FUNCTION 13.2 - update_struct_value() ****

FUNCTION DESCRIPTION:
   Given a structure, a field, and a value, if the position
   indicated by field exists within the struct, replace
   the old value with the value given as an argument. If
   not, create that field within the given struct.
DRIVER:
   Calls update_struct_value() with a struct(curval),
   a field(field), and a value(val).
STUBS:
   Simplify() - for all test cases, echos its argument
   as its return value.
TEST CASES:
   1. Field is defined in struct. Replace current
      field value with val.
   2. Field is not defined in struct. Insert field
      with val into struct.
   3. Struct is nil. Create struct with field and
      val as only entry.

**** FUNCTION 13.3 - add_mem() ****

FUNCTION DESCRIPTION:
   Adds an item to a set of items as a result of a
   Membership operator.
   If the new_value of the variable is null, assign
   a new symbolic value to the variable and form the
```

```
   union of the symbolic value and the expression.
   If the new_value is not null, union the expression
   with the current value of the new_value attribute.
DRIVER:
   Initializes variable value. Calls add_mem() with
   variable name(name) and set expression(exp).
STUBS:
   Simplify() - for all test cases, echos its argument
   as its return value.
TEST CASES:
   1. New_value of variable name is nil. Set new_value
      of variable to union of symbolic value and the
      given expression.
   2. New_value of variable name is a set. Reset
      new_value to be union of current set and exp.

**** FUNCTION 13.4 - make_free_constraint() ****

FUNCTION DESCRIPTION:
   Constructs a conjunctive statement where each conjunct
   asserts the equality of a parameter with its
   corresponding actual.
DRIVER:
   Initializes variable attributes. Calls make_free_constraint()
   with an id(id) and its actual arguments(args).
STUBS:
   Simplify() - for all test cases, echos its argument
   as its return value.
TEST CASES:
   1. Variable id is a single parameterized variable. Return
      single equality predicate.
   2. Variable id is a two-parameter variable. Return
      conjunction of two equality predicates.

**** FUNCTION 13.5 - has_Cur_Ref() ****

FUNCTION DESCRIPTION:
   Given an expression id = expr, returns true if expr contains
   a reference to id's current value.
DRIVER:
```

Initializes global variable Cur_Ref, and calls has_Cur_Ref() with an expression(e).

TEST CASES:

1. Argument e contains a reference to the expression contained in Cur_Ref. Return true.
2. Argument e does not contain a reference to Cur_Ref. Return nil.

**** FUNCTION 13.6 - sub_old_bound_values() ****

FUNCTION DESCRIPTION:

Replaces occurrences of a current value reference as described above by their current value, which is stored in the global variable Cur_Val.

DRIVER:

Initializes Cur_Ref and Cur_Val. Calls sub_old_bound_values() with an expression(e).

TEST CASES:

1. Argument e contains a reference to Cur_Ref. Return e with references to Cur_Ref replaced by Cur_Val.
2. Argument e does not contain a reference to Cur_Ref. Return e unchanged.

**** FUNCTION 13.7 - flatten() ****

FUNCTION DESCRIPTION:

Given a list of lists, constructs a single list, using sub_old_bound_values to replace current variable references where ever it can.

DRIVER:

Initializes Cur_Ref and Cur_Val. Calls flatten() with a list of lists(c_stack).

TEST CASES:

1. C_stack is a list of lists and doesn't contain any references to Cur_Ref. Return a single list.
2. C_stack is a list of lists and does contain ref. to Cur_Ref. Return a single list with all references replaced by Cur_Val.

**** FUNCTION 13.8 - make_bound_constraint() ****

```
FUNCTION DESCRIPTION:
   Constructs a single bound constraint from a global
   list of constraints and a constraint provided as
   an argument by conjoining them together.
DRIVER:
   Initializes Cur_Ref, Cur_Val, bound_constraints. Calls
   make_bound_constraint() with an assertion(c), a list of
   actual parameters(a), and a list of formal parameters(f).
STUBS:
   sub_params2() : echos its third argument as a return value.
TEST CASES:
   1. Bound_constraints don't contain any references to Cur_Ref.
      Return a conjunction of flattened bound_constraints and
      argument c.
   2. Bound_constraints contain references to Cur_Ref. Return
      a conjunction of flattened bound_constraints and argument
      c, with all references replaced by Cur_Val.

**** FUNCTION 13.9 - set_explicit() ****

FUNCTION DESCRIPTION:
   Given a list of explicit values for a parameterized variable
   a field and an index, update the list.
   If field is nil, then this is an update of a parameterized
   variable. Look for the index in the list of values. If
   found, replace the current value with val. If not found,
   conjoin to the end of the list.
DRIVER:
   Calls set_explicit() with the following arguments:
   a list of explicit values(e_vals), an index(index),
   a field(field), a value(val), and the name of the
   variable to which this pertains(name).
TEST CASES:
   1. INITIALIZED VALUES:
         var_list = (a)
         params attr. of a = (_p1)
      FUNCTION ARGUMENTS:
         id = a
         args = (2)
```

```
      EXPECTED RESULTS:
         return (.EQ _p1 2)
   2. INITIALIZED VALUES:
         var_list = (a)
         params attr. of a = (_p1 _p2)
      FUNCTION ARGUMENTS:
         id = a
         args = (4 2)
      EXPECTED RESULTS:
         return (.AND (.EQ _p1 4)(.EQ _p2 2))

**** FUNCTION 13 - update_value() ****

FUNCTION DESCRIPTION:
   Updates state variables during a change of state.
DRIVER:

STUBS:
   Simplify() - for all test cases, echos its argument
   as its return value.
   sub_params2() - returns its third argument as a value.
TEST CASES:
   1. INITIALIZED VALUES:
         var_list = (a)
         params attr. of a = (_p1)
      FUNCTION ARGUMENTS:
         id = a
         args = (2)
      EXPECTED RESULTS:
         return (.EQ _p1 2)
   2. INITIALIZED VALUES:
         var_list = (a)
         params attr. of a = (_p1 _p2)
      FUNCTION ARGUMENTS:
         id = a
         args = (4 2)
      EXPECTED RESULTS:
         return (.AND (.EQ _p1 4)(.EQ _p2 2))
```

### 2.5.2 The Types Module

The types module is part of the library routines. The following are its test cases.

---

```
********** FUNCTION 1: add_type() ******************
TEST FUNCTION NAME: add_type
CALLED BY: aslantest.lisp initialization file.
PURPOSE: Adds a type id and its type to the type list.
INTERNAL FUNCTIONS CALLED: None.

FUNCTION DESCRIPTION:
   Takes as an argument a type id and its type. If the
   type is nil, then it is an unspecified type. If the
   type is enumerated, each of the ids in the enumerated
   list is added to the constant list and given a numeric
   value consistent with their ordering.
DRIVER:
   Calls add_type() with a type id(type_id) and type(type).
STUBS:
   add_const() returns nil.
TEST CASES:
   1. type is nil. Type_id should be added to type_list, and
      its type attribute should be nil.
   2. type is enumerated. Type_id should be added to type_list,
      its type attr. should be equal to type, and each
      constant in the enumeration should be added to the
      constant list.
```

---

```
********** FUNCTION 2: check_const() ******************
TEST FUNCTION NAME: check_const
CALLED BY: transitions module
PURPOSE: Checks a constant given as a parameter to a
         transition for type correctness.
INTERNAL FUNCTIONS CALLED: implicit_decl(), type_mismatch().

**** FUNCTION 2.1 - implicit_decl() ****
```

```
FUNCTION DESCRIPTION:
   Declares a new constant.
DRIVER:
   Calls implicit_decl() with a constant name(actual) and
   a type(formal-type).
TEST CASES:
   1. Call function with a name and type. Should add constant
      to temporary list, and produce msg announcing new const.

**** FUNCTION 2.2 - type_mismatch() ****

FUNCTION DESCRIPTION:
   Produces error message announcing type conflict.
DRIVER:
   Calls type_mismatch() with a transition name(name), a
   constant id(actual), and a type(formal-type).
TEST CASES:
   1. Transition name is nil. Message should only mention
      type error.
   2. Transition name is given. Msg should include
      transition name.

**** FUNCTION 2 - check_const() ****

FUNCTION DESCRIPTION:
   Takes as arguments a transition name, an undefined constant,
   and a formal parameter. It either declares the constant,
   placing it in the constant list upon determining its type,
   or determines a type mismatch error. Type mismatch errors
   occur with bound variables or enumerated types.
DRIVER:
   Calls check_const() with a transition name(name), a
   constant(actual), and a formal-type(formal-type).
TEST CASES:
   1. Formal-type is integer. Should declare constant of type
      integer.
   2. Formal-type is boolean. Should declare constant of type
      boolean.
   3. Constant is bound and its type is consistent with formal-
```

type. Return true.

4. Constant is bound and its type is not consistent with formal-type. Type mismatch error.
5. Formal-type is enumerated, and constant is a valid member. Return true.
6. Formal-type is enumerated, and constant is not a valid member. Type mismatch error.

NOTES:

Sets, lists, typedefs, and subtypes.

---

********** FUNCTION 3: int_tst() ******************

TEST FUNCTION NAME: int_tst

CALLED BY: transitions module

PURPOSE: Checks an integer to ensure correct type.

INTERNAL FUNCTIONS CALLED: type_mismatch(){Function 2.2}

FUNCTION DESCRIPTION:

Takes as arguments a transition name(name), an integer (actual), and a formal type(formal-type). Ensures that integers are valid instances of the given type. Checks subtypes, typedefs.

DRIVER:

Calls int_tst() with the required parameters.

STUBS:

eval_condition() responds as described in each test.

TEST CASES:

1. Formal-type is integer. Return true.
2. Formal-type is boolean. Type mismatch error.
3. Formal-type is a subtype of integers and the integer is within the range. Return true.
4. Formal-type is a subtype of integers and the integer is not within the range. Type mismatch error.
5. Formal-type is any non-numeric type. Respond with error-message announcing error.

NOTES:

Test 5 should be run with enumerated, unspecified, sets, lists and structs.

Bool_tst() and const_tst() are simple boolean and constant checking functions that can be tested in the same way.

```
********** FUNCTION 4: find_name() *****************
TEST FUNCTION NAME: find_name()
CALLED BY: transitions module
PURPOSE: Returns a variable id from within a ref. structure.
INTERNAL FUNCTIONS CALLED: None.

FUNCTION DESCRIPTION:
   Takes as an argument an expression which is a variable
   reference. Parses the expression to return the id of
   the variable.
DRIVER:
   Calls find_name() with a reference expression(e).
TEST CASES:
   1. Call Function with a single id. Return the id.
   2. Call function with an id within a reference
      structure. Return the id.
   3. Call function with an id within a NEW structure.
      Return the id.
NOTES:
   Tests 2 and 3 should be run in combination with
   varying complexity.
```

---

```
********** FUNCTION 5: check_ptype() *****************
TEST FUNCTION NAME: check_ptype()
CALLED BY: transitions module
PURPOSE: Check a parameter for correct type.
INTERNAL FUNCTIONS CALLED:
   implicit_decl(){Function 2.2}

FUNCTION DESCRIPTION:
   Takes as arguments an actual parameter and a type.
   The function verifies the parameter to be of that
   type.
DRIVER:
   Calls check_ptype() with a parameter(a) and a type(p).
STUBS:
```

```
   is_var(), is_const(), is_def(), and is_int_trans()
   all return false unless stated otherwise in test case.
TEST CASES:
   1. Call Function with an integer parameter.
   2. Call Function with a variable id as parameter.
   3. Call Function with a constant id as parameter.
   4. Call Function with a define id as parameter.
   5. Call Function with an undefined id as parameter.
      Should use implicit_decl(){Function 2.1} to
      declare the id.
NOTES:
   Tests 1 - 4 should be run with all different types
   to ensure function is working for all possibilities.
```

---

```
********** FUNCTION 6: check_types() *****************
TEST FUNCTION NAME: check_types()
CALLED BY: predicates module
PURPOSE: Check expressions for undeclared constants.
INTERNAL FUNCTIONS CALLED:
   find_type(), check_set_type()

**** FUNCTION 6.1 - find_type() ****

FUNCTION DESCRIPTION:
   Find_type() attempts to determine the type of an
   expression. If it can, it returns the type; if not
   it returns nil.
DRIVER:
   Calls find_type() with an expression(e).
TEST CASES:
   1. Call function with ‘true’ as parameter. Should
      return boolean.
   2. Call function with an integer as parameter. Should
      return integer.
   3. Call function with an id as parameter. Should
      return id’s type, if it exists; otherwise, return nil.
   4. Call function with an arithmetic expression.
      Should return integer.
```

```
   5. Call function with a set expression. Should return
      nil.
   6. Call function with a list expression. If it is
      List length, should return integer. Otherwise,
      return nil.
   7. Call function with an expression not included in
      tests 4-6. Should return nil.
NOTES:
   Test 1 should also be run with False.
   Tests 5, 6, and 7 reflect the current operation of the tool.

**** FUNCTION 6.2 - check_set_type() ****

FUNCTION DESCRIPTION:
   Given a set expression, looks through elements of set for
   any undefined ids.
DRIVER:
   Calls check_set_type() with a set expression(e) and
   the type of the set's elements(etype).
TEST CASES:
   1. Call function with a complex set. Should return with
      no changes.
   2. Call function with a type name for etype. Should
      break down etype to find actual type of set elements.
   3. Call function with etype equal to a set type, and e
      representing a simple set. Should check each element
      of e.
   4. Call function e a simple set containing a type conflicting
      element. Should cause type_mismatch error.

