Prototype Code

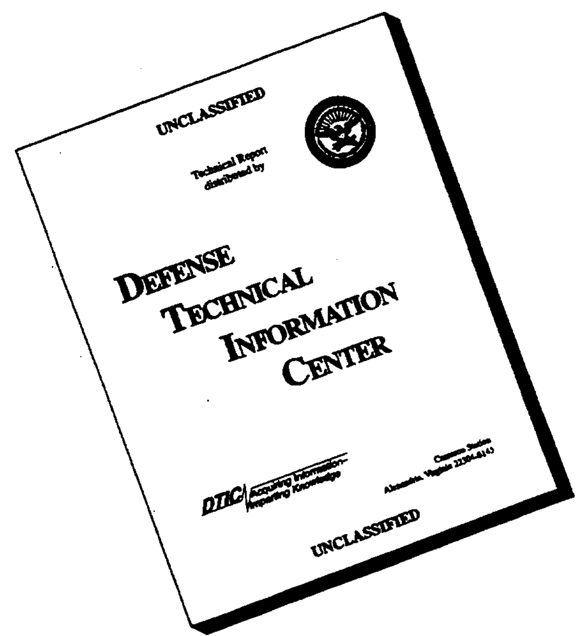
David Guaspari

N00014-95-C-0349
Attachment to:
CDRL A002

ODYSSEY RESEARCH ASSOCIATES

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

ORA TM-96-0019(Attachment A) 12 April 1996

Prototype Code

David Guaspari

N00014-95-C-0349
Attachment to:
CDRL A002

Prepared for:

Program Manager, Ralph F. Wachter
ONR 311
800 North Quincy Street
Arlington, VA 2221705660

Prepared by:

David Guaspari
Odyssey Research Associates, Inc.
301 Dates Drive
Ithaca, NY 14850-1326
(607) 277-2020

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

19960912 153

Contents

1 Outline **1**

2 The grammar of RSML **3**

3 Making `grammar.cc` **15**
- 3.1 The outline for grammar.cc . . . 15
- 3.2 The top-level noweb modules . . . 17
- 3.3 Assumptions and limitations . . . 18

4 Parsing preliminaries **19**
- 4.1 Tokens . . . 19
- 4.2 Precedence . . . 20

5 Collecting global information **21**
- 5.1 The variable `glob` . . . 21
- 5.2 Completing `glob` . . . 22
- 5.3 Traversal actions . . . 23

6 Attribute declarations **25**
- 6.1 Attributes of tokens . . . 25
- 6.2 Attributes for system . . . 26
- 6.3 Attributes for components . . . 26
- 6.4 Attributes for states . . . 27
- 6.5 Attributes for constants . . . 29
- 6.6 Attributes for variables . . . 30
- 6.7 Attributes for events . . . 33
- 6.8 Attributes for interface definitions . . . 34
- 6.9 Attributes for macros . . . 35
- 6.10 Attributes for functions . . . 36
- 6.11 Attributes for transition busses . . . 37
- 6.12 Attributes for predicates . . . 41
- 6.13 Attributes for conditions . . . 43

6.14 Attributes for expressions . . . 45
6.15 The various kinds of names . . . 46

7 Attribute definitions **50**
7.1 Attributes for system . . . 50
7.2 Attributes for components . . . 50
7.3 Attributes for states . . . 52
7.3.1 Auxiliary functions . . . 52
7.3.2 Definitions . . . 52
7.3.3 Bodies of auxiliary functions . . . 59
7.4 Attributes for constants . . . 59
7.5 Attributes for variables . . . 60
7.6 Attributes for events . . . 62
7.7 Attributes for interfaces . . . 63
7.8 Attributes for macros . . . 65
7.9 Attributes for functions . . . 65
7.10 Attributes for transitions and busses . . . 66
7.11 Attributes for predicates . . . 72
7.12 Attributes for conditions . . . 74
7.13 Attributes for expressions . . . 75
7.14 Attributes for names . . . 78

8 EVES **80**
8.1 Currently implemented . . . 80
8.2 Preliminaries . . . 81
8.3 Attribute declarations . . . 82
8.4 Auxiliary functions . . . 82
8.4.1 Printing axioms . . . 82
8.4.2 Printing function declarations . . . 84
8.4.3 Printing tabular definitions . . . 84
8.4.4 Loading external EVES theories . . . 84
8.4.5 Commonly used C macros . . . 85
8.4.6 Auxiliary operations for the `eves2` traversal . . . 85
8.5 Attribute definitions . . . 86
8.6 Traversal actions . . . 86
8.6.1 For `system` productions . . . 86
8.6.2 For the `component` productions . . . 86
8.6.3 For the `state_def` productions . . . 87
8.6.4 For the `constants` productions . . . 89
8.6.5 For the `variables` productions . . . 91
8.6.6 For the `events` productions . . . 93
8.6.7 For the `interfaces` productions . . . 95
8.6.8 For the `macros` productions . . . 95
8.6.9 For the `functions` productions . . . 95

8.6.10 For the **transition busses** productions . . . 97
8.6.11 For the **transitions** productions . . . 97
8.6.12 For the **predicate** productions . . . 97
8.6.13 For the **conditions** productions . . . 97
8.6.14 For the **expressions** productions . . . 97
8.7 Bodies of auxiliary functions . . . 97
8.7.1 eves2_skeleton . . . 97
8.7.2 print_eves_axiom . . . 99
8.7.3 print_eves_family . . . 99
8.7.4 eves_taut_check . . . 100
8.7.5 eves_table_check . . . 100
8.7.6 print_eves_function_stub . . . 101
8.7.7 declare_in_state . . . 101
8.7.8 print_tabular_definition . . . 101
8.7.9 declare_disjoin . . . 101
8.7.10 two_child_or_rule . . . 102
8.7.11 alternative_or_rules . . . 103

9 PVS **104**
9.1 Currently implemented . . . 104
9.2 Preliminaries . . . 105
9.3 Attribute declarations . . . 105
9.4 Auxiliary functions . . . 106
9.4.1 Printing constant declarations . . . 106
9.4.2 Printing formulas . . . 106
9.4.3 PVS skeleton . . . 108
9.5 Attribute definitions . . . 108
9.6 Traversal actions . . . 108
9.6.1 For the production **system** . . . 108
9.6.2 For the **component** productions . . . 108
9.6.3 For the **states** productions . . . 109
9.6.4 For the **constants** productions . . . 111
9.6.5 For the **variables** productions . . . 111
9.6.6 For the **events** productions . . . 111
9.6.7 For the **interfaces** productions . . . 111
9.6.8 For the **macros** productions . . . 111
9.6.9 For the **functions** productions . . . 111
9.6.10 For the **transition busses** productions . . . 112
9.6.11 For the **transitions** productions . . . 112
9.6.12 For the **predicate** productions . . . 112
9.6.13 For the **conditions** productions . . . 112
9.6.14 For the **expressions** productions . . . 112
9.7 Bodies of auxiliary functions . . . 112
9.7.1 pvs_const_decl . . . 112

9.7.2 pvs_declare_in_state . . . 112
9.7.3 print_pvs_formula . . . 112
9.7.4 print_pvs_family . . . 113
9.7.5 pvs_taut_check . . . 113
9.7.6 pvs_skeleton . . . 113

10 SPIN **116**
10.1 The strategy . . . 116
10.2 Preliminaries . . . 116
10.3 Attribute declarations . . . 117
10.4 Auxiliary functions . . . 117
10.5 Attribute definitions . . . 118
10.6 Traversal actions . . . 118
10.6.1 For the production `system` . . . 118
10.6.2 For the `component` productions . . . 118
10.6.3 For the `states` productions . . . 119
10.6.4 For the `constants` productions . . . 122
10.6.5 For the `variables` productions . . . 123
10.6.6 For the `events` productions . . . 126
10.6.7 For the `interfaces` productions . . . 127
10.6.8 For the `macros` productions . . . 129
10.6.9 For the `functions` productions . . . 129
10.6.10 For the `transition busses` productions . . . 129
10.6.11 For the `transitions` productions . . . 129
10.6.12 For the `predicate` productions . . . 132
10.6.13 For the `conditions` productions . . . 132
10.6.14 For the `expressions` productions . . . 132
10.7 Bodies of auxiliary functions . . . 132

11 Parsing RSML **138**

12 Auxiliary C++ code **147**
12.1 c_style.h . . . 147
12.2 c_style.cc . . . 151
12.3 conditions.h . . . 157
12.4 conditions.cc . . . 158
12.5 configuration.h . . . 158
12.6 event_value.h . . . 159
12.7 event_value.cc . . . 160
12.8 event_value_utilities.h . . . 160
12.9 event_value_utilities.cc . . . 160
12.10 eves2_state.h . . . 161
12.11 eves2_state.cc . . . 162
12.12 formula_table.h . . . 162

12.13 formula_table.cc . . . 164
12.14 formula_table_utilities.h . . . 165
12.15 formula_table_utilities.cc . . . 166
12.16 function_definition.h . . . 166
12.17 function_definition.cc . . . 168
12.18 function_definition_utilities.h . . . 168
12.19 function_definition_utilities.cc . . . 168
12.20 global_table.h . . . 169
12.21 global_table.cc . . . 172
12.22 load.h . . . 177
12.23 load.cc . . . 177
12.24 main.cc . . . 179
12.25 misc_utilities.cc . . . 179
12.26 misc_utilities.h . . . 180
12.27 pvs_state.h . . . 180
12.28 pvs_state.cc . . . 181
12.29 pvs_types.h . . . 181
12.30 pvs_types.cc . . . 182
12.31 rsml_state.h . . . 182
12.32 rsml_state.cc . . . 183
12.33 scaled_integer.h . . . 183
12.34 scaled_integer.cc . . . 185
12.35 s_exp.cc . . . 189
12.36 s_exp.h . . . 195
12.37 s_exp_utilities.cc . . . 201
12.38 s_exp_utilities.h . . . 204
12.39 simple_list.h . . . 205
12.40 skeleton_utilities.h . . . 207
12.41 skeleton_utilities.cc . . . 207
12.42 s_list_table.h . . . 208
12.43 s_list_table.cc . . . 209
12.44 s_list_table_utilities.h . . . 211
12.45 s_list_table_utilities.cc . . . 211
12.46 spin_state.h . . . 212
12.47 state_value.h . . . 212
12.48 state_value.cc . . . 213
12.49 state_value_utilities.h . . . 214
12.50 state_value_utilities.cc . . . 214
12.51 str.h . . . 215
12.52 str.cc . . . 216
12.53 strlist_utilities.h . . . 217
12.54 strlist_utilities.cc . . . 217
12.55 table.h . . . 217
12.56 table.cc . . . 218

12.57 tagged_table.h 219
12.58 tagged_table.cc 223
12.59 transition_value.h 225
12.60 transition_value.cc 227
12.61 combinatorics.h 227
12.62 combinatorics.cc 228

13 The Makefile **230**

Chapter 1

Outline

The executable `translator` is built from the following sources:

- A collection of auxiliary C++ classes and utilities residing in `.h` and `.cc` files.
- A file `grammar.l` that serves as one of the inputs to Ox and amounts to a Lex input specification for RSML.
- A set of noweb files (named with extension `nw`) that are notangled to produce `grammar.cc`, which serves as the other input to Ox. These files define attributes and traversal actions.

To make `translator` from these pieces one needs Ox (which itself requires either Yacc or Bison, and either Lex or Flex); notangle; and a C++ compiler, to compile the output from Ox. This document contains the following things:

- A grammar for RSML, in chapter 2
- The noweb outline for generating `grammar-cc.tex`, in chapter 3.
- The definitions of the grammars tokens and operator precedence, in chapter 4.
- Declarations of the "basic" attributes, in chapter 6.
- Definitions of attributea, in chapter 7.
- The partially filled-in template for encoding RSML models as EVES theories, in chapter 8. This includes the definitions of attribute, auxiliary functions, and traversals.
- The lex file for parsing RSML, in chapter 11
- The auxiliary C++ code, in chapter 12
- The makefile for code and documentation, in chapter 13.

A sample invocation of `translator` from the command line looks something like

```
translator -ixc -e -s blort
```

where `blort` is the name of the RSML definition to be translated. The parameters `x`, `c`, `e`, and `s` set "traversal switches"—and each traversal generates an encoding of `blort` for some formal analysis tool. In the current implementation, the very first switch must always be `i`, which codes a special traversal that initializes certain global data.