**** FUNCTION 6 - check_types() ****

FUNCTION DESCRIPTION:
   Given an user-entered expression, parses the expression,
   looking for undefined ids. Attempts to infer the
   type of the constants from the surrounding operators
   and operands.
DRIVER:
   Sets addingpred to true(indicates user-entered predicate);
```

```
   Initializes constant, variable, define lists;
   Calls check_types() with an expression(e).
STUBS:
   is_var(), is_const(), is_def(), and is_int_trans()
   all return false unless stated otherwise in test case.
   get_variable_ptypes() performs as described in each test.
TEST CASES:
   1. Expression contains no undefined ids. Return expression
      as entered, with no change to any id lists.
   2. Expression contains one undefined id of integer type.
      Return expression as entered, with id added to
      temporary list.
   3. Expression contains one undefined id as part of a
      reduces_to operation.
   4. Expression contains one undefined id as part of a
      set-membership operation.
NOTES:
   Tests 1 - 4 should be run with many different types to ensure
   the function can infer when involved with the above-mentioned
   operators.
```

---

### 2.5.3 The Transitions Module

The following outline the unit tests for the transitions module.

---

```
********** FUNCTION 1: set_transition() *****************
TEST FUNCTION NAME: set_transition()
CALLED BY: aslantest.lisp initialization file
PURPOSE: correctly store the components of a transition.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Takes as an argument a transition id, a value, and
   a attribute. If the attribute is args, entry,
   effect, or line_no, it simply assigns the given
   value to that attribute. If it is except or exc_exit,
```

```
   it must append the value to the end of the approp.
   list and correctly manipulate the modified vars.
DRIVER:
   Initializes the relevant transition lists and
   attributes, and calls set_transition() with a
   transition id(trans_id), a value(value), and
   an attribute(property).
TEST CASES:
   1. Call set_transition() with an entry attribute.
      Should set the attribute value and return.
   2. Call with an except attribute. Should append
      the given except condition to the current list
      of excepts for that transition.
   3. Call with an except exit condition. Should
      add the except exit to the current list, xfer the
      list of modified variables from the temporary list
      to the permanent list, and set the temporary list
      to nil.
NOTES:
   Test 1 should be run with all the trivial attributes.
   Add_transition() is a trivial function that adds a
   transition id to the temporary list of ids.
```

---

```
********** FUNCTION 2: get_transition() ******************
TEST FUNCTION NAME: get_transition()
CALLED BY: predicates and driver modules
PURPOSE: retrieve specified attributes of a transition.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Takes a transition id and an attribute as arguments. If
   the id is a defined transition, retrieves the attribute
   value. If not, responds with an error message.
DRIVER:
   Initializes the transition lists and relevant attributes.
   Calls get_transition() with an id(t_id) and an
   attribute(property).
TEST CASES:
```

```
   1. Id is a valid transition and property a valid
      attribute. Return the attribute value.
   2. Id is a valid transition but property is not
      a valid attribute. Return nil.
   3. Id is no a valid transition. Respond with an
      error message.
NOTES:
   Similar tests can be run on get_transition_args(),
   get_exit(), get_entry(), get_except(), and
   get_exc_exit(). All are simply extensions of
   get_transition().
```

---

```
********** FUNCTION 3: is_transition() *****************
TEST FUNCTION NAME: is_transition()
CALLED BY: predicates and driver modules
PURPOSE: Return true if an id is a transition id.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given an id, returns true if the id is in either the
   permanent or temporary transition list. Otherwise,
   return nil.
DRIVER:
   Initializes the transition lists. Calls is_transition()
   with an id(trans_id).
TEST CASES:
   1. Id is in temporary transition list. Return true.
   2. Id is in permanent transition list. Return true.
   3. Id is not in either list. Return nil.
```

---

```
********** FUNCTION 4: make_trans_id() *****************
TEST FUNCTION NAME: make_trans_id()
CALLED BY: input/output and driver modules
PURPOSE: Given a string or number return a trans. id.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
```

```
   Takes as input either a string or a number. If it's
   a number, returns the transition in the list
   corresponding to that number. If its a string, returns
   the string.
DRIVER:
   Initializes the transition list. Calls make_trans_id()
   with a string or number(id).
TEST CASES:
   1. Id is a string. Return the string.
   2. Id is a number within the range of the transition
      list. Return the transition id.
   3. Id is a number outside the range of the list.
      Return nil.
```

---

```
********** FUNCTION 5: add_modified_var() *****************
TEST FUNCTION NAME: add_modified_var()
CALLED BY: aslantest.lisp initialization file
PURPOSE: adds a modified variable to the list for current trans.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Takes as input a transition id and a variable id. If
   the variable id is not a define id or is not already in
   the list of modified variables for that transition, it
   is added. Otherwise, it is ignored.
DRIVER:
   Initializes the modified list for a transition.
   Calls add_modified_var() with a transition id(t_id)
   and an id(v).
TEST CASES:
   1. V is a define id. Function is a No-Op.
   2. V is already in the modified list. Function is a No-Op.
   3. V is not a define id and is not in the modified list.
      V should be added to the modified list.
NOTES:
   add_mod_var_exc() is a very similar function used for
   except-exit pairs.
```

---

```
********** FUNCTION 6: adj_mvars() *****************
TEST FUNCTION NAME: adj_mvars()
CALLED BY: predicates module
PURPOSE: Updates the modified variable list.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Takes as input an expression and a modified variable
   list. Goes through the expression, looking for modified
   variables in the expression which aren't in the list.
   Adds them to the list. Returns the updated list.
DRIVER:
   Calls adj_mvars() with an expression(wff) and a list of
   modified variables(mvars).
TEST CASES:
   1. mvars contains all modifieds in wff. Return original
      mvars.
   2. wff contains modifieds not in mvars. Return updated
      mvar list.
```

---

```
********** FUNCTION 7: init_transitions() *****************
TEST FUNCTION NAME: init_transitions()
CALLED BY: control module
PURPOSE: Expands transition bodies.
INTERNAL FUNCTIONS CALLED: fix_trans(), fix_excepts()

**** FUNCTION 7.1 - fix_excepts() ****

FUNCTION DESCRIPTION:
   Takes as an input a list of modified variables, a list
   of expressions corresponding to a transition's except
   or exit conditions, and a type, which tells us whether
   these are except bodies or except exits. Expands each
   expression in the list, replacing define references
   with define bodies, expanding NoChange operators, and
   giving bound variable references unique ids. Updates
   modified variable lists as it goes along.
```

DRIVER:
   Sets up a transition and its attributes. Calls fix_excepts() with a list of expressions(ex_list), the type of the expression(ex_type), and a list of modified var lists(ex_mvars).
STUBS:
   expand_NOCHANGE(), expand_wff(), and fix_BOUND() all return NOT_CHANGED.
TEST CASES:
   1. A single expression list.
   2. A multiple expression list.
NOTES:
   All the work for this function is done by the three functions in the predicates module. All that can be tested here is that the number of items in each list argument remains intact.

**** FUNCTION 7.2 - fix_trans() ****

FUNCTION DESCRIPTION:
   Very similar to fix_excepts(), but more general. Expands transition entry's and exit's, uses fix_excepts() to expand its except-exit pairs.
DRIVER:
   Sets up a transition and its attributes. Calls fix_trans() with transition id(tr).
STUBS:
   expand_NOCHANGE(), expand_wff(), and fix_BOUND() all return NOT_CHANGED.
TEST CASES:
   1. Transition has no except-exit pairs.
   2. Transition has multiple except-exit pairs.
NOTES:
   Again, most of the work is done in the predicates module. The real testing of this function will occur at the integration stage.

---

********** FUNCTION 8: fixed_exit() *****************
TEST FUNCTION NAME: fixed_exit()

```
CALLED BY: driver module
PURPOSE: Establishes new bound variable ids.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Takes as input a transition id. If the id is in the
   transition list, establishes all new bound variable
   ids in the exit body. Called before each execution
   of the transition.
DRIVER:
   Initializes the transition and its attributes.
   Calls fixed_exit() with a transition id(x).
STUBS:
   fix_BOUND() returns its argument unchanged.
TEST CASES:
   1. X is a valid transition. Return exit body.
   2. X is not a valid transition. Respond with
      error message - unknown transition.
NOTES:
   fix_entry() and fix_except() perform the same
   function for entry and except clauses, respectively.
```

---

```
********** FUNCTION 9: chk_trans_call() *****************
TEST FUNCTION NAME: chk_trans_call()
CALLED BY: driver module
PURPOSE: Verifies correctness of transition arguments.
INTERNAL FUNCTIONS CALLED: chk_trans_args(), chk_trans_arg()

**** FUNCTION 9.1 - chk_trans_arg() ****

FUNCTION DESCRIPTION:
   Checks an individual transition argument to make sure
   its of the correct type. If the type is correct, returns
   true. If the type is incorrect, returns nil.
   If the argument is a variable, responds with an error
   message.
DRIVER:
   Sets up ids and types. Calls chk_trans_arg() with a
```

```
   transition name(name), an actual argument(actual), and
   what should be its corresponding type(formal).
STUBS:
   check_types(), int_tst(), bool_tst(), const_tst(),
   check_const() perform as described in each test case.
TEST CASES:
   1. actual is a variable id. Respond with an error message.
   2. Actual is a bound variable. Return nil.
   3. Actual is a number, and formal is a type for which
      numbers are ok. Return true.
   4. Actual is a number, and formal is a type for which
      number aren't ok. Return nil.
   5. Actual is 'true', and formal is boolean. Return true.
   6. Actual is 'false', and formal is boolean. Return false.
   7. Actual is a constant of the correct type. Return true.
   8. Actual is a constant of incorrect type. Return nil.
NOTES:
   As with Functions 7 and 8, most of the work for this
   function is done by external functions (in the types
   module).  Hence, the significant testing of this module
   will be done at the integration level.

**** FUNCTION 9.2 - chk_trans_args() ****

FUNCTION DESCRIPTION:
   Given a list of actual parameters and formal types, verifies
   that each actual parameter is of the correct type. Also
   verifies that the number of actuals matches the number of
   formal types. Calls chk_trans_arg() to verify each actual.
DRIVER:
   Sets up ids and types. Calls chk_trans_args() with a
   transition name(name), a list of actual parameters(actuals)
   and a list of formal parameters(formals).
TEST CASES:
   1. Number of Actuals is different than number of formals.
      Respond with Error message.
   2. List of actuals and formals are both nil. Return true.
   3. List of actuals and formals are non-nil.
NOTES:
   Test 3 should be run with actual-formal combinations
```

```
   described in Function 9.1 tests 1-8 above.

**** FUNCTION 9 - chk_trans_call()

FUNCTION DESCRIPTION:
   Verifies the transition id is a valid transition.
   Calls chk_trans_args() to verify arguments.
DRIVER:
   Initializes the transition. Calls chk_trans_call()
   with a transition id(name) and a list of arguments(actuals).
TEST CASES:
   1. Name is not a valid transition id. Respond with
      an error message.
NOTES:
   Additional tests should be run with valid transition
   ids, using argument lists described in Function 9.2
   test cases.
```

---

### 2.5.4 The Predicates Module

This module handles all manipulation of Aslan predicates and expressions.

---

```
********** FUNCTION 1: sub_params() *****************
TEST FUNCTION NAME: sub_params()
CALLED BY: driver, variables, and transitions module
PURPOSE: Substitutes actuals for formals in an expression.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Takes as input an expression, a list of actuals,
   and a list of formals and their types. Replaces
   all instances of each formal with its corresponding
   actual.
DRIVER:
   Calls sub_params() with an expression(e), a list
   of actuals(actuals), and a list of formals(formals).
```

```
TEST CASES:
   1. E contains no formal references. Should return
      E unchanged.
   2. E contains formal references. Should return e
      with references replaced by actuals.
   3. List of actuals is nil. Should return e
      unchanged.
   4. List of formals is nil. Should return e
      unchanged.
NOTES:
   sub_params2() performs the same function, but the
   formal list is simply a list of ids.
```

---

```
********** FUNCTION 2: sub_old_values() *****************
TEST FUNCTION NAME: sub_old_values()
CALLED BY: driver, variables, transitions and
           evaluator modules
PURPOSE: Replaces current value variable references with
         current values.
INTERNAL FUNCTIONS CALLED:
   is_boundtype(),
   sub_old_wo_params(),
   sub_old_with_params()

**** FUNCTION 2.1 - is_boundtype() ****

FUNCTION DESCRIPTION:
   Returns true if the argument is a bound variable
   structure.
DRIVER:
   Calls is_boundtype() with an expression(e).
TEST CASES:
   1. E is a bound variable reference. Return true.
   2. E is not a bound variable reference. Return nil.

**** FUNCTION 2.2 - sub_old_wo_params() ****

FUNCTION DESCRIPTION:
```

```
   If the given id is a variable or constant, returns
   its value. Otherwise, returns the id.
DRIVER:
   Calls sub_old_wo_params() with an id(id).
STUBS:
   is_var(), is_const(), variable_value(), and
   constant_value() perform as described in each
   test case.
TEST CASES:
   1. Id is a non-parameterized variable id. Should
      return its value.
   2. Id is a non-parameterized constant id. Should
      return its value.
   3. Id is not a valid id. Should return the id.

**** FUNCTION 2.3 - sub_old_with_params() ****

FUNCTION DESCRIPTION:
   Returns current value associated with a parameterized
   id reference. If the id is not a variable or constant,
   simply returns the reference.
DRIVER:
   Calls sub_old_with_params() with a parameterized
   id reference(e).
STUBS:
   sub_old_values(), get_list_value(), get_struct_value(),
   is_var(), is_const(), variable_value(), and constant_value(),
   perform as described in each test case.
TEST CASES:
   1. E is a reference to a valid single_parameter variable.
      Return its value.
   2. E is a reference to a valid double parameter variable.
      Return its value.
   3. E is a reference to a list structure variable. Return
      its value.
   4. E is a reference to a struct variable. Return its value.
   5. E is a reference to a parameterized list variable.
      Return its value.
   6. E is a reference to a parameterized struct variable.
      Return its value.
```

```
   7. E is a parameterized constant; return e.
   8. E is not a valid reference. Return e.
NOTES:
   sub_old_with_params() is interesting because it makes a call
   back to the function that called it(sub_old_values). To
   ensure proper testing of this function, we use a stub for
   sub_old_values().

**** FUNCTION 2 - sub_old_values() ****

FUNCTION DESCRIPTION:
   Moves through an expression, replacing all current
   value variable and constant references with the actual
   current values.
DRIVER:
   Calls sub_old_values() with an expression(e).
TEST CASES:
   1. E contains no current value references. Return e.

   All other test cases involve combinations of tests
   listed for Function 2.2 and 2.3. Three examples are
   given in the test file. With the exception of the
   sub_old_values() stub for Function 2.3, all stubs
   remain and function as described in each test.
```

---

```
********** FUNCTION 3: add_wff() *****************
TEST FUNCTION NAME: add_wff()
CALLED BY: initialize and evaluator modules
PURPOSE: Conjoins two boolean expressions.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Conjoins two boolean expressions.
DRIVER:
   Calls add_wff() with two boolean expressions(wff1 wff2).
STUBS:
   ANDIZE() returns '(.AND wff1 wff2)'.
TEST CASES:
```

```
   1. wff1 and wff2 are both 'true'.
   2. wff1 is 'true'.
   3. wff2 is 'true'.
   4. wff1 and wff2 are non-trivial boolean expressions.
NOTES:
   These test cases will be most useful at the
   integration stage.
```

---

```
********** FUNCTION 4: has_NEW() *****************
TEST FUNCTION NAME: has_NEW()
CALLED BY: evaluator module
PURPOSE: Looks for new value references in expressions.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Takes as input an expression. Returns true if the
   expression contains a new value reference, other-
   wise, returns nil.
DRIVER:
   Calls has_NEW() with an expression(expr).
TEST CASES:
   1. E contains new value references. Return true.
   2. E contains no new value references. Return nil.
   3. E is null. Return nil.
   4. E is a single id. Return nil.
NOTES:
   has_BOUND() performs the same function, looking
   for bound variable references.
```

---

```
********** FUNCTION 5: is-bound() *****************
TEST FUNCTION NAME: is-bound()
CALLED BY: variables module
PURPOSE: Returns true if bound variable is valid.
INTERNAL FUNCTIONS CALLED: BOUNDP1()

**** FUNCTION 5.1 - BOUNDP1() ****
```

```
FUNCTION DESCRIPTION:
   Given a bound variable reference and a list of
   bound variables, returns true if the reference
   is in the list.
DRIVER:
   Calls BOUNDP1() with a bound var reference(id)
   and a list of bound variables(bound).
TEST CASES:
   1. Id is in bound list. Return true.
   2. Id is not in bound list. Return nil.
   3. Bound list is null. Return nil.

**** FUNCTION 5 - is-bound() ****

FUNCTION DESCRIPTION:
   Given a bound variable reference, returns true
   if it is in the list of currently used bound
   variables. Otherwise, returns nil.
DRIVER:
   Sets up bound variable list. Calls is-bound()
   with a bound variable reference(id).
TEST CASES:
   Same test cases as for function 5.1
```

---

```
********** FUNCTION 6: fix_BOUND() *****************
TEST FUNCTION NAME: fix_BOUND()
CALLED BY: driver, transitions, variables,
           and evaluator modules
PURPOSE: Generates unique ids for each bound ref.
INTERNAL FUNCTIONS CALLED: gen_bound(), genid()

**** FUNCTION 6.1 - genid() ****

FUNCTION DESCRIPTION:
   Generates unique three digit ids, preceded by
   an underscore.
DRIVER:
   Initializes gencount variable. Calls genid().
```

```
TEST CASES:
   1. Gencount = 0. Return '_001' and gencount = 1.
   2. Gencount = 9. Return '_010' and gencount = 10.
   3. Gencount = 99. Return '_100' and gencount = 100.

**** FUNCTION 6.2 - gen_bound() ****

FUNCTION DESCRIPTION:
   Given a type, calls genid() to generate a unique
   bound id and adds the id and the type to the bound
   variable list.
DRIVER:
   Initializes gencount and boundlist.
   Calls gen_bound() with a type(type).
TEST CASES:
   1. Boundlist is null. Should return unique bound id
      and boundlist contains single item.
   2. Boundlist is not null. Should return unique id
      and boundlist should be appended by that id and
      the given type.
NOTES:
   Tests 1 and 2 should be run with gencount as described
   in tests 1 - 3 for Function 6.1.

**** FUNCTION 6 - fix_BOUND() ****

FUNCTION DESCRIPTION:
   Takes as input an expression. Moves through expression,
   replacing all bound variable references with unique
   bound ids.
DRIVER:
   Initializes gencount and boundlist. Calls fix_BOUND()
   with an expression(e).
TEST CASES:
   1. E is null. Gencount and boundlist are unchanged
      and return nil.
   2. E contains no quantified expressions. Gencount
      and boundlist are unchanged and return e.
   3. E contains quantified expressions. E is returned
      with unique bound ids for each bound variable in
```

```
      e. Gencount and boundlist are modified accordingly.
```

---

```
********** FUNCTION 7: expand_NOCHANGE() *****************
TEST FUNCTION NAME: expand_NOCHANGE()
CALLED BY: variables, and transitions modules
PURPOSE: Replaces NOCHANGE ops with equality ops.
INTERNAL FUNCTIONS CALLED: expand_nc1(), expand_NC()

**** FUNCTION 7.1 - expand_NC() ****

FUNCTION DESCRIPTION:
   Takes as input an id or a reference structure that is
   part of a NOCHANGE statement. Returns assignment
   statement corresponding to the nochange operator.
DRIVER:
   Calls expand_NC() with a variable reference(ref).
TEST CASES:
   1. Ref is an id. Return id = id'.
   2. Ref is a parameterized reference(id(a)).
      Return id(a) = id'(a).

**** FUNCTION 7.2 - expand_nc1() ****

FUNCTION DESCRIPTION:
   Takes a list of variable references as input. For
   each reference, call expand_NC() to get appropriate
   assignment statement. If the list is null, indicates
   the NOCHANGE for all modified variables. Use list
   of modified variables for current transition as list.
DRIVER:
   Initializes a transition. Calls expand_nc1() with
   a list of vars(vars).
TEST CASES:
   1. List has a single reference. Return assignment.
   2. List has multiple references. Return conjunction
      of assignment statements.
   3. List is null. Retrieve modified list, return
      assignment statements.
```

```
**** FUNCTION 7 - expand_NOCHANGE() ****

FUNCTION DESCRIPTION:
   Takes as input an expression. Moves through it,
   looking for NOCHANGE operators. Uses expand_nc1()
   to convert them to assignment statements.
DRIVER:
   Initializes a transition. Calls expand_NOCHANGE()
   with an expression(wff).
TEST CASES:
   1. Wff contains no NOCHANGE ops. Return wff unchanged.
   2. Wff contains NOCHANGE op with no variable
      arguments. Return wff with NOCHANGE assignments
      for each modified variable.
   3. Wff contains multiple NOCHAANGE ops. Return wff
      with NOCHANGE assignments for each mod. var.
   4. Wff is null. Return nil.
```

---

```
********** FUNCTION 8: fix_non_boolean_COND() *****************
TEST FUNCTION NAME: fix_non_boolean_COND()
CALLED BY: evaluator module
PURPOSE: Translates non-boolean cond. to boolean conditional.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Converts expression of the form:
     a = (if b then c else if d then e)
   to:
    (if b then a = c else if d then a = e)
DRIVER:
   Calls fix_non_boolean_COND() with a variable
   reference(var_ref) and a conditional(if_expr).
TEST CASES:
   1. Conditional has no else clause.
   2. Conditional has multiple else clauses.
```

---

```
********** FUNCTION 9: make_NEW() *****************
TEST FUNCTION NAME: make_NEW()
CALLED BY: evaluator module
PURPOSE: Converts all variable references to new value
         references.
INTERNAL FUNCTIONS CALLED: makenew1(), makenew2()

**** FUNCTION 9.1 - makenew2() ****

FUNCTION DESCRIPTION:
   Replaces non-parameterized ids with new value
   references. If the id is a state variable, returns
   a new reference structure. If the id is a define,
   retrieves the define body, and calls make_NEW()
   to find other new value references. If the id
   is not a variable or define, an error message
   is displayed.
DRIVER:
   Initializes defines and variable lists. Calls
   makenew2() with an id reference(e).
STUBS:
   Make_NEW() returns its argument.
   Get_definition() returns a definition body.
   Is_def() performs as described in each test.
TEST CASES:
   1. E is a variable reference. Return new value
      reference.
   2. E is a define reference. Return body of
      define.
   3. E is an unknown reference. Respond with
      error message.

**** FUNCTION 9.2 - makenew1() ****

FUNCTION DESCRIPTION:
   Replaces parameterized id references with new
   value references. Uses check_types() to check
   the type of the parameters. Calls make_NEW()
   to look for new value references in parameter
   lists and define bodies.
```

```
DRIVER:
   Initializes defines and variable lists. Calls
   makenew1() with a parameterized id reference(e).
STUBS:
   Is_var(), is_def(), is_int_trans() perform
   as needed in each test.
   Check_types() performs as needed in each test.
   Make_NEW() returns its argument.
   Get_definition(), get_variable_args() return
   requested values.
TEST CASES:
   1. E is a variable reference with valid parameters.
      Return new value reference.
   2. E is a define reference with valid parameters.
      Return new value reference.
   3. E is neither 1 nor 2. Return original reference.

**** FUNCTION 9 - make_NEW() ****

FUNCTION DESCRIPTION:
   Takes as input an expression. Moves through it,
   looking for variable references and replacing
   them with new value references.
DRIVER:
   Initializes variable and define lists. Calls
   make_NEW() with an expression(e).
STUBS:
   is_variable(), is_def(), is_int_trans() perform
   as needed for each test.
   All stubs listed for functions 9.1 and 9.2 still
   apply, with the exception of make_NEW().
TEST CASES:
   1. E contains no variable references. Should return
      E unchanged.
   2. E contains variable references. Should return e
      with references replaced by new value references.
NOTES:
   Test 2 should be run with combinations of tests
   described in Functions 9.1 and 9.2.
```

```
********** FUNCTION 10: expand_wff() *****************
TEST FUNCTION NAME: expand_wff()
CALLED BY: driver, variables, transitions,
           and evaluator modules
PURPOSE: Expands expressions
INTERNAL FUNCTIONS CALLED: expand_wff_new()

**** FUNCTION 10.1 - expand_wff_new() ****

FUNCTION DESCRIPTION:
   Replaces a new value reference of a define with
   its definition. Replaces all variable references
   with new value references.
DRIVER:
   Initializes defines. Calls expand_wff_new()
   with a reference(ref).
STUBS:
   Is_def(), is_int_trans() perform as needed in each
   test; get_definition() returns a definition body.
   Get_variable_args() returns a definitions parameters.
TEST CASES:
   1. Ref is a State variable reference. Return it
      unchanged.
   2. Ref is a define reference. Return its body with
      new value references.
   3. Ref is an internal transition. Return its body.
NOTES:
   Tests 1 - 3 should be run with parameterized and
   non-parameterized references.

**** FUNCTION 10 - expand_wff() ****

FUNCTION DESCRIPTION:
   Takes as input an expression. Moves through it,
   expanding define references and replacing actual
   arguments for parameters.
DRIVER:
   Initializes define and variable lists. Calls
```

```
   expand_wff() with an expression(wff).
STUBS:
   Is_def(), is_transition() perform as needed in each
   test.
   Get_variable_args() returns id parameters.
   Get_transition_args() returns transition arguments.
   Get_definition() returns define bodies.
TEST CASES:
   1. Wff contains no define references. Return wff.
   2. Wff contains define references with old value
      operators. Return wff with bodies replacing
      references.
   3. Wff contains new value references to define
      bodies. Return wff with references replaced by
      bodies.
NOTES:
   Tests should be run with parameterized and non-
   parameterized references.
```

---

### 2.5.5 The Initialize Module

The initialize module is a fairly trivial module, containing simple function calls to other modules to initialize the system and set up system variables. All calls to this module are made from the aslantest.lisp initialization file. Functions like init_spec(), init_global_consts(), and init_global_vars() are completely made up of assignment statements. Testing them is simply a matter of running them and making sure each assignment statement was invoked.

---

```
********** FUNCTION 1: add_initial_cond() ******************
TEST FUNCTION NAME: add_initial_cond()
CALLED BY: aslantest.lisp initialization file
PURPOSE: Adds an expression to the initial conditions.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Adds an expression to the initial conditions.
```

DRIVER:

Initializes the initial conditions.
Calls add_initial_cond() with an expression(wf_formula).

STUBS:

add_wff() returns the conjunction of its arguments.

TEST CASES:

1. Wf_formula is ‘true’. Return conjunction.
2. Initial conditions are ‘true’. Return conjunction.
3. Wf_formula and initial conditions are non-trivial expressions. Return conjunction.

NOTES:

Tests take on more substance at the integration stage, where simplification can be tested.
Functions add_criterion() and add_constraint() are identical in form.

---

********** FUNCTION 2: add_axiom() *****************

TEST FUNCTION NAME: add_axiom()
CALLED BY: aslantest.lisp initialization file
PURPOSE: Adds an axiom to the axiom list.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:

Conjoins an axiom to the current axiom list.

DRIVER:

Initializes axiom list. Calls add_axiom() with an axiom(wf_formula).

TEST CASES:

1. Axiom list is null. Set axiom list equal to wf_formula.
2. Axiom list has single axiom. Set axiom list equal to conjunction of two axioms.
3. Axiom list is a conjunction. Add wf_formula to the list of conjuncts.

---

### 2.5.6 The Input/Output Module

This module breaks into a collection of input routines and a collection of output routines. This module is unique because many of its routines are not called from outside the module. The input routines are outlined first.

#### 2.5.6.1 The Input Routines

The input routines consist of a collection of functions that manipulate and check user input, as well as a collection of routines that retrieve input from the user.

---

```
********** FUNCTION 1: is_space() *****************
TEST FUNCTION NAME: is_space()
CALLED BY: driver module
PURPOSE: Return true if the character is a space or a tab.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a character, returns true if the character is
   a space or a tab. Otherwise, returns nil.
DRIVER:
   Calls is_space() with a character(c).
TEST CASES:
   1. C is a space. Return true.
   2. C is a tab. Return true.
   3. C is not a tab or space. Return nil.
```

---

```
********** FUNCTION 2: make_num() *****************
TEST FUNCTION NAME: make_num()
CALLED BY: driver module
PURPOSE: Converts a string to its numeric equivalent.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a numeric input, returns a number. If the input
```

```
   is an integer, returns the integer. If the input
   is a numeric string, returns the integer value. If
   the input is a non-numeric string, returns the string.
DRIVER:
   Calls make_num() with an input(sym).
TEST CASES:
   1. Sym is an integer. Return the integer.
   2. Sym is a numeric string. Return the integer value.
   3. Sym is non-numeric. Return the string.
```

---

```
********** FUNCTION 3: check_bound() ******************
TEST FUNCTION NAME: check_bound()
CALLED BY: predicates module
PURPOSE: Converts a user-entered bound ref. to internal ref.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given an id, if it is a user-entered bound variable,
   returns the internal representation. If not, returns
   the id.
DRIVER:
   Calls check_bound() with an id(id).
TEST CASES:
   1. C is a user-entered bound variable. Return the
      internal rep.
   2. C is not a bound variable reference. Return id.
```

---

```
********** FUNCTION 4: set_cmd_file() ******************
TEST FUNCTION NAME: set_cmd_file()
CALLED BY: driver module
PURPOSE: Redirect stdin to a textfile.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a filename, if the file exists and is readable,
   open it and set stdin to read from the file.
DRIVER:
```

```
   Calls set_cmd_file() with a filename(file_id).
TEST CASES:
   1. File_id is a accessible file. Open file, and reset
      stdin.
   2. File_id is not an accessible file. Respond with an
      error message.
```

---

```
********** FUNCTION 5: Command() ******************
TEST FUNCTION NAME: Command()
CALLED BY: driver module
PURPOSE: Parses user input for a command.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given user input, searches it for a command. If
   it is nothing but spaces and tabs, returns nil.
   Otherwise, returns the command.
DRIVER:
   Calls Command() with user input(cmd).
TEST CASES:
   1. Cmd is all spaces and tabs. Return nil.
   2. Cmd contains a command. Return the command.
   3. Cmd begins with spaces and tabs followed by a
      command. Return the command.
```

---

```
********** FUNCTION 6: form_args() ******************
TEST FUNCTION NAME: form_args()
CALLED BY: driver module
PURPOSE: Converts a string of input to internal reps.
INTERNAL FUNCTIONS CALLED: Convertargs()

**** FUNCTION 6.1 - Convertargs()

FUNCTION DESCRIPTION:
   Given a list of ids, converts them to the proper
   internal representation. This involves translating
   parameterized variables and bound variables.
```

DRIVER:
   Call Convertargs() with a list of ids(idlist).
TEST CASES:
   1. Idlist contains no parameterized ids or bound
      references. Return idlist.
   2. Idlist contains parameterized ids. Return idlist
      with internal representations.
   3. Idlist contains bound references. Return idlist
      with proper internal representations.

**** FUNCTION 6.2 - form_args() ****

FUNCTION DESCRIPTION:
   Given a transition id and a list user-entered arguments,
   converts the arguments to their correct internal
   representations and return the id and the idlist.
DRIVER:
   Calls form_args() with a transition id(id) and a list
   of arguments(idlist).
TEST CASES:
   1. Id is null. Return nil.
   2. Idlist is null. Return id.
   Tests 3 - 5 set idlist to values used in Function 6.1
   tests 1 - 3. Return id and converted idlist.

---

********** FUNCTION 7: read_cmd_file() *****************
TEST FUNCTION NAME: read_cmd_file()
CALLED BY: evaluator module
PURPOSE: retrieve user input.
INTERNAL FUNCTIONS CALLED:

FUNCTION DESCRIPTION:
   Retrieves user input. If the input is coming from a file,
   should echo the input. If the input is an Aslantest
   comment or consists only of spaces, should return nil.
DRIVER:
   Uses a special function to call read_cmd_file() so it
   can read user input.