Chapter 2

The grammar of RSML

```
/* ----------------------------------------------------------------
 * SYSTEM
 *
 * A system is a collection of compnents
 */

system                  : SYSTEM SYSTEM_NAME ':'
                          component_list
                          END SYSTEM
  SECTION_SEPARATOR
                        ;

component_list          :  /* empty */
                        |  component_list component_def
                        ;

/* ----------------------------------------------------------------
 * COMPONENTS
 *
 * A component does not have to serve any purpose and the body of the
 * component can be left empty.  A strict ordering of the parts of a
 * components is enforced.  The main reason for this action is the need for
 * a structured approach to typechecking/declaration/use checking.
 */
component_def           : COMPONENT COMPONENT_NAME ':'
                          END COMPONENT
                        | COMPONENT COMPONENT_NAME ':'
                           state_def
                           constant_def_list
```

```
                          event_def_list
                          in_variable_def_list
                          out_variable_def_list
                          in_interface_def_list
                          out_interface_def_list
                          macro_def_list
                          function_def_list
                          transition_or_transition_bus_def_list
                          END COMPONENT
                        ;

component_location      : COMPONENT_NAME
                        | EXTERNAL
                        ;


/* ---------------------------------------------------------------*/
/* States */
/* ---------------------------------------------------------------*/


/* If a component has a body it has to contain some state. An empty */
/* state space is not allowed */

state_def               : OR_STATE_TOKEN STATE_NAME DEFAULT STATE_NAME ':'
                          state_charts_list
                          END OR_STATE_TOKEN
                        | AND_STATE_TOKEN STATE_NAME ':'
                          state_charts_list
                          END AND_STATE_TOKEN
                        | OR_ARRAY STATE_NAME
                          '[' INTEGER ']' ':'
                          state_charts_list
                          END OR_ARRAY
                        | AND_ARRAY STATE_NAME
                          '[' INTEGER ']' ':'
                          state_charts_list
                          END AND_ARRAY
                        | ATOMIC STATE_NAME ':'
                        | CONDITIONAL STATE_NAME ':'
                        ;

state_charts_list       : state_def
                        | state_charts_list state_def
                        ;

/* For IN_ONE_OF predicates */
```

```
state_list              : STATE_NAME
                        | state_list ',' STATE_NAME
                        ;

/* For transition locations, IN STATE predicates */
simple_state_path       : STATE_NAME
                        | simple_state_path '.' STATE_NAME
                        ;

state_path              : STATE_NAME '[' parameter ']'
                        | STATE_NAME '[' parameter ']' '.' simple_state_path
                        | simple_state_path '.' STATE_NAME '[' parameter ']'
                        | simple_state_path '.'
                          STATE_NAME '[' parameter ']' '.'
                          simple_state_path
                        | simple_state_path
                        ;

/* currently not used - Vivek - 3/31 */
state_path_ref          : PREV '(' REAL ')' state_path
                        | state_path
                        ;


/* ---------------------------------------------------------------*/
/* Constants */
/* ---------------------------------------------------------------*/

constant_def            :  CONSTANT CONSTANT_NAME ':'
                           VALUE ':' REAL
                           END CONSTANT
                        |  CONSTANT CONSTANT_NAME ':'
                           VALUE ':' '-' REAL
                           END CONSTANT
                        ;

constant_ref            : CONSTANT_NAME
                        ;

constant_def_list:      /* empty */
                        | constant_def_list constant_def
                        ;


/* ---------------------------------------------------------------*/
/* Variables */
/* ---------------------------------------------------------------*/
```

```
in_variable_def : IN_VARIABLE IN_VAR_NAME ':'
                             TYPE ':' NUMERIC
   expected_min
                             expected_max
                             min_gran
                             max_gran
                             END IN_VARIABLE
/* I'm not sure if we suport parameterized vars in
the simulator. Hence I'm commenting this out for now.
- Vivek - 6/8/95
                           | IN_VARIABLE IN_VAR_NAME '[' REAL ']' ':'
                             TYPE ':' NUMERIC
   expected_min
                             expected_max
                             min_gran
                             max_gran
                             END IN_VARIABLE
 */
                           ;

out_variable_def : OUT_VARIABLE OUT_VAR_NAME ':'
                             TYPE ':' NUMERIC
   expected_min
                             expected_max
                             min_gran
                             max_gran
                             ASSIGNMENT ':' expression
                             TRIGGER ':' event_ref
                             END OUT_VARIABLE
/* I'm not sure if we suport parameterized vars in
the simulator. Hence I'm commenting this out for now.
- Vivek - 6/8/95
                           | OUT_VARIABLE OUT_VAR_NAME '[' REAL ']' ':'
                             TYPE ':' NUMERIC
   expected_min
                             expected_max
                             min_gran
                             max_gran
                             ASSIGNMENT ':' expression
                             TRIGGER ':' event_ref
                             END OUT_VARIABLE
*/
                           ;

expected_min :  /* empty */
| EXPECTED_MIN ':' REAL
```

```
;

expected_max  :  /* empty */
| EXPECTED_MAX ':' REAL
;

min_gran :  /* empty */
| MIN_GRAN ':' REAL
;

max_gran :  /* empty */
| MAX_GRAN ':' REAL
;

in_variable_ref         : IN_VAR_NAME
/* I'm not sure if we suport parameterized vars in
the simulator. Hence I'm commenting this out for now.
- Vivek - 6/8/95
                        | IN_VAR_NAME '[' parameter ']'
*/
                        ;

out_variable_ref        : OUT_VAR_NAME
/* I'm not sure if we suport parameterized vars in
the simulator. Hence I'm commenting this out for now.
- Vivek - 6/8/95
                        | OUT_VAR_NAME '[' parameter ']'
*/
                        ;

in_variable_ass         : IN_VAR_NAME
/* I'm not sure if we suport parameterized vars in
the simulator. Hence I'm commenting this out for now.
- Vivek - 6/8/95
                        | IN_VAR_NAME '[' parameter ']'
*/
                        ;

out_variable_ass        : OUT_VAR_NAME
/* I'm not sure if we suport parameterized vars in
the simulator. Hence I'm commenting this out for now.
- Vivek - 6/8/95
                        | OUT_VAR_NAME '[' parameter ']'
*/
                        ;

/* In-variables are optional */
```

```
in_variable_def_list : /* empty */
                       | in_variable_def_list in_variable_def
                       ;

/* Out-variables are optional */
out_variable_def_list : /* empty */
                       | out_variable_def_list out_variable_def
                       ;

in_variable_list       : /* empty */
                       | in_variable_ass
                       | in_variable_list ',' in_variable_ass
                       ;

variable               : out_variable_ref
                       | in_variable_ref
                       ;

variable_ref           : PREV '(' REAL ')' in_variable_ref
                       | in_variable_ref
                       ;


/* ----------------------------------------------------------------*/
/* Events */
/* ----------------------------------------------------------------*/

event_def_list :        /* empty */
                       | event_def_list event_def
                       ;

event_def :  EVENT EVENT_NAME STATE ';'
| EVENT EVENT_NAME VARIABLE ';'
| EVENT EVENT_NAME INTERFACE ';'

/* I'm not sure if we suport parameterized events in
the simulator. Hence I'm commenting this out for now.
- Vivek - 6/8/95
                       | EVENT EVENT_NAME '[' INTEGER ']' ';'
*/
                       ;

event_ref              : EVENT_NAME
/* I'm not sure if we suport parameterized events in
the simulator. Hence I'm commenting this out for now.
- Vivek - 6/8/95
                       | EVENT_NAME '[' parameter ']'
```

```
 */
                           ;

event_list                 : /* empty */
                           | event_ref
                           | event_list ',' event_ref
                           ;

/* ------------------------------------------------------------*/
/* Interface definitions */
/* ------------------------------------------------------------*/

in_interface_def_list      : /* empty */
                           | in_interface_def_list in_interface_def
                           ;

in_interface_def           : IN_INTERFACE IN_INTERFACE_NAME ':'
                             SOURCE ':' component_location
                             TRIGGER ':' RECEIVE '(' in_variable_list ')'
                             selection_condition
                             ACTION ':' event_list
                             END IN_INTERFACE
                           ;

/* Output Interface is optional */
out_interface_def_list  :  /* empty */
                        |  out_interface_def_list out_interface_def
                        ;

out_interface_def       :  OUT_INTERFACE OUT_INTERFACE_NAME ':'
                           DESTINATION ':' component_location
                           TRIGGER ':' event_ref
                           selection_condition
                           ACTION ':' SEND '(' parameter_list ')'
                           END OUT_INTERFACE
                        |  OUT_INTERFACE OUT_INTERFACE_NAME ':'
                           DESTINATION ':' component_location ',' IN_INTERFACE_NAME
                           TRIGGER ':' event_ref
                           selection_condition
                           ACTION ':' SEND '(' parameter_list ')'
                           END OUT_INTERFACE
                        ;

selection_condition :  /* empty */
|  SELECTION ':' condition
;
```

```
/* --------------------------------------------------------------*/
/* Macros */
/* --------------------------------------------------------------*/

macro_def_list          :  /* empty */
                        |  macro_def_list macro_def
                        ;

macro_def               :  MACRO MACRO_NAME '(' local_var_list ')' ':'
                           condition
                           END MACRO
                        ;


/* --------------------------------------------------------------*/
/* Functions */
/* --------------------------------------------------------------*/

function_def_list       :  /* empty */
                        |  function_def_list function_def
                        ;

function_def            :  FUNCTION FUNCTION_NAME '(' local_var_list ')' ':'
                           RETURN ':' CASE case_list END CASE
                           END FUNCTION
                        ;

case_list               :  case
                        |  case_list case
                        ;

case                    :  expression IF pred_ref ';'
                        ;

/* --------------------------------------------------------------*/
/* Transition busses
/* --------------------------------------------------------------*/

transition_or_transition_bus_def_list :  /* empty */
  | transition_or_transition_bus_def_list
    transition_or_transition_bus_def
  ;


transition_or_transition_bus_def : transition_def
     | transition_bus_def
```

```
    ;

transition_bus_def      : TRANSITIONBUS TRANSITIONBUS_NAME ':'
  transition_def_list
  END TRANSITIONBUS
;


/* ------------------------------------------------------------------*/
/* Transitions */
/* ------------------------------------------------------------------*/

transition_def_list       :  /* empty */
                          | transition_def_list transition_def
                          ;

transition_def            :  TRANSITION TRANSITION_NAME FROM STATE_NAME TO STATE_NAME ':'
/*                             LOCATION ':' simple_state_path */
                    transition_maybe_trigger
                             transition_maybe_cond
                             transition_maybe_action
                             END TRANSITION
                          ;

transition_maybe_trigger : /* empty */
                          |  TRIGGER ':' event_ref
                          |  TRIGGER ':' TIMEOUT '(' simple_expression ',' expression ')'
                          ;

transition_maybe_cond   : /* empty */
| CONDITION ':' condition
                          ;

transition_maybe_action : /* empty */
                          | ACTION ':' event_list
                          ;

/* ------------------------------------------------------------------*/
/* Predicates */
/* ------------------------------------------------------------------*/

pred_ref                  :  predicate
;


predicate                 :  neg_predicate
                          |  macro_ref
```

```
                        |  math_predicate
                        |  state_predicate
                        |  forall_predicate
                        |  exists_predicate
                        |  TRUE_TOKEN
                        |  FALSE_TOKEN
                        ;

neg_predicate           :  NOT pred_ref
                        ;

macro_ref               :  MACRO_NAME '(' parameter_list ')'
                        ;

math_predicate          :  expression /* changed simple_expression to
expression - Vivek - 3/9 */
                           boolean_math_operator
                           expression /* changed simple_expression to
expression - Vivek - 3/9 */
                        |  simple_expression EQ_ONE_OF '{' parameter_list '}'
                        ;

boolean_math_operator   :  '='
                        |  '>'
                        |  '<'
                        |  LESS_OR_EQUAL
                        |  GREATER_OR_EQUAL
                        |  NOT_EQUAL
                        ;


state_predicate         :  simple_state_path IN_STATE STATE_NAME
                           /* should be state_path_ref */
                        |  simple_state_path IN_ONE_OF
                           /* should be state_path_ref */
                           '{' state_list '}'
                        ;

forall_predicate        :  FORALL LOCAL_VAR_NAME
                           '[' constant_ref ',' constant_ref ']' ','
                           pred_ref
                        ;

exists_predicate        :  EXISTS LOCAL_VAR_NAME
                           '[' constant_ref ',' constant_ref ']' ','
                           pred_ref
                        ;
```

```
/* -----------------------------------------------------------------*/
/* Conditions & and/or tables */
/* -----------------------------------------------------------------*/

condition               : FORALL LOCAL_VAR_NAME
                          '[' constant_ref ',' constant_ref ']' ','
                          condition
                        | EXISTS LOCAL_VAR_NAME
                          '[' constant_ref ',' constant_ref ']' ','
                          condition
                        | TABLE
                          row_list
                          END TABLE
                        | TRUE_TOKEN
                        | FALSE_TOKEN
                        ;

row_list                : pred_ref ':' truth_value_list ';'
                        | row_list pred_ref ':' truth_value_list ';'
                        ;

truth_value_list        : truth_value
                        | truth_value truth_value_list
                        ;

truth_value             : 'T'
                        | 'F'
                        | '.'
                        ;


/* -----------------------------------------------------------------*/
/* Expressions */
/* -----------------------------------------------------------------*/

/* Some expressions are missing: abs, sqrt. Check with SRS */
expression              : expression '*' expression
                        | expression '/' expression
                        | expression '+' expression
                        | expression '-' expression
                        | '-' expression %prec UMINUS
                        | '(' expression ')'
| simple_expression
                        ;
```

```
simple_expression       : variable_ref
                        | function_ref
                        | constant_ref
                        | LOCAL_VAR_NAME
                        | TIME
                        | TIME '(' event_ref ')'
                        | TIME '(' PREV '(' REAL ')' event_ref ')'
                        | TIME '(' variable_ref ')'
                        | TIME '(' ENTERED STATE_NAME ')'
                        ;


function_ref            : FUNCTION_NAME '(' parameter_list ')'
                        ;

parameter_list          : /* empty */
                        | parameter
                        | parameter_list ',' parameter
                        ;

parameter               : expression
/* changed simple_expression to expression - Vivek -5/22 */
                        /*| THIS
*/
                        ;

local_var_list          : /* empty */
                        | LOCAL_VAR_NAME
                        | local_var_list ',' LOCAL_VAR_NAME
                        ;
```

Chapter 3

Making grammar.cc

This chapter provides the noweb outline from which **grammar.cc** is generated, and a brief explanation of each of the top-level modules. The last section lists of assumptions and limitations governing this prototype code.