```
TEST CASES:
   1. User enters all spaces and tabs. Return nil.
   2. User enters non-whitespace. Return the input.
   3. Set stdin to a file. Input should be echoed.
NOTES:
   The same testing approach should be used to test the
   other retrieval functions: read_optional() and
   read_line_cmd_file(). Also, get_cmd_file() should
   be tested this way.
```

---

```
********** FUNCTION 8: get_command() ******************
TEST FUNCTION NAME: get_command()
CALLED BY: control module
PURPOSE: retrieve user inputted commands.
INTERNAL FUNCTIONS CALLED: make_cmd()

**** FUNCTION 8.1 - make_cmd() ****

FUNCTION DESCRIPTION:
   Given a user-entered string, determines if it is a
   valid Aslantest command.
DRIVER:
   Calls make_cmd() with a string(name).
TEST CASES:
   1. Name is not a valid command. Return "unknown name".
   2. Name is a valid command. Return command.

**** FUNCTION 8 - get_command() ****

FUNCTION DESCRIPTION:
   This function provides a user prompt, and then calls
   read_cmd_file to retrieve the user input, and make_cmd()
   to determine which command was entered.
DRIVER:
   Calls get_command().
TEST CASES:
   1. User enters invalid command. Return unknown command.
   2. User enters valid command. Return command.
```

```
   3. User enters all whitespace. Return nil.
   4. User enters Aslantest comment. Return nil.
```

---

```
********** FUNCTION 9: get_choice() *****************
TEST FUNCTION NAME: get_choice()
CALLED BY: evaluator module
PURPOSE: Retrieve user's choice of disjunct.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a list of disjuncts, presents them to the user
   and retrieves their selection. If the user's choice
   is invalid, assume the first is chosen.
DRIVER:
   Calls get_choice() with a list of disjuncts(disjuncts).
STUBS:
   asltest_pp() echoes its input.
TEST CASES:
   1. A single disjunct and user enters 1. Return disjunct.
   2. A single disjunct and user enters 2. Return disjunct.
   3. Multiple disjuncts and user enters valid selection.
      Return chosen disjunct.
   4. Multiple disjuncts and user enters invalid selection.
      Return first disjunct.
```

---

#### 2.5.6.2 The Output Routines

The output routines have three routines that are called from outside the module: asltest_pp(), outlist(), and out_typed_idlist(). Each of these use a collection of formatting and operator-special routines to output Aslan expressions.

---

```
********** FUNCTION 1: fits() *****************
TEST FUNCTION NAME: fits()
```

```
CALLED BY: Internal
PURPOSE: Return true if the exp. will fit on the current line.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Returns true if the given expression will fit on the
   current output line.
DRIVER:
   Initializes format variables. Calls fits() with an
   expression(e).
TEST CASES:
   1. E will fit. Return true.
   2. E will not fit. Return nil.
NOTES:
   Each of the following functions are similarly trivial:
   prnln() - returns length of an expression.
   blanks() - outputs n blank spaces.
   terprix() - starts a new line at current indentation.
   indent() - indents the line.
   outdent() - 'outdents' the line.
   prname() - Returns the output name of the given op.
   prec() - Returns the precedence of the given op.
```

---

```
********** FUNCTION 2: outATOM() *****************
TEST FUNCTION NAME: outATOM()
CALLED BY: Internal
PURPOSE: Outputs simple objects.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Prints its input. If the input is a variable or
   define reference, and the input flag is true,
   then the old value operator is outputted along
   with the reference. If the global variable
   'in_exit' is true, then the output is part of
   an exit condition and so the input may require
   the old value operator.
DRIVER:
```

```
   Initializes variable and define lists. Sets
   the global variable 'in_exit'. Calls
   outATOM() with an expression(e) and the old
   value flag(flag).
TEST CASES:
   1. In_exit is false. Echo input.
   2. In_exit and flag are true; input is a state variable.
      Echo input with an old value operator.
   3. In_exit and flag are true; input is a define reference.
      Echo input with old value operator.
   4. In_exit is true; flag is nil. Echo input.
NOTES:
   The formatting variable that tracks column position
   should also be updated as a result of the call.
   OutNEW(), outBOUND(), outSTRUCT(), and outVALUE()
   are all simple extensions of outATOM() and can be
   tested in the same manner.
```

---

```
********** FUNCTION 3: outEXPR() *****************
TEST FUNCTION NAME: outEXPR()
CALLED BY: Internal
PURPOSE: Outputs Aslan expressions.
INTERNAL FUNCTIONS CALLED:
   All operator special functions(outBOOL(), outREF(), etc.)

**** FUNCTION 3.1 - outBOOL() ****

FUNCTION DESCRIPTION:
   outBOOL() outputs a conjunction or disjunction.
DRIVER:
   Calls outBOOL() with a precedence(p) and an expression(e).
STUBS:
   outEXPR() simply echoes its output.
TEST CASES:
   1. E is a conjunction. Output the conjuncts in an
      in-order format.
   2. E is a disjunction. Output the disjuncts in an
      in-order format.
```

```
NOTES:
   For these test cases, precedence can be set to zero.
   When testing outEXPR() itself, will test the precedence.
   Out_.AND() and out.OR() simply call outBOOL().
   All the other operator special functions are tested
   in the same manner: outQUANT(), outCOND(), outNOT(),
   outMINUS(), outSTRUCT(), and outREF().

**** FUNCTION 3.2 - outSET_TYPE() ****

FUNCTION DESCRIPTION:
   Outputs a set representation.
DRIVER:
   Calls outSET_TYPE() with a set representation(e), and
   the precedence of the parent expression(p).
STUBS:
   outEXPR(), outlist(), and out_typed_idlist() simply
   echo their inputs.
TEST CASES:
   1. E is a simple set. Output the set enclosed in brackets.
   2. E is a complex set. Output the complex set
      representation.
NOTES:
   Again, precedence will be tested in the outEXPR() tests.
   outLIST_TYPE() is very similar to outSET_TYPE() and
   can be tested in the same fashion.

**** FUNCTION 3 - outEXPR() ****

FUNCTION DESCRIPTION:
   Outputs Aslan expressions. Each operator has a precedence
   associated with it. These are recorded in the variable
   pp_attrs. The print functions use these precedences to
   dictate the order of output and use of parentheses.
DRIVER:
   Initializes variable and define lists. Sets
   the global variable 'in_exit'. Calls pp_init()
   to initialize formatting variables. Calls outEXPR()
   with an expression(e) and a precedence(p).
TEST CASES:
```

```
   Test cases consist of combinations of cases given
   in Functions 2, 3.1 and 3.2.
```

---

```
********** FUNCTION 4: outlist() *****************
TEST FUNCTION NAME: outlist()
CALLED BY: driver module
PURPOSE: Outputs a list of expressions.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a list of expressions and a set of delimiters
   outputs the list.
DRIVER:
   Calls outlist() with a list of expressions(elist)
   a delimiter(delim), and opening and closing
   characters(lbracket, rbracket).
TEST CASES:
   1. Elist is a single expression. Output expression
      enclosed by bracket chars.
   2. Elist is multiple expressions. Output expressions
      separated by delimiters, enclosed by bracket chars.
NOTES:
   out_typed_idlist() is very similar. It outputs a list
   of id references. Both function assume a non-null
   list to be outputted.

   asltest_pp() is very simple. Given an expression, it
   simply sets up format variables and call outEXPR().
```

---

### 2.5.7 The Simplifier Module

Figure 2.2 shows a breakdown of the simplifier module. The simplifier consists of three main modules: a basic simplifier, an extended simplifier, and a set simplifier.

#### 2.5.7.1 The Set Simplifier

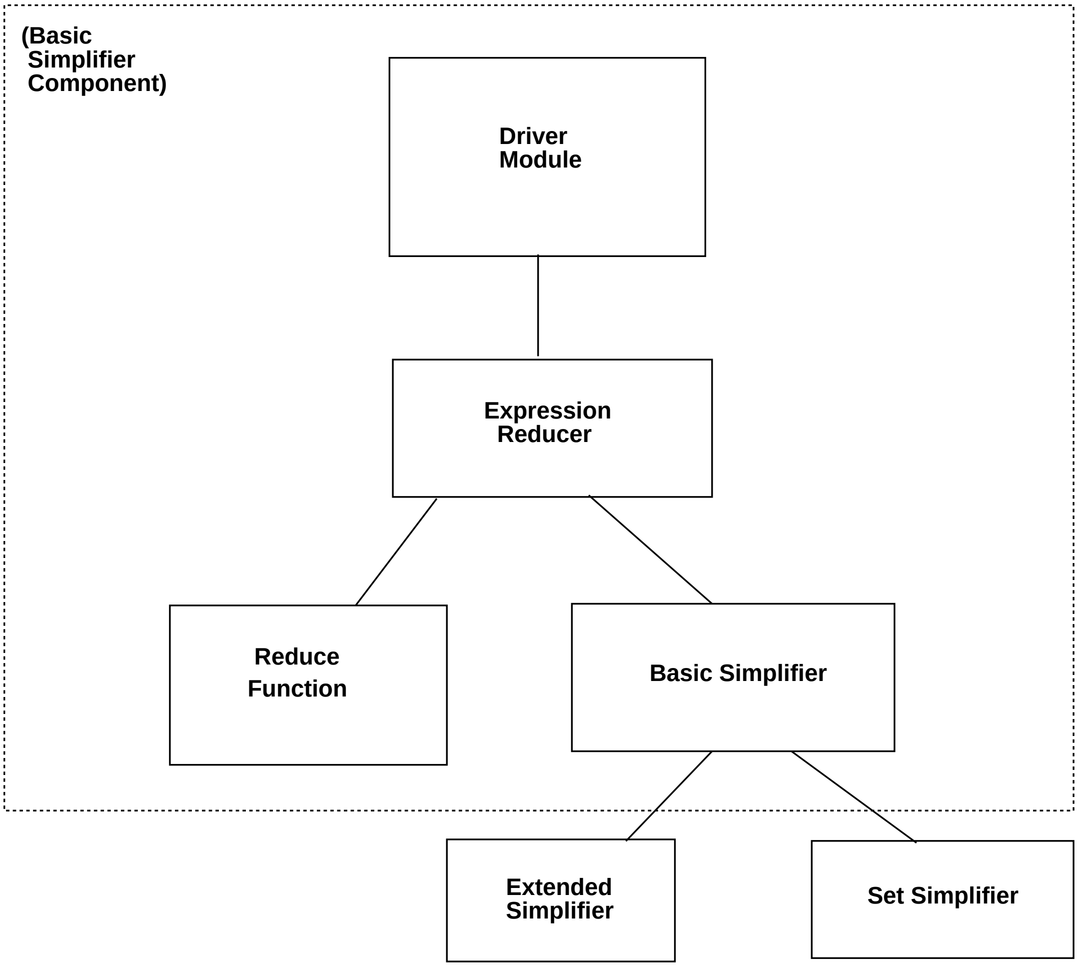


Figure 2.2 Aslantest Simplifier Module

The set simplifier contains routines to apply basic set operations, simplify sets, and convert complex sets to simple representations. The set simplification takes sets that are represented in the form of set union or set difference operations and attempts to produce the simplest represetation by combining elements whereever possible. It does this by breaking the set into its elements and then rebuilding a possibly simpler version.

The set simplifier also contains a routine to recognize a simple set in a complex set representation.

All of the routines in the set simplifier are fairly early in their development, and take very conservative approaches to their simplifications and reductions. This is an area of the simplifier that can definitely be further developed.

---

```
********** FUNCTION 1: Simp_Complex_Set() *****************
TEST FUNCTION NAME: Simp_Complex_Set()
CALLED BY: simplifier module(basic simplifier component)
PURPOSE: Attempts to recognize a simple set in a complex form.
INTERNAL FUNCTIONS CALLED: get_items()

**** FUNCTION 1.1 - get_items() ****

FUNCTION DESCRIPTION:
   Moves through a list of items collecting set elements.
   If an item in the list is not of the correct form, then
   it returns, alerting the caller of the error. It does
   this by returning True.
   For each item, the correct form is:
      (.EQ bvar item)   or  (.EQ item bvar)
   where bvar is a bound variable determined by the
   caller and item is any value other than bvar.
DRIVER:
   Calls get_items() with a list of items(blist).
TEST CASES:
   1. Blist is a valid list of elements. Return the
      list of elements.
   2. Blist contains an invalid element. Return True.
   3. Blist contains no valid elements. Return True.
   4. Blist is null. Return t.
```

```
NOTES:
   If successful the function returns 't', and the
   list of items is in the variable items. If not,
   it returns True using the lisp command throw.