3.1 The outline for grammar.cc

⟨*⟩≡

```
%{

#include <stdlib.h>
#include <stdio.h>
#include "configuration.h"
#include "lex.yy.h"
#include "load.h"
#include "generated_include_file.h"
#include "generated_C_macros.h"

⟨Function headers⟩
⟨Globals⟩


%}

⟨Tokens⟩
⟨Auto-attributes⟩
⟨Attributes⟩
⟨Precedence⟩
```

```
⟨Traversals⟩
⟨Ox Macros⟩

%%

⟨rules⟩

%%

⟨Bodies⟩

// This assumes that the command line looks like
//      whatever -axc -e -s blort
// It ignores the first, sets the traversal switches denominated
// by a, x, c, e, and s, and returns a pointer to blort.

char *handleCLopts(int argc, char **argv)
  {int i;

   for (i=1;i<argc;i++)
     {int j,len;

      if (argv[i][0] != '-') return argv[i];
      len = strlen(argv[i]);
      for (j=1;j<len;j++)
        {
         switch (argv[i][j])
           {

                ⟨Switches for traversals⟩

            default: goto err02;
           }
        }
      continue;
err02 : fprintf(stderr,
        "error in command line option: %s\n",argv[i]);
     }
  }
```

3.2 The top-level noweb modules

Here is a brief description of the purposes and use of the top-level noweb modules. We expect a developer to write auxiliary C++ functions in `.h` and `.cc` files, and to define the attributes and traversals for a translation by writing a new noweb file that fleshes out a predefined skeleton that serves as a template. The current, partially fleshed out, skeleton for the EVES prover is given in chapter 8.

Deciding which C++ code counts as "auxiliary" and which should be included in the noweb source is a matter of taste, but when C++ code is included in the noweb files it must be assigned to the proper noweb modules. All files containing auxiliary code must be guarded with `#ifndef` commands to avoid problems with multiple declarations.

First, the modules occurring in, or incorporated direclty within, the top-level modules shown above:

- "Function headers" and "Bodies": prototypes and bodies for C++ code defined in noweb files belong in these.
- "Globals": gathers up declarations of global variables. On the whole, we avoid these in favor of a functional programming style.
- "Attributes": gathers up declarations of Ox attributes. The developer never writes an "Attributes" module. Instead, he puts attribute declarations for a particular grammar symbol in a module for that symbol, and these individual modules are automatically collected into the top-level module "Attributes". The modules for individual symbols are defined in chapter 6.
- "Traversals": gathers up declarations of traversals.
- "Ox macros": gathers up declarations of Ox macros.
- "rules": gathers up all the Ox attribute definitions and traversal code. The developer never writes a "rules" module directly. Instead, he puts attribute declarations for a particular grammar production in a module for that production, and these individual modules are automatically collected into the top-level module called "rules".

The following two modules are not reachable from the root module defined above.

- "Includes": `#include` commands occurring in "Includes" modules are automatically gathered together and placed in the file `generated_include_files.h`. Sample use: when the developer declares a new attribute of type `T`, that definition should be preceded by an "Includes" module containing the definition of `T`. This is helpful documentation, and also ensures that the compiler will find the definition.
- "C Macros": `#define` commands occurring in "C macros" modules are automatically gathered together and placed in the file `generated_C_macros.h`.

The developer should not modify the modules "Tokens" and "Precedence", which pertain to parsing of RSML.

3.3 Assumptions and limitations

To simplify certain parsing problems we currently assume

- All names occurring in the rsml text, other than local variables, are distinct.
- No more than 1000 names occur in the text. (This is set in a global variable, `MAX_SPEC_NAMES` defined in `configuration.h`.
- A system has only one component. The name of the top-level state of that component is stored in the global state information.

Some features are not (fully) supported because of unresolved semantic questions:

- OR-array states, AND-array states, or transition busses. Applying the translators to code that contains these may result in a crash, or may generate meaningless text.
- Conditional states—the SPIN translation prints an error message when it encounters them. The SPIN code output by the translator in this case will not, in general, compile.
- The SPIN translation treats real constants like integers, and assumes that they have the form `xxx.0`.

Chapter 4

Parsing preliminaries

These two modules in this chapter are concerned solely with lexing and parsing RSML, and a developer need not (must not) change them.

4.1 Tokens

Here is the complete list of tokens of the RSML grammar.

⟨*Tokens*⟩≡

```
%token ACTION AND_ARRAY AND_STATE_TOKEN ATOMIC CASE COMPONENT CONDITION
%token CONSTANT DESTINATION END ENTERED EVENT EXISTS EXPECTED_MAX
%token EXPECTED_MIN FORALL FROM FUNCTION GREATER_OR_EQUAL IF IN_INTERFACE
%token INTEGER IN_ONE_OF IN_STATE IN_VARIABLE VARIABLE LESS_OR_EQUAL LOAD
%token MACRO MAX_GRAN MIN_GRAN NOT NOT_EQUAL NUMERIC OR_ARRAY
%token OR_STATE_TOKEN OUT_INTERFACE OUT_VARIABLE REAL RETURN SOURCE SYSTEM
%token TO TRANSITION TRANSITIONBUS TRIGGER TYPE VALUE CONDITIONAL
%token ASSIGNMENT SECTION_SEPARATOR

%token TIMEOUT LOCATION EXTERNAL STATE INTERFACE RECEIVE SELECTION SEND
%token PREV TIME THIS TRUE_TOKEN FALSE_TOKEN TABLE DEFAULT EQ_ONE_OF

%token LEX_ERROR

%token STATE_NAME SYSTEM_NAME COMPONENT_NAME EVENT_NAME
%token OUT_INTERFACE_NAME IN_INTERFACE_NAME CONSTANT_NAME
%token IN_VAR_NAME OUT_VAR_NAME FUNCTION_NAME LOCAL_VAR_NAME
%token MACRO_NAME TRANSITION_NAME TRANSITIONBUS_NAME
```

4.2 Precedence

Here are the precedence rules for evaluating mathematical expressions.

⟨*Precedence*⟩≡

```
%left '+' '-'
%left '*' '/'
%left UMINUS
```

Chapter 5

Collecting global information

Instead of proceeding in a purely functional programming style and passing a symbol table attribute through the parse tree, it seems convenient to define a global data structure that summarizes a small amount of global semantic information and is updated by side-effects during attribute evaluation. Section 5.1 describes this data structure; section 5.2 describes the strategy for making sure that all necessary writes are done before each read of this structure; and section 5.3 describes the *initialization* traversal that completes its initialization, and which must be done before any other traversal. The initialization traversal is not strictly necessary, since we could arrange that everything is done during attribute evaluation. Currently, only the SPIN traversal needs access to this global data.

5.1 The variable `glob`

Global semantic information will be stored in a variable `glob` of type `global_state`. The code for this class can be found in `global_table.h` and `combinatorics.h`.

Currently, `glob` contains the following (private) data. We describe the meaning of this data *when* `glob` *is complete.* How and when it gets completed are described in section 5.2:

- Integers `num_states`, `num_events`, and `num_transitions` recording, respectively, the total number of states, events, and transitions declared in the specification.
- One-dimensional arrays `state_codes` and `transition_codes`. For each `i` < `num_states`, `state_codes[i]` is a pointer to the `state_value` of the state whose unique integer code is `i`; `transition_codes` does the analogous thing for transitions.
- Two-dimensional arrays of integers:
  - `child_of[i][j]` == 1 iff state `i` is a child of state `j`
  - `leq` is the reflexive transitive closure of the relation coded by `child_of`. That is, `leq[i][j]` == 1 iff state `i` is state `j` or a descendant of state `j`
  - `compatible[i][j]` == 1 iff transition `i` and transition `j` are compatible.

The global_table class provides methods for allocating, setting, accessing, and printing this information.

5.2 Completing glob

```
⟨Include files⟩≡
  #include "global_table.h"
  #include "rsml_state.h"
```

```
⟨Globals⟩≡
  global_table glob;
  rsml_state rsml;
```

An auxiliary global variable rsml, of class rsml_state, does two things:

- It keeps track of various statistics about the specification, in components updated during the first lexing pass. It currently counts declarations of states, events, and transitions.
- During attribute evaluation rsml is used to dole out unique integer id's for state, event, and transition declarations.

When the first lexing pass is over, the statistics stored in rsml are correct. At that point, and before the final lexing/parsing/Ox pass begins (see load.cc) the variable glob is updated as follows:

- num_states, num_events, and num_transitions are set to their proper values
- All the arrays in glob are allocated. In addition, the arrays [leq, child_of, and compatible are initialized with 0's.

Currently, all the information needed in glob can be obtained from the productions state_def, state_charts_list, and transition_def. The information is actually inserted into glob in the following ways:

- As a side effect of evaluating the glob attribute of state_def, state_charts_list, and transition_def. The actual value of this attribute is irrelevant.
- At the beginning of the init traversal.

The order in which things happen matters.

In each state_def production, evaluation of the glob attribute has the effect of updating state_codes with the state_value of the state being defined, and updating child_of to model the relation between the state defined and its children (if any). We introduce explicit dependencies, via the Ox @e command, to guarantee that the glob attribute of a state cannot be evaluated before the glob attribute of any its children. These updates are all logically independent of one another, but by forcing bottom-up evaluation we guarantee that when state_def.glob is evaluated in the following top-level production

```
component_def                : ...
                             | COMPONENT COMPONENT_NAME ':'
                               state_def
                               ...
                               transition_or_transition_bus_def_list
                               END COMPONENT
                             ;
```

both `state_codes` and `child_of` are complete. This is important because these two arrays must be complete before we can correctly update any of the other array components of `glob`. These other updates are performed when visiting the `transition_def` descendant of `transition_or...`.

We delay evaluation of the attribute `transition_def.glob` as follows: All productions in the chain from `transition_..._list` down to `transition_def` are given an auto-inherited attribute `state_def_done`; and within `component_def` we use `@e` to force `transition_..._list.state_def_done` to depend on `state_def.glob`. That trick effects the delay. In addition, evaluation of the top level `state_def_done` is accompanied by a side effect on `glob`, which uses Warshall's algorithm to calculate `leq` from `child_of`.

We therefore evaluate each `transition_def.glob` in a context in which all of `state_codes`, `child_of`, and `leq` are complete. This context permits each `transition_def` production to properly update its piece of `transition_codes`. Therefore, at the end of attribute evaluation, all of `glob` except `compatible` has been correctly initialized. The first step in the `init` traversal is to call `glob.set_compatible()`, which completes the initialization. As noted above, we could easily arrange to call `glob.set_compatible()` during attribute evaluation, after all the other parts of `glob` are complete.

5.3 Traversal actions

⟨*Globals*⟩+≡

```
int init = 0;
```

⟨*Traversals*⟩≡

```
@traversal @preorder init
```

The command line argument choosing this traversal is `i`

⟨*Switches for traversals*⟩≡

```
case 'i' : init = 1; break;
```

The only effect of this traversal is to complete the initialization of `glob` by calculating the table of compatible transitions.

⟨*system:SYSTEM SYSTEM_NAME ':' component_list END SYSTEM SECTION_SEPARATOR* ⟩≡

```
@init if (init) { glob.set_compatible(); }
```

Chapter 6

Attribute declarations

This is the master file containing the declarations of all the basic attributes (and, when notangled, gathers them all up into the top-level "Attributes" module). Each declaration provides a module into which developers may place additional attribute declarations needed for particular traversals. So, for example, a developer wanting to declare additional attributes for an **INTEGER** token will put such declarations into the "INTEGER attributes" module. We provide the developer with a list of all such modules in the file **attribute_modules.nw**, and that file serves as part of the template for integrating a new tool.

6.1 Attributes of tokens

⟨*Include files*⟩+≡
```
#include "scaled_integer.h"
```

⟨*Attributes*⟩≡
```
@attributes {
  int val;
  ⟨INTEGER attributes⟩
} INTEGER
```

⟨*INTEGER attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
  scaled_integer *val;
  ⟨REAL attributes⟩
} REAL
```

⟨*REAL attributes*⟩≡

6.2 Attributes for system

⟨*Attributes*⟩+≡
```
@attributes {
⟨system attributes⟩
} system
```

⟨*system attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
⟨component_list attributes⟩
} component_list
```

⟨*component_list attributes*⟩≡

6.3 Attributes for components

`*in_top_state` is the `S_exp` asserting that the state machine is always in the top-level state.

⟨*Attributes*⟩+≡
```
@attributes {

  S_exp *in_top_state;

⟨component_def attributes⟩
} component_def
```

⟨*component_def attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
⟨component_location attributes⟩
} component_location
```

⟨*component_location attributes*⟩≡

6.4 Attributes for states

The data associated with a STATE_NAME is a state_value containing the name of the state, represented as a Str, its integer code, and the kind of state it is (OR, AND, etc.).

⟨*Include files*⟩+≡

```
#include "state_value.h"
```

If state_def defines the state Foo, then parent_as_S_exp is the s-expression (Foo) and children_as_S_exps is the list of all such s-expressions for the children of Foo.

The remaining attributes have the following meanings, written in infix form rather than as S-expressions:

- parent_implies_children is parent -> child1 and child2 and ... if state_def is and AND-state and parent -> child1 or child2 or ... if state_def is an OR-state; and is otherwise true.
- children_imply_parent is the list child1 -> parent, child2 -> parent ...
- If state_def is an OR-state incompatible_siblings is the list

```
child1 -> (not child2) and (not child3) and (not child4) and ...
child2 -> (not child3) and (not child4) and ...
child3 -> (not child4) and ...
...
```

  and is otherwise the empty list.

The value of the glob attribute is a dummy. The point of evaluating it is to have a side effect initializing certain global variables with semantic information. See chapter 5.

The value of code is an integer uniquely associated with the state name introduced by this declaration. The inherited parent_code is used to fill in the child_of array.

conditionals_so_far is a synthesized attribute of state_def and state_charts_list. It returns a *StateValueList containing all conditional states so far encountered (bottom up).

ancestor is an inherited attributed containing the Str* naming the parent, if one exists—and, for a top-level state, containing "".

⟨*Attributes*⟩+≡

```
@attributes {
  state_value *val;
  S_exp *parent_as_S_exp;
  S_exp_list *children_as_S_exps;

  S_exp *parent_implies_children;
```

```
    S_exp_list *children_imply_parent;
    S_exp_list *incompatible_siblings;

    int glob; // I'll eventually change this to initialize
    int code;

    StateValueList *conditionals_so_far;
    Str *ancestor; // Inherited

    ⟨state_def attributes⟩
  } state_def
```

⟨*state_def attributes*⟩≡

⟨*Attributes*⟩+≡

```
  @attributes {
    StateValueList *comp_vals;     // Constructed from
                                   // current val and
                                   // inherited values
    StateValueList *inherited_vals;        // Inherited

    int glob;

    StateValueList *conditionals_so_far;
    Str *ancestor;

    ⟨state_charts_list attributes⟩
  } state_charts_list
```

⟨*state_charts_list attributes*⟩≡

The usual boilerplate: At the top level, `computed_exps` returns the list of all the states in the list, as S-expressions.

⟨*Attributes*⟩+≡

```
  @attributes {
    S_exp_list* comp_exps;
    S_exp_list* inherited_exps;
  ⟨state_list attributes⟩
  } state_list
```

⟨*state_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*simple_state_path attributes*⟩
```
} simple_state_path
```

⟨*simple_state_path attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*state_path attributes*⟩
```
} state_path
```

⟨*state_path attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*state_path_ref attributes*⟩
```
} state_path_ref
```

⟨*state_path_ref attributes*⟩≡

6.5 Attributes for constants

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*constant_def attributes*⟩
```
} constant_def
```

⟨*constant_def attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
  S_exp *exp;
```
⟨*constant_ref attributes*⟩
```
} constant_ref
```

⟨*constant_ref attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
⟨constant_def_list attributes⟩
} constant_def_list
```

⟨*constant_def_list attributes*⟩≡

6.6 Attributes for variables

Currently: a very partial version of in variables and out variables (but not interfaces). It seems that, right now, all variables are real variables. The proper model of such a variable is a map from time to real values, but for now I'll just make them unspecified real-valued constants. (Semantically, of course, variables are logical constants.) I don't know whether such notions as **expected_min**, **max_gran**, etc., have any semantic significance. For now I will not model them in any way. An **in_variable_def** or **out_variable_def** introduces a new unspecified constant (of type real). So, these productions must generate appropriate declarations.

IN_VAR_NAMEs and **OUT_VAR_NAME**s wind up in expressions via in_variable_ref, out_variable_ref, and variable; so those chain of attributes are declared as well.

⟨*Attributes*⟩+≡

```
@attributes {
⟨in_variable_def attributes⟩
} in_variable_def
```

⟨*in_variable_def attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
⟨out_variable_def attributes⟩
} out_variable_def
```

⟨*out_variable_def attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
⟨expected_min attributes⟩
} expected_min
```

⟨*expected_min attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
⟨expected_max attributes⟩
} expected_max
```

⟨*expected_max attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*min_gran attributes*⟩
```
} min_gran
```

⟨*min_gran attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*max_gran attributes*⟩
```
} max_gran
```

⟨*max_gran attributes*⟩≡

Computing an `in_variable_ref` or `out_variable_ref` with `str` "`foo`" as an s-expression simply means computing the s-expression "`(foo)`". A proper model of variables will presumably have to change this.