**** FUNCTION 1 - Simp_Complex_Set() ****

FUNCTION DESCRIPTION:
   If the simplify flag is off, simply returns the original
   complex set representation. If it is on, the function
   checks the bound variable argument. It must be a
   single currently defined bound variable. Also the
   body of the set must be a disjunction or a single
   equivalence statement. Otherwise,
   the original set is returned. If the checks go through,
   get_items() is called to attempt to pull out each
   element.
DRIVER:
   Initializes simplify flag and boundlist. Calls
   Simp_Complex_Set() with a bound variable reference(bv)
   and the body of a complex set.
TEST CASES:
   1. Bv defines multiple bound variables. Return the
      original set.
   2. Bv defines an undefined bound variable. Return the
      original reference.
   3. Body is not a disjunction or equivalence statement.
      Return the original reference.
   4. Body and bv make up a single element simple set.
      Return the simple set.
   5. Body and bv make up a multiple element simple set.
      Return the simple set.
   6. Body and bv are valid, but the disjuncts within
      body are not. Return the original reference.
```

---

```
********** FUNCTION 2: Simp_Set() *****************
TEST FUNCTION NAME: Simp_Set()
CALLED BY: simplifier module(basic simplifier comp)
```

PURPOSE: Attempts to simplify set representations.
INTERNAL FUNCTIONS CALLED: build_comp_set(),
   build_dcomp_set(), rebuild_set(), break_down(),
   Simp_MEMBER()

**** FUNCTION 2.1 - build_comp_set() ****

FUNCTION DESCRIPTION:
   Given a list of complex set representations, builds
   the set union of the representations.
DRIVER:
   Calls build_comp_set() with a list of complex set
   representations(cs).
TEST CASES:
   1. Cs is null. Return nil.
   2. Cs is a single element. Return the element.
   3. Cs is a multiple element list. Return the
      union.
NOTES:
   build_dcomp_set() is the same function except it
   builds a set difference representation.

**** FUNCTION 2.2 - rebuild_set() ****

FUNCTION DESCRIPTION:
   This function uses four lists to build a single
   set representation. 'simp_set' is a list of
   simple elements in the set. 'dsimp_set' is a
   list of simple elements not in the set. 'comp_set'
   is a list of complex sets in the set. 'dcomp_set'
   is a list of complex sets not in the set.
   The resulting set is a single set using union
   and set diff where necessary.
DRIVER:
   Sets up lists. Calls rebuild_set().
STUBS:
   red_SET_DIFF() returns the set difference.
TEST CASES:
   1. Simp_set and dsimp_set contain elements, but
      others are empty. Return {simp_set} - {dsimp_set}.

2. Comp_set contains a list of complex sets.
   Return the union of the complex sets.

**** FUNCTION 2.3 - break_down() ****

FUNCTION DESCRIPTION:
   Break_down() breaks down set expressions consisting
   of union and set-diff operators and attempts to
   simplify their representations.
DRIVER:
   Calls break_down() with an op(op) - either UNION
   or SET_DIFF and two arguments(arg1 and arg2).
TEST CASES:

NOTES:

**** FUNCTION 2.4 - Simp_MEMBER() ****

FUNCTION DESCRIPTION:
   Simp_MEMBER() attempts to determine if a set element
   is contained within a set. Returns true, false or
   nil.
DRIVER:

TEST CASES:

**** FUNCTION 2 - Simp_Set() ****

FUNCTION DESCRIPTION:
   Given an operator and two arguments, returns a
   set representation. If the operator is MEMBER,
   UNION or SET_DIFF the function calls appropriate
   functions to attempt to simplify it before
   returning it. Otherwise, it simply returns the
   set representation.
DRIVER:
   Calls outlist() with a list of expressions(elist)
   a delimiter(delim), and opening and closing
   characters(lbracket, rbracket).
TEST CASES:

```
1. Elist is a single expression. Output expression
   enclosed by bracket chars.
2. Elist is multiple expressions. Output expressions
   separated by delimiters, enclosed by bracket chars.
```

---

#### 2.5.7.2 The Extended Simplifier

The extended simplifier is called if the basic simplifier is unable to simplify an expression. The extended simplifier uses the current contents of the state predicate to attempt to simplify the expression. In performing its evaluation, the simplifier first normalizes each predicate in the state predicate and adds it to one of four lists: the zero list(for equality exprs), the nonzero list(for inequality expr), the positive list(for greater than/less than expr) and the non simple list (for all other expressions). It then normalizes the expression in question and compares it against items in the lists.

---

```
********** FUNCTION 3: Normal_exp() *****************
TEST FUNCTION NAME: Normal_exp()
CALLED BY: extended simplifier
PURPOSE: Normalizes an expression.
INTERNAL FUNCTIONS CALLED:
   Convert(), Rid-(), Dist(), Rid+*(), Lexisort(),
   Docoeff(), Collect(), Insert0()

**** FUNCTION 3.1 - Convert() ****

FUNCTION DESCRIPTION:
   Given an expression, if it is an arithmetic
   equality/inequality operation, converts it
   so it takes the form of e = 0, e <> 0,
   or e > 0, where e is all the operands of
   the expression. If the expression is of the
   proper type operation it is simply returned.
   Negations of arithmetic operations are also
   converted.
DRIVER:
```

```
   Calls Convert() with an expression(e).
TEST CASES:
   1. Call Convert() with the expression a=b
      where a and b are expressions. Should
      return a-b=0.
   2. Call Convert() with the expression a<=b.
      Should return b-a >= 0.
   3. Call Convert() with the expression a<b.
      Should return b-a-1 >= 0.
   4. Call Convert() with a non-arithmetic
      expression e. Should return e.
NOTES:
   Should repeat these tests using the NOT
   operator.

**** FUNCTION 3.2 - Rid-() ****

FUNCTION DESCRIPTION:
   Given an expression, converts all minus signs to
   plus signs.
DRIVER:
   Calls Rid-() with an expression(e).
TEST CASES:
   1. E contains no minuses. Return e.
   2. Call Rid-() with a-b=0. Return a+(-1*b)=0.
   3. Call Rid-() with -b. Return (-1*b).

**** FUNCTION 3.3 - Dist() ****

FUNCTION DESCRIPTION:
   Applies distributive laws to the expression. After
   completion, all plus symbols are outside all
   multiplication symbols.
DRIVER:
   Calls Dist() with an expression(e).
TEST CASES:
   Call Dist() with a variety of arithmetic expressions.

**** FUNCTION 3.4 - Rid+*() ****
```

```
FUNCTION DESCRIPTION:
   Converts predicates to a sum of products form, returning
   pairs.
DRIVER:
   Calls Rid+*() with a distributed expression(e).
TEST CASES:
   Call Rid+*() with results from the tests run in
   3.2. Should return a list of pairs, where the elements
   of each pair are to be multiplied together and the
   results summed.

**** FUNCTION 3.5 - Lexisort() ****

FUNCTION DESCRIPTION:
   Given a list, sorts the elements of the list
   lexicographically.
DRIVER:
   Calls Lexisort with a list(e).
TEST CASES:
   1. Call Lexisort() with a single element list.
      Should return the single element.
   2. Call Lexisort() with a multiple element list.
      (see below).
NOTES:
   Test 2 should be run using combinations of ids, numbers,
   expressions, etc. They are sorted such that all numbers
   are first, then ids, then expressions.

**** FUNCTION 3.6 - Docoeff() ****

FUNCTION DESCRIPTION:
   Takes the lexicographically ordered terms from
   Function 3.5 and combines the constant(numeric)
   terms to produce a single numeric coefficient.
DRIVER:
   Calls Docoeff() with a lexicographically ordered
   list of terms(term).
TEST CASES:
   1. Terms contain multiple numeric terms. Combine
      into a single numeric term.
```

2. Terms contains no numeric terms. Add the number 1 to the front of the list.

**** FUNCTION 3.7 - Collect() ****

FUNCTION DESCRIPTION:
Given a list of terms, combines all like terms and removes terms with a zero coefficient.
DRIVER:
Calls Collect() with a list of terms(e).
TEST CASES:
1. E contains no like terms. Return e.
2. E contains like terms. Return e with like terms combined.
3. E contains terms with zero coefficients. Return e with such terms removed.

**** FUNCTION 3.8 - Insert0() ****

FUNCTION DESCRIPTION:
Adds a zero coefficient to lists without a coefficient.
DRIVER:
Calls Insert0() with a list of terms(e).
TEST CASES:
1. E has a coefficient. Return e.
2. E has no coefficient. Return (0 e).

**** FUNCTION 3 - Normal_exp() ****

FUNCTION DESCRIPTION:
Normalizes an expression.
DRIVER:
Calls Normal_exp() with an expression(e).
TEST CASES:
1. E is an arithmetic expression. Return the normalization of e.
2. E is non-arithmetic. Return e.
NOTES:
Test 1 should return normalized expressions for the following operators: =, ~=, <, >, <=, >=. Also, all

```
   of these with the NOT operator in front of them.
```

---

```
********** FUNCTION 4: InList() *****************
TEST FUNCTION NAME: InList()
CALLED BY: extended simplifier
PURPOSE: Checks an expression to see if in a list.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a normalized expression and a list of
   norm. expr, checks to see if the non-numeric
   portion of the expression is in the list.
DRIVER:
   Calls InList() with a expression(e) and a list(l).
TEST CASES:
   1. E is in l. Return true.
   2. E is not in l. Return nil.
NOTES:
   Remember that this function is ignoring the
   numeric coefficient in this comparison.

   The functions Grtconst(), Greconst(), Lstconst(),
   Lseconst(), Equconst() all compare the numeric
   coefficients of two normalized expressions.
```

---

```
********** FUNCTION 5: add_expr() *****************
TEST FUNCTION NAME: add_expr()
CALLED BY: simplify module(basic component)
PURPOSE: Adds an expression to a simple list.
INTERNAL FUNCTIONS CALLED: eDelete(), eInclude()

**** FUNCTION 5.1 - eDelete() ****

FUNCTION DESCRIPTION:
   Removes an expression from a list.
DRIVER:
   Calls eDelete() with an expression(e) and a
```

```
   list(L).
TEST CASES:
   1. E is in L. Return L with e removed.
   2. E is not in L. Return L.
NOTES:
   eInclude() adds an expression to a list.

**** FUNCTION 5 - add_expr() ****

FUNCTION DESCRIPTION:
   Add_expr() adds an expression to the current
   state predicate. If the addition causes a
   modification to the state predicate, then the
   expression is added to the proper simplification
   list and the list is returned. A non-null return
   value indicates to the caller that Rebuild_SP()
   needs to be called to rebuild the state predicate
   b/c the contents have been modified. If the expr
   is not in any of the simplification lists, the
   function returns nil, indicating to the caller
   that the expression is unique and can be simply
   'ANDed' to the state predicate.
DRIVER:
   Initializes the four simplification lists. Calls
   add_expr() with an expression to be added(e).
TEST CASES:
   1. E is unique. Return nil.
   2. E is redundant. Ignore it.
   3. E is more specific information. For example,
      we know x >= 0. Now e states x = 0. We need
      to remove x >= 0 and add x = 0.
```

---

```
********** FUNCTION 6: Rebuild_SP() *****************
TEST FUNCTION NAME: Rebuild_SP()
CALLED BY: simplify module(basic component)
PURPOSE: Rebuilds state predicate from normal form
INTERNAL FUNCTIONS CALLED: Pos-neg(), Insert*(),
   Insert+-(), Rid*1(), undo_exp(), Revert()
```

```
**** FUNCTION 6.1 - Pos-neg() ****

FUNCTION DESCRIPTION:
   Moves all subexpressions with a positive
   coefficient to the beginning of the list.
DRIVER:
   Calls Pos-neg() with a list of normalized
   expressions(e).
TEST CASES:
   1. E contains no terms with positive coeff.
      Return e.
   2. E contains such terms. Return e with
      positive terms in the front of the list.

**** FUNCTION 6.2 - Insert*() ****

FUNCTION DESCRIPTION:
   Takes the list of pairs in sum of product form
   and insert multiplication symbols, putting the
   pairs in a binary form.
DRIVER:
   Calls Insert*() with a list of terms(e).
TEST CASES:
   1. E contains one term. Return the term.
   2. E contains an even number of terms.
      Return them multiplied together.
   3. E contains an odd number of terms.
      Return them multiplied together.

**** FUNCTION 6.3 - Insert+-() ****

FUNCTION DESCRIPTION:
   Converts a list containing a series of
   multiplications to show their sum. All
   +'s and -'s are on the outside of the
   final expression.
DRIVER:
   Calls Insert+-() with a product list.
TEST CASES:
```

```
   1. Call with ((3*3)(2*4)).
      Return (((3*3) + (2 * 4))).
   2. Call with ((3*3)(-2*4)).
      Return (((3*3) - (2*4))).

**** FUNCTION 6.4 - Rid*1() ****

FUNCTION DESCRIPTION:
   Searches list of terms, removing all
   coefficients of value 1 or -1.
DRIVER:
   Calls Rid*1() with a list of terms(e).
TEST CASES:
   1. E contains no coeffs of 1 or -1.
      Return e.
   2. E contains coeffs of 1. Return e
      and remove coeffs of 1.
   3. E contains coeffs of -1. Return e
      and replace -1's with minus signs.

**** FUNCTION 6.5 - undo_exp() ****

FUNCTION DESCRIPTION:
   Calls functions 5.1 - 5.4 to convert a
   normalized expression to a readable form.
DRIVER:
   Calls undo_exp() with a normalized exp(e).
TEST CASES:
   Test cases involve normalized expressions
   to test various test tested for functions
   5.1 - 5.4.

**** FUNCTION 6.6 - Revert() ****

FUNCTION DESCRIPTION:
   Uses Revert() to convert all normalized
   expressions in a list to a readable form.
   After converting the main part of the
   expression, the numeric coefficient is
   added in.
```

```
DRIVER:
   Calls Revert() with an operator(O) and
   a list of normalized terms(L).
TEST CASES:
   Should run tests from 5.5 with different
   operators and different numeric coeffs.

**** FUNCTION 6 - Rebuild_SP() ****

FUNCTION DESCRIPTION:
   Builds the state predicate from the four lists
   mentioned earlier by 'un-normalizing' each
   expression and conjoining them together.
DRIVER:
   Initializes four lists; calls Rebuild_SP().
TEST CASES:
   Tests should be run with many diff. combinations of
   values in each list.
NOTES:
   The internal functions that are called perform
   the 'denormalization'.
```

---

```
********** FUNCTION 7: Simplified() *****************
TEST FUNCTION NAME: Simplified()
CALLED BY: simplify module(basic component)
PURPOSE: Returns a simplified form of an expr.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given an expression, uses the normalization
   process to simplify the expression as much
   as possible. This includes consolidating like
   terms, etc. A flag is set so that the terms
   are rearranged as little as possible. Also,
   the function looks at a global flag, the
   SIMPLIFY flag. If it is true, then it performs
   the simplification. Otherwise, it returns.
DRIVER:
```

```
   Set SIMPLIFY flag to true. Call Simplified()
   with an expression(e).
TEST CASES:
   Test cases should be run with various unsimplified
   arithmetic expressions. For example,
   ‘i + 1 = j + 2’ should be return as ’i = j + 1’.
   Non-arithmetic expressions should be returned
   untouched.
```

---

```
********** FUNCTION 8: E_thm_prvr() *****************
TEST FUNCTION NAME: E_thm_prvr()
CALLED BY: simplify module(basic component)
PURPOSE: Attempts to reduce an expression to true or false.
INTERNAL FUNCTIONS CALLED: add_to_list(), E_Simp()

**** FUNCTION 8.1 - add_to_list() ****

FUNCTION DESCRIPTION:
   Adds an expression to one of the simplification
   lists.
DRIVER:
   Calls add_to_list() with an expression.
TEST CASES:
   Test cases should ensure that for all arithmetic
   expressions, they are normalized and placed in
   the proper lists. Non-arithmetic expressions are
   simply added to the non simple list.

**** FUNCTION 8.2 - E_Simp() ****

FUNCTION DESCRIPTION:
   Uses the simplification lists to attempt to
   reduce an expression to true or false.
   For example, given an expression ‘a > 0’
   if ‘a<=0’ is in the non negative list, then
   the expression is false.
DRIVER:
   Initializes the simplification lists. Calls
```

```
   E_Simp() with an expression(e).
TEST CASES:
   A variety of test cases should be run to
   test that each possible reduction is found.
NOTES:
   Remember that expressions are stored in the
   simplification lists normalized. E is not
   normalized upon entry to E_Simp(); it is
   normalized within the function.