⟨*Attributes*⟩+≡
```
@attributes {
  Str *str;
  S_exp *as_S_exp;
```
⟨*in_variable_ref attributes*⟩
```
} in_variable_ref
```

⟨*in_variable_ref attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*out_variable_ref attributes*⟩
```
  Str *str;
  S_exp *as_S_exp;
} out_variable_ref
```

⟨*out_variable_ref attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
  Str *str;
```
⟨*in_variable_ass attributes*⟩
```
} in_variable_ass
```

⟨*in_variable_ass attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
  Str *str;
```
⟨*out_variable_ass attributes*⟩
```
} out_variable_ass
```

⟨*out_variable_ass attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*in_variable_def_list attributes*⟩
```
} in_variable_def_list
```

⟨*in_variable_def_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*out_variable_def_list attributes*⟩
```
} out_variable_def_list
```

⟨*out_variable_def_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
  StrList *computed_val;
  StrList *inherited_val;
```
⟨*in_variable_list attributes*⟩
```
} in_variable_list
```

⟨*in_variable_list attributes*⟩≡

Computing a `variable` or `variable_ref` with `str` "`foo`" as an s-expression simply means computing the s-expression "`(foo)`". A proper model of variables will presumably have to change this.

⟨*Attributes*⟩+≡
```
@attributes {
  Str *str;
  S_exp *as_S_exp;
```
⟨*variable attributes*⟩
```
} variable
```

⟨*variable attributes*⟩≡

.

⟨*Attributes*⟩+≡
```
  @attributes {
    Str *str;
    S_exp *as_S_exp;
  ⟨variable_ref attributes⟩
  } variable_ref
```

⟨*variable_ref attributes*⟩≡

6.7 Attributes for events

⟨*Include files*⟩+≡
```
  #include "event_value.h"
```

The events for a component are declared in its `event_def_list` child. The names of these events then occur (as `event_refs`) in the following ways:

- As triggers in definitions of out variables (`out_var_def_list`) and out interfaces (`out_interface_def_list`), declarations of transitions (`transition_...._def_list`).
- As actions in definitions of in interfaces (`in_interface_def_list`) and declarations of transitions (`transition_...._def_list`).

Declarations of events declare them as being “state” events (which I believe can be triggers for transitions), “variable” events (which I believe cause the setting of out variables), and “interface” events (which I believe are triggered by the receipt of input).

We'll generate a table that associates each event name with the disjunction of the conditions of all its triggering transitions. To do this, the top-level `event_def_list` synthesizes the list of all declared (state) events—in the attribute `computed_events`, which can be handed laterally to the `transition_...._def_list`. A synthesized attributed of *that* production will generate the appropriate set of tests.

The value of `code` is an integer uniquely associated with the event name introduced by an event declaration.

For now, we'll just define a `str` attribute with the name of each event, and a list of them with `event_def_list`. The attributes for this are boiler plate.

⟨*Attributes*⟩+≡
```
  @attributes {

          EventValueList *computed_events;
          EventValueList *inherited_events;
  ⟨event_def_list attributes⟩
  } event_def_list
```

⟨*event_def_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {

        event_value *val;
        int code;
```
  ⟨*event_def attributes*⟩
```
} event_def
```

⟨*event_def attributes*⟩≡

The category **event_ref** is used in the productions for out variables, out interfaces, and transition triggers. The category **event_list** is used in defining in interfaces and transition actions. There is no context information—only their names, which can then, presumably, be looked up in a symbol table to find the corresponding event value.

⟨*Attributes*⟩+≡
```
@attributes {

        Str *str;
```
  ⟨*event_ref attributes*⟩
```
} event_ref
```

⟨*event_ref attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {

        StrList *computed_events;
        StrList *inherited_events;
```
  ⟨*event_list attributes*⟩
```
} event_list
```

⟨*event_list attributes*⟩≡

6.8 Attributes for interface definitions

⟨*Attributes*⟩+≡

```
@attributes {
⟨in_interface_def_list attributes⟩
} in_interface_def_list
```

⟨*in_interface_def_list attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
⟨in_interface_def attributes⟩
} in_interface_def
```

⟨*in_interface_def attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
⟨out_interface_def_list attributes⟩
} out_interface_def_list
```

⟨*out_interface_def_list attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
⟨out_interface_def attributes⟩
} out_interface_def
```

⟨*out_interface_def attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
⟨selection_condition attributes⟩
} selection_condition
```

⟨*selection_condition attributes*⟩≡

6.9 Attributes for macros

⟨*Attributes*⟩+≡

```
@attributes {
⟨macro_def_list attributes⟩
} macro_def_list
```

⟨*macro_def_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*macro_def attributes*⟩
```
} macro_def
```

⟨*macro_def attributes*⟩≡

6.10 Attributes for functions

⟨*Include files*⟩+≡
```
#include "function_definition.h"
```

At the moment we need no top-level information about `function_def_lists`.

⟨*Attributes*⟩+≡
```
@attributes {
```
⟨*function_def_list attributes*⟩
```
} function_def_list
```

⟨*function_def_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
  function_definition *val;
```
⟨*function_def attributes*⟩
```
} function_def
```

⟨*function_def attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
  CaseList *inherited_val;
  CaseList *computed_val;
```
⟨*case_list attributes*⟩
```
} case_list
```

⟨*case_list attributes*⟩≡

```
⟨Attributes⟩+≡
  @attributes {
    rsml_case *val;
  ⟨case attributes⟩
  } case

⟨case attributes⟩≡
```

6.11 Attributes for transition busses

These declarations come in two parts: functional-style mechanisms for the sake of theory translations into EVES and PVS; mechanisms, for the sake of SPIN, which involve reading and setting information in a global variable `glob`. The stuff with global variables seems unavoidable, and some of the functional stuff could probably be simplified by piggy-backing onto it. (That's for the future.)

```
⟨Include files⟩+≡
  #include "transition_value.h"
  #include "event_value.h"
  #include "formula_table.h"
  #include "s_list_table.h"
```

The `code` attribute associates a unique integer with the transition defined by a `transition_def`. (I haven't yet done anything about transition bus definitions.)

By the "event tautology test" we mean the check that disjoining the conditions of all transitions triggered by any given state event forms a tautology. FLAW 1: For now, we don't keep track of which transition is associated with which of the disjuncts. To generate more helpful error reports we need to do that, so the attributes of type `formula_table` should be replaced by attributes of type `str_tagged_table` (see `str_tagged_table.h`).

For the top-level `transition_or_..._list`, the attribute `computed_etest` will synthesize a `formula_table` that associates the name of each event with the disjunction of all the conditions associated with that event. The formula entries in this table are candidate theorems (although, as noted above, we should only be making entries for the "state" events).

The construction of `computed_etest` is boilerplate: `inherited_etests` is passed down, modified as it goes along, and the final result is synthesized up in `computed_etests`.

The initial inherited table is constructed from the list of all event names (is it here that we should cut down to the list of "state" events?) by associating each of them with the condition "false." The list of event names comes from the `computed_events` attributed of `event_def_list`.

FLAW 2 : A transition is permitted to have no trigger. I'm not sure what the semantics of that is supposed to be. We currently associate such a transition with a special event, the `NullEvent`, whose name is the empty string—i.e., a `Str` whose associated string is `""`.

By the "condition E-and-E test" we mean a check that the exit conditions for each conditional state form an exhaustive and exclusive set. We generate these in a way analogous to the event tautology tests.

The relevant attributes are the computed and inherited ctests. However, the values of these attributes are s_list_tables, not formula_tables. (And the etests should also be changed to s_list_tables.)

The transition_def attributes state_def_done and glob: Setting transition_def.glob will manipulate the global variable glob. These manipulations must be delayed until the relevant parts of the global have been initialized by all the state_def productions. So, we will define transition_def.glob to depend on transition_def.state_def_done, and state_def_done will be an inherited attribute (in fact, auto-inherited). At the very top level t_or_tbd_list, inheritance of state_def_done will explicitly depend on state_def.glob, which achieves the necessary delay.

⟨*Auto-attributes*⟩≡

```
@autoinh state_def_done
```

⟨*Attributes*⟩+≡

```
@attributes {

        formula_table *inherited_etests;
        formula_table *computed_etests;

        s_list_table *inherited_ctests;
        s_list_table *computed_ctests;

        int state_def_done;
```

⟨*transition_or_transition_bus_def_list attributes*⟩

```
} transition_or_transition_bus_def_list
```

⟨*transition_or_transition_bus_def_list attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {

        formula_table *inherited_etests;
        formula_table *computed_etests;

        s_list_table *inherited_ctests;
        s_list_table *computed_ctests;

        int state_def_done;
```

⟨*transition_or_transition_bus_def attributes*⟩

```
} transition_or_transition_bus_def
```

⟨*transition_or_transition_bus_def attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {

        formula_table *inherited_etests;
        formula_table *computed_etests;

        s_list_table *inherited_ctests;
        s_list_table *computed_ctests;

        int state_def_done;
```
⟨*transition_bus_def attributes*⟩
```
} transition_bus_def
```

⟨*transition_bus_def attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {

        formula_table *inherited_etests;
        formula_table *computed_etests;

        s_list_table *inherited_ctests;
        s_list_table *computed_ctests;

        int state_def_done;
```
⟨*transition_def_list attributes*⟩
```
} transition_def_list
```

⟨*transition_def_list attributes*⟩≡

`from_code` and `to_code` are the unique integer codes of the source and destination states of the transition. `leaves_code` is the integer code of the state that the transition arrow truly leaves.