**** FUNCTION 8 - E_thm_prvr() ****

FUNCTION DESCRIPTION:
   Given an expression, attempts to use info in
   the state predicate to simplify the expression
   to true or false. E_thm_prvr() actually does
   some preliminary trivial checks and calls
   add_to_list() and E_Simp() to do the reduction.
DRIVER:
   Calls E_thm_prvr() with a list of expressions(L1)
   corresponding the the contents of the state
   predicate, and an expression to be proved(L2).
TEST CASES:
   1. L1 is 'true'. Return L2.
   2. L1 is non-trivial and L2 reduces to true.
   3. L1 is non-trivial and L2 reduces to false.
   4. L1 is non-trivial and L2 doesn't reduce.
```

---

#### 2.5.7.3 The Basic Simplifier

The basic simplifier consists of a routine to break expressions into component parts, a reduction routine (to apply the reduces_to operator), and a collection of routines that are specialized to apply to a particular operator. The testing of this sub-component involves testing the traversal routines, testing the specialized reduce functions, and testing the other miscellaneous functions.

---

```
********** FUNCTION 9: theorem_pr() *****************
TEST FUNCTION NAME: theorem_pr()
CALLED BY: evaluator and driver modules
PURPOSE: Asks the user if an expression is true or false.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   This is the stub that eventually will be the
   interactive theorem prover. Right now it simply
   asks the user to respond true false or neither.
   The predicate argument is not used. The only
   two things that need to happen now are: the
   user's response is returned, and the operation
   mode upon entering the function is restored.
DRIVER:
   Sets up the restricted commands.
   Calls theorem_pr() with a predicate(predicate).
TEST CASES:
   1. Respond true. Return true.
   2. Respond false. Return false.
   3. Respond neither. Return neither.
```

---

```
********** FUNCTION 10: assume() *****************
TEST FUNCTION NAME: assume()
CALLED BY: evaluator module
PURPOSE: Adds an assumed pred to the state pred.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   This function adds an expression to the state
   predicate. It uses the extended simplifier to
   make sure the simplest form of the state pred
   is returned. If the expression is false or
   causes a contradiction with the state predicate,
   it is not added.
DRIVER:
   Sets initializing_state global variable and
   state predicate. Calls assume() with an exp(e).
```

```
TEST CASES:
   1. Initializing_state is true and e is false.
      Warning msg and e is not added.
   2. Initializing_state is false and e is false.
      Warning msg and e is not added.
   3. State predicate is true. Set state pred
      equal to e.
   4. E is true. Return.
   5. E is non-trivial and unique. Add e to
      state predicate.
   6. E is non-trivial and cause modifications
      to state predicate. Produce simplest state
      predicate.
NOTES:
   Tests 5 and 6 should be run with a variety of
   combinations of state predicate and expression.
```

---

```
********** FUNCTION 11: Simplify() *****************
TEST FUNCTION NAME: Simplify()
CALLED BY: driver and evaluator modules
PURPOSE: traverses expressions for simplification.
INTERNAL FUNCTIONS CALLED: normalize(), traverse(),
   traverse_down()

**** FUNCTION 11.1 - traverse_down() ****

FUNCTION DESCRIPTION:
   Traverse_down() recursively breaks an expression
   down, reduces its parts and puts them back
   together. Most expressions it simply passes on
   to the reduce-expr function, but some it must
   handle specially. For example:
     - quantified expr: the body of the expr is
       traversed.
     - Conditional exp: Each clause is traversed.
     - New or bound var ref: ignored. They aren't trav.
DRIVER:
   Calls traverse_down() with a traversal function
```

```
   (local_trav) and an expression(l).
STUBS:
   The argument local_trav should simply echo its
   input.
TEST CASES:
   1. Call function with a simple expression.
   2. Call function with a quantified expression.
   3. Call function with a conditional expression.
   4. Call function with a bound var ref.

**** FUNCTION 11.2 - traverse() ****

FUNCTION DESCRIPTION:

DRIVER:
   Calls traverse() with a reduce function(reduce_fnctn)
   and an expression(e).
STUB:
   reduce_fnctn echoes its input.
TEST CASES:
   Tests should be run using a variety of expressions
   to ensure that the breakdown is being done properly.
   Can use cases from function 11.1.
NOTES:
   Normalize() simply records the expression argument
   and calls traverse() to simplify it. If the result
   is different than the original expression, the
   expression is sent through the reduction process
   again. This is repeated until the expression is
   unchanged.

**** FUNCTION 11 - Simplify() ****

FUNCTION DESCRIPTION:
   Simplify() uses its internal functions to traverse
   an expression in a 'left to right - bottom up'
   fashion.
DRIVER:
   Calls Simplify() with an expression(e).
STUBS:
```

```
   reduce_fnctn echoes its input.
TEST CASES:
   Can run the same tests here that were run for
   function 11.2.
NOTES:
```

---

```
********** FUNCTION 12: it_equal() *****************
TEST FUNCTION NAME: it_equal()
CALLED BY: simplify module(basic simplifier)
PURPOSE: compares two terms for equality.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given two expressions, compares them for equality.
   If they are equal, returns true. Otherwise, returns
   nil.
DRIVER:
   Calls it_equal() with two terms(x,y).
TEST CASES:
   1. X and y are numbers. Return comparison.
   2. X and y are expressions. Return comparison.
   3. X and y are constants. Return comparison.
NOTES:
   There are many other similar utility functions
   used within the simplify module:
   it_nequal() - test for inequality.
   it_member() - test for set membership.
   it_notmem() - test for set exclusion.
   it_dedup() - test for duplicate items in a list.
   pos-and-neg-in() - test for a term and its
     negation in the same list.
   associate() - removes unnecessary parens levels
     in an expression.
   normal-sort() - sorts a list of terms
     lexicographically.
```

---

```
********** FUNCTION 13.1: red-expr() *****************
```

TEST FUNCTION NAME: it_equal()
CALLED BY: internal
PURPOSE: Compares two expressions for equality.
INTERNAL FUNCTIONS CALLED: All reduce functions for
   operators. Operators that have a specialized
   reduce function include: equal, not equal, not,
   and, or, member, not member, union, implies,
   forall, exists, set type, plus, differ, times,
   cond, set_diff, concat, listlen, gt, lt, ge, le,
   ref.

**** FUNCTION 13.1: red_.EQ() ****

FUNCTION DESCRIPTION:
   Attempts to simplify equality expressions to
   true or false.
DRIVER:
   Initializes state pred; calls red_.EQ() with
   an equality expression(e).
TEST CASES:
   1. E is an equality of two numbers.
   2. E is an equality of two constants.
   3. E can be shown true using the state pred.
   4. E cannot be shown true using the state pred.
NOTES:
   Tests 2 - 4 involve the use of the extended simplifier.
   This is the nature of all testing involving the
   specialized functions. For each function, test cases
   must be generated for all basic simplification cases
   as well as cases which require the use of the
   extended simplifier.

**** FUNCTION 13 - red-expr() ****

FUNCTION DESCRIPTION:
   Routes reduce expressions to the proper function.
DRIVER:
   Calls red-expr() with an expression(e).
TEST CASES:
   All tests that were run on specialized reduce

```
   functions before should be run again.
NOTES:
   red-shell() is simply an entry point to
   red-expr().
```

---

### 2.5.8 The Evaluator Module

The evaluator module determines all states resulting from state transitions. In doing so, it evaluates all expressions. Figure 2.3 shows a breakdown of the evaluator module. There are three entry points to the evaluator, one for evaluating predicates, one for state initialization, and one for evaluating transition exit predicates. In testing the evaluator module, first the evaluator is tested, then the special functions for evaluating Aslan operators are tested, then the state initialization and exit predicate evaluation functions are tested.

---

```
********** FUNCTION 1: eval_condition() *****************
TEST FUNCTION NAME: eval_condition()
CALLED BY: driver module
PURPOSE: evaluates a predicate.
INTERNAL FUNCTIONS CALLED: evaluator()

**** FUNCTION 1.1 - evaluator() ****

FUNCTION DESCRIPTION:
   The evaluator is called if we are not initializing
   state. The evaluator attempts to reduce the expression
   to true or false using the simplifier. If this fails,
   it then asks the user to resolve the expression.
DRIVER:
   Calls evaluator() with
   an expression(term) and a message(msg) - a prompt
   given to the user if the expression cannot be
   resolved.
STUBS:
   asltest_pp(), assume(), sub_old_values() simply
   echo/return their input.
```

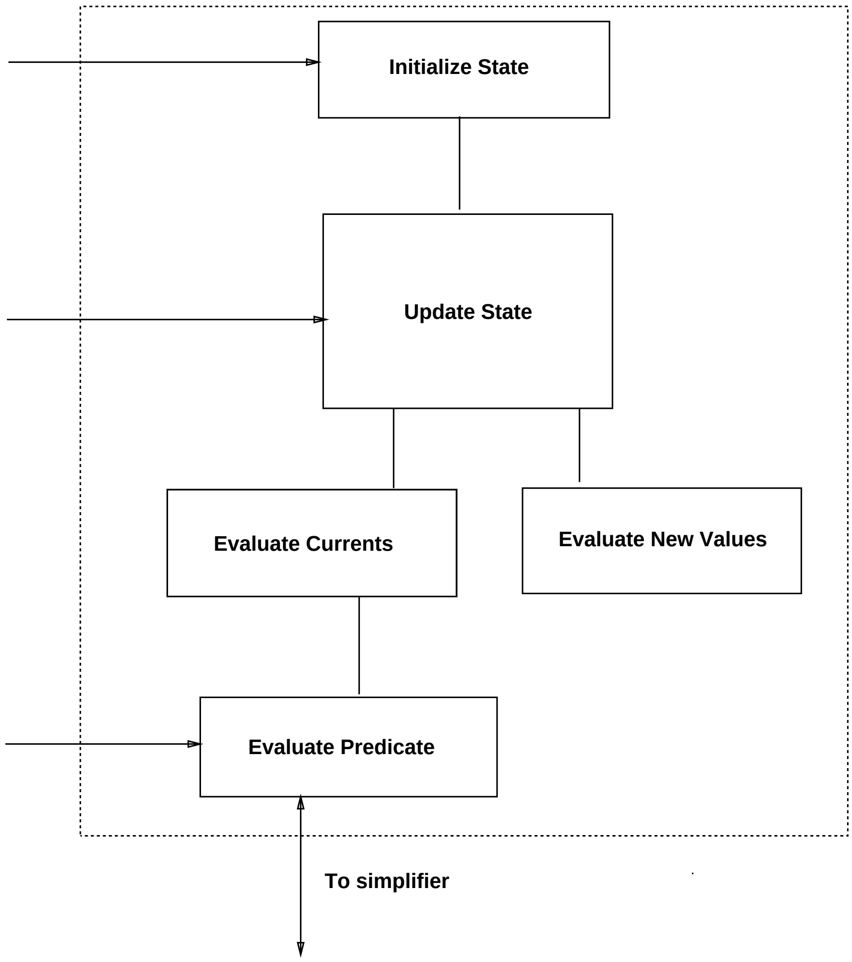


Figure 2.3 Aslantest Evaluator Module

```
   Simplify(), theorem_pr(), and command_interpreter()
   perform as necessary in each test.
TEST CASES:
   1. Simplifier returns true. Return true.
   2. Simplifier returns false. Return false.
   3. Simplifier returns expression, theorem_pr()
      returns true. Return true.
   4. Simplifier returns expression, theorem_pr()
      returns false. Return false.
   5. Simplifier returns expression, theorem_pr()
      returns neither, command_interpreter() returns
      true. Return true.
   6. Simplifier returns expression, theorem_pr()
      returns neither, command_interpreter() returns
      false. Return false.

**** FUNCTION 1 - eval_condition() ****

FUNCTION DESCRIPTION:
   Eval_condition() takes an expression. If initializing
   state, it simplifies the expression and adds it to
   the state predicate. If not, it passes it on to the
   evaluator. If the expression is a conjunction, then
   each conjunct is passed to the evaluator separately.
   If a conjunct is false, false is immediately returned.
DRIVER:
   Initializes global variable initializing_state.
   Calls eval_condition() with an expression(term).
STUBS:
   asltest_pp(), assume(), sub_old_values() simply
   echo/return their input.
   Simplify(), theorem_pr(), and command_interpreter()
   perform as necessary in each test.
TEST CASES:
   1. Initializing_state is true. Term is added to
      state predicate.
   2. Initializing_state is false. Term is a conjunction
      and all conjuncts are true. Return true.
   3. Initializing_state is false. Term is a conjunction
      and a conjunct is false. Return false upon evaluating
```

```
      the false conjunct.
   4. Initializing_state is false and term is not
      a conjunction.
NOTES:
   Test 4 should be run for each of the six cases above.
```

---

```
********** FUNCTION 2: update_state() *****************
TEST FUNCTION NAME: update_state()
CALLED BY: driver module
PURPOSE: evaluates new state.
INTERNAL FUNCTIONS CALLED:
   eval_*() {eval_.OR, eval_.EQ, etc}
   eval_new_*() {eval_new_.AND, eval_new_.BEC, etc.}
   explicitly_evaluable(), eval_implicit_expr(),
   make_consistent()

**** FUNCTION 2.1 - eval_.AND() ****

FUNCTION DESCRIPTION:
   Eval_.AND() evaluates conjunctions containing only
   current value operands. It does so by evaluating
   each conjunct separately and examining the results.
   It returns true, false, or a simplified version of
   the conjunction.
DRIVER:
   Calls eval_.AND() with a list of conjuncts.
STUBS:
   Eval_expr() returns its input.
   Include all stubs used by eval_condition().
TEST CASES:
   1. List of conjuncts contains all true predicates.
      Return true.
   2. List of conjuncts contain one false predicate.
      Return false.
   3. List of conjuncts contain one true predicate.
      Return conjunction with true predicate removed.
   4. List of conjuncts contain a new value reference.
      Return list of conjuncts.
```

```
NOTES:
   The eval_*() functions all operate in very much the
   same manner. They evaluate operations for a particular
   operator. If the operation contains a new value ref.
   it is returned. If not, it is evaluated. There are
   eval_*() routines for the following operators:
   equals, and, or, conditionals, not, the reduce operator,
   exists, unique, forall, implies, greater than, less
   than, not equals, greater than or equal, less than
   or equal, set membership.

   eval_expr() is the routing function that directs expressions
   to the proper eval_* function.

**** FUNCTION 2.2 - eval_implicit_expr() ****

FUNCTION DESCRIPTION:
   This function takes an expression that cannot be evaluated
   and adds it to the state predicate. In doing so, it replaces
   all new value variable references with fresh symbolic values.