FLAW: The value of `event` shouldn't be an `event_value`. It should contain one or both of the `Str` naming the event or the integer code of the event.

⟨*Attributes*⟩+≡
```
@attributes {

        Str *trigger_name;
        S_exp *fml;
        int code;
```

```
        transition_value *val;

        int state_def_done;
        int from_code;
        Str *to_name;
        int to_code;
        Str *from_name;
        int leaves_code;
        Str *leaves_name;
        StrList *actions_names;
        int glob;
```

⟨transition_def attributes⟩

```
} transition_def
```

⟨transition_def attributes⟩≡

If a transition has no triggering event, we associate it with the null event, whose name is "". We are not currently modeling timeout events; and to distinguish these from null events we currently model them with the event named "?". We treat the null event as an event that is always enabled; and, for now, do the same with the dummy event.

⟨C Macros⟩≡

```
#define DummyEventName new Str("?")
#define NullEventName new Str("")
```

⟨Attributes⟩+≡

```
@attributes {

        Str *str;
```

⟨transition_maybe_trigger attributes⟩

```
} transition_maybe_trigger
```

⟨transition_maybe_trigger attributes⟩≡

⟨Attributes⟩+≡

```
@attributes {
        S_exp *fml;
```

⟨transition_maybe_cond attributes⟩

```
} transition_maybe_cond
```

⟨transition_maybe_cond attributes⟩≡

⟨*Attributes*⟩+≡
```
@attributes {

        StrList *strlist;
```
⟨*transition_maybe_action attributes*⟩
```
} transition_maybe_action
```
⟨*transition_maybe_action attributes*⟩≡

6.12 Attributes for predicates

For now we dummy out macros and quantified expressions: Note that we'll have to maintain a symbol table in order to recognize the name of the macro.

I'm assuming that `x EQ_ONE_OF { a, b }` is supposed to be a short hand for `x = a or x = b` .

Finally, operators are simply associated with like-named strings. That won't be right. We may have to give them distinctive names to distinguishm real operations from others—or, at least, the PVS and EVES print functions will have to choose separate names for them.

⟨*Attributes*⟩+≡
```
@attributes {
        S_exp *fml;
```
⟨*pred_ref attributes*⟩
```
} pred_ref
```
⟨*pred_ref attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
        S_exp *fml;
```
⟨*predicate attributes*⟩
```
} predicate
```
⟨*predicate attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
        S_exp *fml;
```
⟨*neg_predicate attributes*⟩
```
} neg_predicate
```

⟨*neg_predicate attributes*⟩≡

⟨*Attributes*⟩+≡
```
  @attributes {
  ⟨macro_ref attributes⟩
  } macro_ref
```

⟨*macro_ref attributes*⟩≡

⟨*Attributes*⟩+≡
```
  @attributes {
    S_exp *fml;
  ⟨math_predicate attributes⟩
  } math_predicate
```

⟨*math_predicate attributes*⟩≡

⟨*Attributes*⟩+≡
```
  @attributes {
    Str *str;
  ⟨boolean_math_operator attributes⟩
  } boolean_math_operator
```

⟨*boolean_math_operator attributes*⟩≡

⟨*Attributes*⟩+≡
```
  @attributes {
          S_exp *fml;
  ⟨state_predicate attributes⟩
  } state_predicate
```

⟨*state_predicate attributes*⟩≡

⟨*Attributes*⟩+≡
```
  @attributes {
          S_exp *fml;
  ⟨forall_predicate attributes⟩
  } forall_predicate
```

⟨*forall_predicate attributes*⟩≡

```
⟨Attributes⟩+≡
  @attributes {
          S_exp *fml;
  ⟨exists_predicate attributes⟩
  } exists_predicate

⟨exists_predicate attributes⟩≡
```

6.13 Attributes for conditions

A **condition** is defined either by quantification (not currently implemented), by a boolean constant, or by a table. For now we'll ignore any presentation issues that may be presented by the tables, and simply glom each table into a formula.

Recall the syntax for tables:

```
condition               : ...

                        | TABLE
                          row_list
                          END TABLE
                        | ...
                        ;

row_list                : pred_ref ':' truth_value_list ';'
                        | row_list pred_ref ':' truth_value_list ';'
                        ;

truth_value_list        : truth_value
                        | truth_value truth_value_list
                        ;

truth_value             : 'T'
                        | 'F'
                        | '.'
                        ;
```

The calculation is straightforward: For each `truth_value_list` the attribute `tv_list` synthesizes the corresponding sequence of of `TruthValues`. Then, letting `pred_ref.fml` point to `Q`,

```
formula_row(Q,truth_value_list.tv_list)
```

calculates the corresponding sequence of the formulas `Q`, `not Q`, and `true`. (See `conditions.h]`.)

Each [[row_list.fml_list contains a sequence of formulas representing "the table so far": For example, if a row_list.0 is

```
row_list.1 pred_ref ':' truth_value_list ';'
```

we generate the pointwise and of row_list.1.fml_list and formula_row(...).

Finally, condition.fml is the Or of the top-level row_list.fml_list.

We have to be careful to avoid getting in trouble from the reversing property of the prepending list constructor.

⟨*Include files*⟩+≡
```
#include "conditions.h"
```

⟨*Attributes*⟩+≡
```
@attributes {
        S_exp *fml;
⟨condition attributes⟩
} condition
```

⟨*condition attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
        S_exp_list *fml_list;
⟨row_list attributes⟩
} row_list
```

⟨*row_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
        TruthValueList *tv_list;
⟨truth_value_list attributes⟩
} truth_value_list
```

⟨*truth_value_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
@attributes {
        TruthValue val;
⟨truth_value attributes⟩
} truth_value
```

⟨*truth_value attributes*⟩≡

6.14 Attributes for expressions

The `val` of an `expression`, `simple_expression`, or `function_ref` is its representation as an s-expression. As noted in `function_definition.h`, it's not clear what data structure should be used for variables and local variable lists. For now, I'll represent local variables as s-expressions of variables (i.e., terms that are not applications).

```
⟨Attributes⟩+≡
  @attributes {
    S_exp *val;
  ⟨expression attributes⟩
  } expression
```

⟨*expression attributes*⟩≡

```
⟨Attributes⟩+≡
  @attributes {
    S_exp *val;
  ⟨simple_expression attributes⟩
  } simple_expression
```

⟨*simple_expression attributes*⟩≡

```
⟨Attributes⟩+≡
  @attributes {
    S_exp *val;
  ⟨expression attributes⟩
  ⟨function_ref attributes⟩
  } function_ref
```

⟨*function_ref attributes*⟩≡

```
⟨Attributes⟩+≡
  @attributes {
    S_exp_list *inherited_val;
    S_exp_list *computed_val;
  ⟨parameter_list attributes⟩
  } parameter_list
```

⟨*parameter_list attributes*⟩≡

⟨*Attributes*⟩+≡
```
  @attributes {
    S_exp *val;
```
  ⟨*parameter attributes*⟩
```
  } parameter
```

⟨*parameter attributes*⟩≡

⟨*Attributes*⟩+≡
```
  @attributes {
    S_exp_list *inherited_val;
    S_exp_list *computed_val;
```
  ⟨*local_var_list attributes*⟩
```
  } local_var_list
```

⟨*local_var_list attributes*⟩≡

6.15 The various kinds of names

Names are tokens. The attributes `str`, `as_S_exp`, and `code` are defined during parsing. (See `grammar.l`.)

⟨*Attributes*⟩+≡
```
  @attributes {
    Str *str;
    S_exp *as_S_exp;
```
    ⟨*STATE_NAME attributes*⟩
```
  } STATE_NAME
```

⟨*STATE_NAME attributes*⟩≡

⟨*Attributes*⟩+≡
```
  @attributes {
    Str *str;
```
    ⟨*SYSTEM_NAME attributes*⟩
```
  } SYSTEM_NAME
```

⟨*SYSTEM_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
  Str *str;
  ⟨COMPONENT_NAME attributes⟩
} COMPONENT_NAME
```