   If the input is the keyword "FALSECOND" then the function
   simply returns. This keyword indicates the original
   expression was a conditional for which no clause evaluated
   to true.
DRIVER:
   Calls eval_implicit_expr() with an expression(e).
STUB:
   get_next_value() returns an symbolic value representation.
   add_wff() returns the conjunction of its inputs.
TEST CASES:
   1. E is FALSECOND. Return.
   2. E contains new value references. Add to state pred.
NOTES:
   This function actually calls a couple of functions to
   complete its task. add_to_new_pred() adds an expression
   to the temporary holder new_pred. sub_new_values()
   replaces new value references with symbolic values.
   These functions are sufficiently simple to test as a
   group.
```

```
**** FUNCTION 2.3 - explicitly_evaluable() ****

FUNCTION DESCRIPTION:
   This function returns true if the expression is considered
   to be explicitly evaluable. It is used with quantified
   expressions. If the expression is evaluable, then it
   is processed. If not it is added to the state predicate.
   Quantified expressions that contain combinations of
   the following operators are considered evaluable:
   forall, exists, and, conditionals, new value, not, and
   equals(without new value references on both sides of
   the operator).
DRIVER:
   Calls explicitly_evaluable() with a quantified expression(e).
TEST CASES:
   Test cases should be run with various combinations of
   evaluable and unevaluable operators.

**** FUNCTION 2.4 - eval_new_.EQ() ****

FUNCTION DESCRIPTION:
   Evaluates equality expressions containing a new value
   reference. Uses update_value() from the variables
   module to do the actual update. This function's job
   is to provide update_value() with the right information.
DRIVER:
   Calls eval_new_.EQ() with a equality expression(e).
STUBS:
   update_value() echoes its arguments.
   fix_non_boolean_COND(), eval_new_expr() echo their
   input.
TEST CASES:
   1. Call function with a = b'.
   2. Call function with b' = a.
   3. Call function with a(c) = b'.
   4. Call function with b' = a(c).
   5. Call function with a = b.
   6. Call function with (a = if b' then c' else d').
NOTES:
```

```
   The other eval_new_*() functions do similar evaluations.
   Their job is to parse the expression looking for variable
   updates. Operations which have eval_new_*() functions
   include: and, or, eq, becomes, if and only if, forall,
   exists, unique, new value, ref(parameterized boolean
   variables), not, set membership, conditionals.

   eval_new_expr() is the routing function for this collection
   of routines.

**** FUNCTION 2.5 - make_consistent() ****

FUNCTION DESCRIPTION:
   This function is used whenever a parameterized variable
   is updated both explicitly and implicitly in the same
   predicate. In such situations, the explicit conditional
   value must be extended to state that if none of the
   stated conditionals hold, the variable has an undefined
   value.
DRIVER:
   Initializes a parameterized variable and its values.
   Calls make_consistent() with an id(id).
STUBS:
   Simplify() returns its argument.
   NOTIZE() returns the negation of its argument.
   make_value(), get_params(), get_var and set_var() perform
   as necessary in each test.
TEST CASES:
   Set up the variable id with a list of values. The returned
   value should be the same list with an additional clause
   which is the negation of all the other clauses. Corresponding
   to that additional clause is a fresh symbolic value.

**** FUNCTION 2.0 - update_state() ****

FUNCTION DESCRIPTION:
   Given an expression, uses that expression to determine
   the new state. First, uses eval_expr() to evaluate all
   current value expressions. Then uses eval_new_expr()
   to determine new variable values. Then, if initializing
```

```
   state it returns. If evaluating an exit predicate, updates
   all variable values.
DRIVER:
   Sets up variable lists and state predicate. Initializes
   initializing_state system variable. Calls update_state()
   with an expression(expr) and a flag(flag) that tells
   whether or not the expression is an exit predicate for
   a transition.
STUBS:
   sub_old_values() returns its input.
   set_var(), set_value(), assume() echo their arguments.
   make_value(), get_new_value(), get_var(), get_genid()
   init_value() perform as needed in each test.

   All other stubs listed for internal functions also
   apply.
TEST CASES:
   This is the hardest function to isolate and test. It
   requires a great deal of setup just to run a single
   test case.
```

---

```
********** FUNCTION 3: initialize_state() *****************
TEST FUNCTION NAME: initialize_state()
CALLED BY: driver and control modules
PURPOSE: initializes state
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Given a predicate, uses it to initialize the state.
   Resets variables and calls update_state() to evaluate
   the predicate.
DRIVER:
   Calls initialize_state() with an expression(expr).
STUBS:
   fix_BOUND(), make_NEW(), expand_wff() all return
   their arguments.
TEST CASES:
   This function is simply an extension of update_state()
```

```
   that resets some system variables. The same expressions
   used to test the update_state() function can be used
   to test initialize_state().
```

---

### 2.5.9 The Driver Module

The driver module contains the functions that execute user commands. Each driver consists of two parts, a top half and a bottom half. The top half retrieves any additional information needed to execute the command. The bottom half actually executes the command. All drivers are called by the control module.

---

```
********** FUNCTION 1: check_flag() *****************
TEST FUNCTION NAME: check_flag()
PURPOSE: sets value of the execution flags.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   If the user provides the keywords 'on' or 'off',
   the function will reset the flag to that value.
   If no keyword is provided the current value of
   the flag is displayed.
DRIVER:
   Calls check_flag() with the execution flag to
   be reset(flag).
STUBS:
   read_optional() performs as specified in each
   test.
TEST CASES:
   1. Read_optional() returns nil. Display value
      of flag.
   2. Read_optional() returns on. Reset flag to
      on.
   3. Read_optional() returns off. Reset flag to
      off.
NOTES:
   Tests should be run using each value of flag:
   simplify, verbose, and autosave.
```

```
   This is an example of a driver with a trivial
   top-half. Other such drivers include:
   restart() - resets the system.
   aslantest_help() - provides help messages.
   display_state() - lists state/start/result pred.
   display_states() - lists saved states.
   display_path() - lists execution path.
   list_spec() - lists the specification.
   list_transitions() - lists current transitions.
   display_vars() - lists current var values.
   get_cmd_file() - resets input for file command.
```

---

```
********** FUNCTION 2: load_new_spec() *****************
TEST FUNCTION NAME: load_new_spec()
PURPOSE: loads in a new specification.
INTERNAL FUNCTIONS CALLED: load_new()

**** FUNCTION 2.1 - load_new() ****

FUNCTION DESCRIPTION:
   Given a filename, attempts to locate the file and
   feed it into the parser to create new aslantest.lisp
   aslantest.pxp files. Then resets some values and
   calls the top level control routine to load in the
   specification.
DRIVER:
   Calls load_new() with the filename of a file containing
   an Aslan specification(f).
STUBS:
   flush_attributes() echoes entry message.
   aslantest_top_level() echoes entry message.
TEST CASES:
   1. F is a valid accessible specification. Generate a
      new aslantest.lisp and aslantest.pxp file.
   2. F is not accessible. Return with error msg.
   3. Parser cannot parse F. Return with error msg.
```

```
**** FUNCTION 2 - load_new_spec() ****

FUNCTION DESCRIPTION:
   If the user provided a filename with the command,
   call load_new(). If not, prompt the user for a filename.
   If one is entered call load_new() otherwise return.
DRIVER:
   Calls load_new_spec().
STUBS:
   read_optional() performs as described in each test.
TEST CASES:
   1. read_optional() returns a filename. Call load_new().
   2. read_optional() returns nil. Prompt for filename.
      If no filename entered return. If filename entered
      call load_new().
NOTES:
   Tests 1 and 2 should be run using filenames from tests
   1 - 3 above.
```

---

```
********** FUNCTION 3: exit_check() *****************
TEST FUNCTION NAME: exit_check()
PURPOSE: exits program.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   Function verifies that user actually wants to quit.
   If the user responds yes, the program exits. If not,
   the function asks the user if they wish to load in
   a new specification. If yes, then the filename is
   retrieved and load_new() is called to translate the
   new spec. If not, the function returns.
DRIVER:
   Calls exit_check().
STUBS:
   Command_interpreter() performs as described.
TEST CASES:
   1. User says yes to quit. Exit program.
   2. User says no to quit; no to load new spec.
```

```
   Return.
3. User says no to quit; yes to new spec.
   Retrieve new spec file name and call load_new().
```

---

```
********** FUNCTION 4: save_state() *****************
TEST FUNCTION NAME: save_state()
PURPOSE: saves the current state.
INTERNAL FUNCTIONS CALLED: None

FUNCTION DESCRIPTION:
   The parameter autoflag is used to indicate a state
   saved by the autosave option and an internal state
   saved by the system. If autoflag = 1 it is the state
   saved immediately upon setting the autosave flag.
   If autoflag = internal it is an internal state being
   saved. If autoflag is not one of these and is non-null,
   then it is a autosave state. If autoflag is null,
   then it is a user requested save. If the user provided
   a comment, it is used. If not, the user is prompted
   for one.
DRIVER:
   Initializes state_num and states_list.
   Sets trans_name and trans_args.
   Calls save_state with a flag value(autoflag).
STUBS:
   get_explicit_values(), get_value(), get_var()
   echo their input.
   read_optional() performs as described.
TEST CASES:
   1. Autoflag = 1. Save state with comment:
      '(AUTO): After autosave flag set'.
   2. Autoflag = internal. Save state with comment:
      'internal state save'.
   3. Autoflag = 2. Save state with comment:
      '(AUTO): After (trans_name trans_args)'.
   4. Autoflag = nil. Read_optional() returns a
      comment. State is saved.
   5. Autoflag = nil. Read_optional() returns nil.
```

```
      User is prompted for comment. State is saved.
NOTES:
   For test 3, the number 2 is arbitrary. It can
   be anything except 1 or internal.

   Restore_state() and remove_state() are very
   similar in execution.
```

---

```
********** FUNCTION 5: exec_transition() *****************
TEST FUNCTION NAME: exec_transition()
PURPOSE: executes a transition
INTERNAL FUNCTIONS CALLED: get_trans(), eval_excepts(),
   exec_trans()

**** FUNCTION 5.1 - get_trans() ****

FUNCTION DESCRIPTION:
   Parses a string corresponding to a transition and
   its arguments. It's result is to pull out the
   transition name, and produce a list of arguments
   in the internal form used by the executor. It
   uses form_args() to do this. This function will
   catch errors having to do with transition format.
   If all goes well, (id idlist) is returned to the
   caller where id is the name of the transition and
   idlist is the list of arguments.
DRIVER:
   Calls get_trans() with a transition id following
   by a parenthesized list of arguments(tinfo).
STUBS:
   form_args() returns the internal representation
   of its input.
   Is_space() returns true if the input is a space.
   Command() returns true if the input is non-white
   space.
TEST CASES:
   1. Tinfo is a valid transition call. Return the
      proper list.
```

```
   2. Tinfo consists of whitespace. Return nil.
   3. Tinfo has no transition id. Return nil.
   4. Tinfo has an incorrect number of parentheses.
      Return nil.
   5. Tinfo has illegal characters, such as # or @.
      Return nil.
NOTES:
   At the integration level, will want to run test 1
   with a number of different combinations.

**** FUNCTION 5.2 - eval_excepts() ****

FUNCTION DESCRIPTION:
   Given a list of except-exit pairs, evaluates them
   in order until an executable one is found.
DRIVER:
   Calls eval_excepts() with:
      t_id - transition id.
      args - user provided arguments.
      except_conds - list of except conditions.
      except_exits - list of corresponding exit predicates.
      mod_vars - list of modified vars for each exit.
      count - number of current except-exit being evaluated.
STUBS:
   fix_except(), sub_old_values(), sub_params(). fix_BOUND()
   return their input.
   eval_condition(), get_transition_args(), get_exc_exit()
   perform as necessary in each test.
   outlist() echoes its input.
   execute_exit() echoes its input.
TEST CASES:
   Test cases should be run to ensure the proper exit
   predicate is executed for the chosen except condition.
   Also that the count is correct.

   exec_trans() is very similar except it is simpler. It
   evaluates the entry condition; if true it executes the
   exit predicate. If false, and there are except-exit
   pairs it calls eval_excepts() to evaluate them.
```

```
**** FUNCTION 5.0 - exec_transition() ****

FUNCTION DESCRIPTION:
   Executes a transition. Gets the transition and arguments
   from the user, call get_trans() to parse them and
   calls exec_trans() to execute the transition.
DRIVER:
   Sets up a transition and its attributes.
   Calls exec_transition().
STUBS:
   read_optional() performs as described.
TEST CASES:
   1. read_optional() returns a transition w/ arguments.
   2. Read_optional() returns nil. User is prompted
      for and enters a transition id and arguments.
   3. Read_optional() returns nil. User is prompted
      for but enters no transition id. Return nil.
NOTES:
   Test data can be drawn from combinations of tests from
   above.
```

---

```
********** FUNCTION 6: check_() *****************
TEST FUNCTION NAME: check_()
PURPOSE: driver for check command.
INTERNAL FUNCTIONS CALLED: check_state(),
   check_cur_state(), check_result_state()

**** FUNCTION 6.1 - check_state() ****

FUNCTION DESCRIPTION:
   Presents an implication to the user for verification.
   Calls theorem_pr() to get user's response.
DRIVER:
   Calls check_state() with two predicates(s1, s2)
   where s1 -> s2.
STUBS:
   theorem_pr() returns true.
   asltest_pp() echos its input.
```

```
TEST CASES:
   There isn't much to test here. Just run it with
   a couple of expressions to make sure they are
   getting outputted.
   More testing should be done at the integration
   level.

**** FUNCTION 6.2 - check_cur_state() ****

FUNCTION DESCRIPTION:
   This function generates and attempts to verify
   the following proof obligation:
      state predicate -> invariant.
   It expands and attempts to reduce the invariant
   to true or false. If successful, it returns the
   result. If not, it calls check_state() to get
   the user's response.
DRIVER:
   Initializes the state predicate and invariant.
   Calls check_cur_state().
STUBS:
   fix_BOUND(), expand_wff(), and sub_old_values()
   return their input.
   Simplify() performs as described.
TEST CASES:
   1. Simplify() returns true. Respond with msg
      stating invariant is true.
   2. Simplify() returns false. Respond with msg
      stating invariant is false.
   3. Simplify() returns expression. Call
      check_state().
NOTES:
   The function check_result_state() performs the
   same duty using the result predicate instead
   of the invariant.

**** FUNCTION 6.0 - check_() ****

FUNCTION DESCRIPTION:
   The driver function for the check command. Checks
```

```
   the option provided by the user. If it is null or
   a ‘c’, calls check_cur_state(). If it is an ‘r’
   calls check_result_state(). Otherwise, gives an
   error message.
DRIVER:
   Initializes result predicate, invariant, and
   state predicate. Calls check_().
STUBS:
   read_optional() performs as described.
TEST CASES:
   1. Read_optional() returns nil, ‘c’, or ‘C’.
      Call check_cur_state().
   2. Read_optional() returns ‘r’ or ‘R’. Call
      check_result_state().
   3. Read_optional() returns anything else.
      Respond with error message.
```

---

```
********** FUNCTION 7: init_state() *****************
TEST FUNCTION NAME: init_state()
PURPOSE: initializes either the start or result predicate.
INTERNAL FUNCTIONS CALLED: get_expr(), get_pred_from_term(),
   get_state_from_file(), get_result_state(),
   get_start_state()

**** FUNCTION 7.1 - get_expr() ****

FUNCTION DESCRIPTION:
   Given a filename of a file containing an expression,
   feeds the expression into the translator for translation.
   The converted expression is returned.
DRIVER:
   Calls get_expr() with a filename(fname).
TEST CASES:
   1. The expression in fname is valid. Return its
      translated form.
   2. The expression in fname contains errors. Respond
      with an error message.
   3. The file isn’t accessible. Respond with an error
```

message.
NOTES:
The results of the parser are written into a temporary file called 'aslantest.in.spec_id' where spec_id is the name of the specification. This file should be destroyed upon exiting of the function.

**** FUNCTION 7.2 - get_pred_from_term() ****

FUNCTION DESCRIPTION:
This function is invoked when the user is entering a predicate from the keyboard. The user terminates the entering by typing a period on a line by itself. This function then writes the expression to a temporary file ('aslantest.out.spec_id') and precedes the expression in the file with the keyword 'symbolic_exec_call'. The function then calls get_expr() to retrieve the translated expression.
The flag argument is set to true if the call is coming from the graphical interface.
DRIVER:
Calls get_pred_from_term() with fromxflag set to nil.
STUBS:
read_line_cmd_file() returns a line of user input.
TEST CASES:
Enter various expressions. The key thing to look for is that the keyword 'symbolic_exec_call' is at the top of the created file. Also, that the period is being recognized for termination.
Also, the temporary file should be deleted.
NOTES:
get_state_from_file() is identical to this, except it reads the expression from a file. It checks to make sure the file is accessible, reads the expression a line at a time until a period is read, and loads it into a temporary file. It then calls get_expr().

**** FUNCTION 7.3 - get_start_state() ****

```
FUNCTION DESCRIPTION:
   Driver for getting the start predicate. Asks the
   user to enter source of the predicate, retrieves
   the predicate, and installs it.
DRIVER:
   Initializes initial conditions.
   Calls get_start_state() with the fromx flag set
   to nil.
STUBS:
   save_state(), restore_state(), error_restore(),
   remove_state() echo their input.
   initialize_state() performs as described.
TEST CASES:
   1. User enters default as source. Start predicate
      set to initial conditions.
   2. User enters file as source. Call get_state_from_file().
      Initialize_state() returns nil.
   3. User enters keyboard as source. Call
      get_pred_from_term(). Initialize_state()
      returns nil.
   4. User enters invalid option. Start predicate
      set to initial conditions.
   5. Initialize_state() 'throws' expression out.
      Original start predicate is restored and error
      message is presented.
NOTES:
     No matter what the outcome of the start predicate
   update, the internal state that was saved should be
   gone, and the count should be as if it had never
   occurred.

   get_result_state() is very much the same, but it is
   only retrieved; it is not evaluated until it is time
   to generate a proof obligation. Also, it has an
   internal function that checks the expression for
   reduces_to operators, which are illegal in a result
   predicate. That function is called has_REDUCE().

   The driver function add_to_state() is also very
```

```
   similar in functionality.

**** FUNCTION 7 - init_state() ****

FUNCTION DESCRIPTION:
   Driver for predicate initialization routines.
   If the user provides an 's', 'S' or null as
   an argument, the start predicate is initialized.
   If the user provides an 'r' or 'R' the result
   predicate is initialized.
DRIVER:
   Calls init_state().
STUBS:
   Read_optional() performs as described.
TEST CASES:
   1. User enters 's', 'S', or nil. Call
      get_start_state().
   2. User enters 'r' or 'R'. Call get_result_state().
   3. User enters something else. Respond with
      an error message.
```

---

### 2.5.10 The Control Module

The control flow handles routing of user requests and error messages. It contains two functions that simply assignment system variables initial values: init_gvars() and init_gvars2().

The error handling functions are very simple. Warning() prints the message provided as an argument. Error_restore() resets the temporary holders used when evaluating state transitions. Err() returns control to the executor() function.

---

```
********** FUNCTION 1: command_interpreter() *****************
TEST FUNCTION NAME: command_interpreter()
PURPOSE: Routes user requests.
INTERNAL FUNCTIONS CALLED: None
```

```
FUNCTION DESCRIPTION:
   Receives input from the input module and calls the
   corresponding driver function.
DRIVER:
   Calls command_interpreter().
STUBS:
   get_command() performs as described.
   aslantest_help() echos its input.
   Any driver function called echos its input.
TEST CASES:
   1. Get_command() returns a valid command. Call the
      appropriate driver function.
   2. Get_command() returns an invalid command. After
      three such returns, aslantest_help() is called.
   3. Get_command() is a valid command, but the command
      is currently restricted. Respond with error msg.
```

---

```
********** FUNCTION 2: aslantest_top_level() *****************
TEST FUNCTION NAME: aslantest_top_level()
PURPOSE: Handles traffic flow in system.
INTERNAL FUNCTIONS CALLED: Executor()

**** FUNCTION 2.1 - executor() ****

FUNCTION DESCRIPTION:
   Executor() is an infinite function, continually polling the
   input for user commands. There isn't much to test here. The
   real testing will be done when this is integrated with the
   input module.

**** FUNCTION 2 - aslantest_top_level() ****

FUNCTION DESCRIPTION:
   Starts up the system. Loads in the aslantest.lisp file,
   initializes state and key system variables, and calls
   the executor.
DRIVER:
   Calls aslantest_top_level().
```

```
STUBS:
   pp_init(), init_transition(), init_gvars(),
   initialize_state() echo their input.
TEST CASES:
   1. Aslantest.lisp is accessible.
   2. Aslantest.lisp is inaccessible. Respond with an error
      message.
```

---

# Chapter 3

# System Integration

# 3 System Integration

Figure 3.1 shows a picture of the system integration. Lines show important interconnections between modules. Integration is a six step process beginning with the library module and completing with the translator and initialize modules. The library module is isolated in the first step because of the interconnection of its sub-components and the importance of the library module to the remainder of the system. All modules interact with the library modules in some fashion. Because of this, it is important that the library module be tested as thoroughly as possible.

At all levels of integration, each function is examined. If stubs were used during its unit testing, then it must be re-tested at the integration level. For each stub used, if the function represented by the stub is currently integrated into the system replace the stub with the actual function and run the test cases again. Results should be examined to ensure the two functions are interacting correctly.

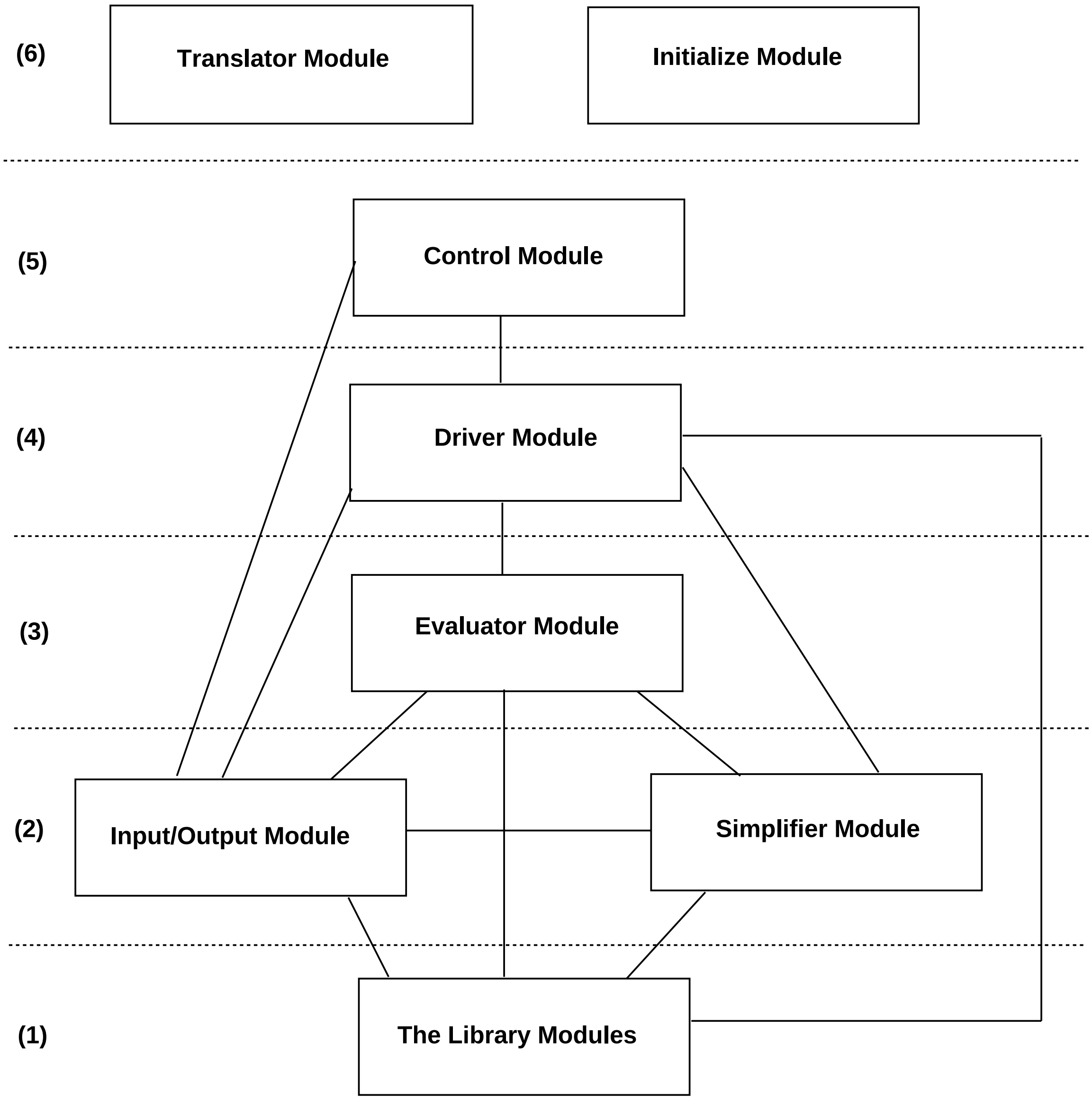


Figure 3.1 Aslantest Integration Scheme

# Chapter 4

# System Testing

# 4 System Testing

Once the system is fully integrated, system testing is performed to ensure the system is performing correctly as a unit. In testing the system, four different kinds of tests are run. In listing different kinds of tests, Pfleeger[3] includes the following four types:

1. *Documentation Tests:* ensure that the system performs as described in the system documentation.
2. *Recovery Tests:* Test how the system responds to errors during use.
3. *Human factors Tests:* Investigate requirements dealing with 'ease of use' aspects of the system. This group of tests is particularly important for the graphical interface.
4. *Volume Tests:* test how the system handles large amounts of data. In the case of ASLANTEST these tests examine the system's performance when dealing with large amounts of data. For the graphical interface, these test how the system deals with large amounts of data and consecutive commands delivered very quickly. Each of the system tests are outlined to be run with or without the graphical interface.

Although these categories do not guarantee a complete testing of the system, they do provide a good composite testsuite for Aslantest.

## 4.1 Documentation Testing

Documentation testing is very similar to functionality testing. Basically, each command is executed as described in the user's manual to ensure that the user's manual and the system are consistent. Tests not only make sure the commands perform as described, but also that the interface is consistent. For example, the following is the description of the 'quit' command contained in the Aslantest User's Manual[4]:

> "Exits the program and returns to UNIX. User is asked to confirm that they wish to exit the session. If he responds in the affirmative, the program exits. If the user responds no, he is asked if he wishes to load another specification. If yes, then a filename prompt is

provided and the new specification is loaded. If no, control returns to the original specification."

Tests should be run to ensure that the command proceeds as described and that the messages provided by Aslantest are consistent with those in the User's Manual.

Tests should also be run that cause the various messages described in the appendix of the manual to appear, to make sure they are consistent.

## 4.2 Recovery Tests

In recovery testing, the user is testing to see how the system handles error conditions. Each command should be run abnormally in an attempt to cause errors to see how the system will respond. Providing improper user input, bad predicates, commands at inappropriate moments are all error-causing scenarios. Error messages and responses should be as described in the Aslantest User's Manual[4].

## 4.3 Human Factors Tests

Human factors tests reveal how user-friendly the system is to use. Qualities that are looked for in these tests include:

- How intuitive are command keywords?
- How helpful are the system messages in providing the user with useful information on how to proceed?
- In error states, how descriptive are the error messages?
- Is the user's manual helpful in providing quick reference?

The best way to perform tests of this type is to invite users unfamiliar with the system to sit down and use the system. They can then report any problems or confusion experienced when using the system which will provide directions on where to improve in this category.

## 4.4 Volume Tests

For Aslantest, volume tests examine how well the system handles large and complex data. Performance as well as correctness are issues here. Although it is difficult to measure quantitatively, how quickly the system processes data is an important issue. Examples of volume tests include:

- Enter an extremely large and complex specification to be translated and executed. The medical specification(med2ls) is a good example.
- Enter an extremely long start predicate that requires expansion of defines, conversion of bound variables and nochange operators, and parameter substitution.
- Add a series of predicates to the state predicate. Add each predicate using a separate call to the addpred command. Check the performance as the predicate increases in size. The same kind of test can be run by adjusting the value of a variable, causing the simplifier to have to work on the value.
- Nest an undeclared constant deep within a predicate to make sure the types module locates and declares it.

# Chapter 5

# Regression Testing

# 5 Regression Testing

Regression testing is performed on a system in two situations. First, if an existing portion of the system is modified. Second, if a new module or function(s) is added to the system. The approach to regression testing is slightly different depending on which type of enhancement has been made.

If a modification is made to an existing portion of the system, then the modified function must be tested as completely as possible in isolation and then in consolidation with any functions it calls within its module. Then integration testing must be performed beginning at either the module containing the function or the lowest module containing a call to the function, whichever is lower in the integration heirarchy. If this is successful, relevant system tests should be run to ensure the external functionality of the system is still correct.

If a new module or function is added to the system, test cases for the module or function should be developed and documented, just as they have for the existing functions. Integration should begin at the lowest level of interaction and go all the way up. The addition of a new module may require an adjustment in the integration scheme. Any new test cases or approaches should be documented in the testplan. System tests should be developed and run which show that the new addition is performing to expectations. Also, existing system tests should be run on any portions of the system that interact with the new addition.

# References

[1] B. Auernheimer and R. Kemmerer, *ASLAN User's Manual*, Department of Computer Science, University of California at Santa Barbara, TRCS 84-10, April 1992.

[2] B. Auernheimer and R. Kemmerer, "Procedural and Nonprocedural Semantics of the ASLAN Formal Specification Language", In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 784-794, Hawaii, 1986.

[3] Shari Lawrence Pfleeger, *Software Engineering: The Production of Quality Software*, MacMillan Publishing Co., (New York, New York), 1987.

[4] Jeffrey G. Douglas, Aslantest User's Manual, Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA, TRCS93-12, July 1993.