⟨*COMPONENT_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
  Str *str;
  ⟨EVENT_NAME attributes⟩
} EVENT_NAME
```

⟨*EVENT_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
  Str *str;
  ⟨OUT_INTERFACE_NAME attributes⟩
} OUT_INTERFACE_NAME
```

⟨*OUT_INTERFACE_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
  Str *str;
  ⟨IN_INTERFACE_NAME attributes⟩
} IN_INTERFACE_NAME
```

⟨*IN_INTERFACE_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
```

```
    Str *str;
    S_exp *as_S_exp;
    ⟨CONSTANT_NAME attributes⟩
  } CONSTANT_NAME
```

⟨*CONSTANT_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
  @attributes {
    Str *str;
    S_exp *as_S_exp;
    ⟨IN_VAR_NAME attributes⟩
  } IN_VAR_NAME
```

⟨*IN_VAR_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
  @attributes {
    Str *str;
    S_exp *as_S_exp;
    ⟨OUT_VAR_NAME attributes⟩
  } OUT_VAR_NAME
```

⟨*OUT_VAR_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
  @attributes {
    Str *str;
    ⟨FUNCTION_NAME attributes⟩
  } FUNCTION_NAME
```

⟨*FUNCTION_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
  @attributes {
    Str *str;
    S_exp *as_S_exp;
    ⟨LOCAL_VAR_NAME attributes⟩
  } LOCAL_VAR_NAME
```

⟨*LOCAL_VAR_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
  Str *str;
  ⟨MACRO_NAME attributes⟩
} MACRO_NAME
```

⟨*MACRO_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
  Str *str;
  ⟨TRANSITION_NAME attributes⟩
} TRANSITION_NAME
```

⟨*TRANSITION_NAME attributes*⟩≡

⟨*Attributes*⟩+≡

```
@attributes {
  Str *str;
  ⟨TRANSITIONBUS_NAME attributes⟩
} TRANSITIONBUS_NAME
```

⟨*TRANSITIONBUS_NAME attributes*⟩≡

Chapter 7

Attribute definitions

Chapter 6 contains a master file explicitly breaking the "Attributes" module into its constituents, one module per symbol. It also contains the definition of basic, globally useful attributes. Attributes specific to particular translations are defined in the chapters devoted to those translations.

The *definitions* of attributes, which are gathered in the "rules" modules, are organized similarly, one module per rule. The file `grammar-productions.nw`, not included in this documentation, explicitly breaks the "rules" module into constituents, one module per production. It is omitted from this documentation because that's all it does—it does not define any attributes.

The long and ungainly module names are an unfortunate necessity, since the only convenient way to name each module uniquely is to name it with the full text of the corresponding production. However, the names are never touched by human hands: the file `grammar-productions.nw` is automatically generated. The file `equations.nw`, also automatically generated, contains a list of names of all the modules and serves as part of the developer's skeleton.

7.1 Attributes for system

⟨*system:SYSTEM SYSTEM_NAME ':' component_list END SYSTEM SECTION_SEPARATOR* ⟩+≡

⟨*component_list:*⟩≡

⟨*component_list:component_list component_def* ⟩≡

7.2 Attributes for components

⟨*component_def:COMPONENT COMPONENT_NAME ':' END COMPONENT* ⟩≡

```
@i @component_def.in_top_state@ = true_S_ptr;
```

The `long_case` is the following rule, which is too long to turn into a module name:

```
component_def             : COMPONENT COMPONENT_NAME ':'
                            END COMPONENT
                          | COMPONENT COMPONENT_NAME ':'
                             state_def
                             constant_def_list
                             event_def_list
                             in_variable_def_list
                             out_variable_def_list
                             in_interface_def_list
                             out_interface_def_list
                             macro_def_list
                             function_def_list
                             transition_or_transition_bus_def_list
                             END COMPONENT
                          ;
```

The `inherited_events` of `event_def_list` is the empty list. That's the boiler-plate way to kick off accumulating all declared events in the `computed_events` of `event_def_list`. NOTE: I'm not now distinguishing between the various kinds of events (state events, variable events, etc.)—they all go in the list.

The `inherited_etests` of `t_or_tbd_list` is the formula table whose keys are the `computed_events` of `event_def_list` and whose associated formulas are all false.

QUESTION: Should we automatically add `NullEvent` to this list? I'm doing nothing about that for now.

NOTE ALSO: I'm not adding the `DummyEvent`, which will mean that, for now, timeout events (which are currently dummied out) will be ignored.

The purpose of the (auto-inherited) `state_def_done` attribute is to delay certain calculations for `transition_defs`. The effect of effect of calling `glob.set_leq()` is a side effect on `glob`, storing the reflexive transitive closure of `child_of` in `leq`.

⟨*component_def:long_case* ⟩≡

```
@i @state_def.ancestor@ = new Str("");

@i @component_def.in_top_state@ =
        @state_def.parent_as_S_exp@->Equals(true_S_ptr);

@i @transition_or_transition_bus_def_list.inherited_etests@ =
        new formula_table(StateEventsToStrList(
                                @event_def_list.computed_events@),
```

```
                    false_S_ptr);

@i @transition_or_transition_bus_def_list.inherited_ctests@ =
       new s_list_table(StateValListToStrList(
                            @state_def.conditionals_so_far@),
                    new S_exp_list);

 @e transition_or_transition_bus_def_list.state_def_done:
      state_def.glob;

      @transition_or_transition_bus_def_list.state_def_done@ =
            glob.set_leq();

@i @event_def_list.inherited_events@ = EmptyEventValueList;
```

⟨*component_location:COMPONENT_NAME* ⟩≡

⟨*component_location:EXTERNAL* ⟩≡

7.3 Attributes for states

7.3.1 Auxiliary functions

If `*parent` is `P` and `*children` is (`child1`, `child2`, ...) then `many_imply_one(parent,children)` is a pointer to the list (`child1 -> P`, `child -> P`, ...).

If `mutual_exclusion(siblings)` points to a list asserting that (conjoined) asserts that the formulas in `*siblings` are mutually exclusive.

These functions are defined in `s_exp_utilities.h`

7.3.2 Definitions

⟨*state_def:OR_STATE_TOKEN STATE_NAME DEFAULT STATE_NAME ':' state_charts_list END OR_STATE_TOKEN* ⟩≡

```
@i @state_charts_list.ancestor@ = @STATE_NAME.0.str@;

@i @state_def.code@ = rsml.get_state_code();

@i @state_def.val@ = new state_value(@STATE_NAME.0.str@,
                                     OrState,
                                     @state_def.code@);
```

```
@e state_def.glob : state_def.val state_charts_list.glob
                         state_charts_list.comp_vals
                         state_def.code;
        @state_def.glob@ =
                glob.set_state_code(@state_def.val@) +
                glob.set_child_of(@state_charts_list.comp_vals@,
                                  @state_def.code@);

@i @state_charts_list.inherited_vals@ = (StateValueList *)0;

@i @state_def.parent_as_S_exp@ =  @STATE_NAME.as_S_exp@;

@i @state_def.children_as_S_exps@ =
        state_vals_to_s_exps(@state_charts_list.comp_vals@);

@i @state_def.parent_implies_children@ =
        @state_def.parent_as_S_exp@->
                Implies(@state_def.children_as_S_exps@->Or());

@i @state_def.children_imply_parent@ =
        many_op_one(@state_def.children_as_S_exps@,
                IMPLIES_SIGN,
                @state_def.parent_as_S_exp@);

@i @state_def.incompatible_siblings@ =
        mutual_exclusion(@state_def.children_as_S_exps@);

@i @state_def.conditionals_so_far@ =
        @state_charts_list.conditionals_so_far@;
```

⟨*state_def:AND_STATE_TOKEN STATE_NAME ':' state_charts_list END AND_STATE_TOKEN* ⟩≡

```
@i @state_charts_list.ancestor@ = @STATE_NAME.0.str@;

@i @state_def.code@ = rsml.get_state_code();

@i @state_def.val@ = new state_value(@STATE_NAME.0.str@,
                                     AndState,
                                     @state_def.code@);

@e state_def.glob : state_def.val state_charts_list.glob
```

```
                              state_charts_list.comp_vals
                              state_def.code;
          @state_def.glob@ =
                    glob.set_state_code(@state_def.val@) +
                    glob.set_child_of(@state_charts_list.comp_vals@,
                                      @state_def.code@);

@i @state_charts_list.inherited_vals@ = (StateValueList *)0;

@i @state_def.parent_as_S_exp@ =  @STATE_NAME.as_S_exp@;

@i @state_def.children_as_S_exps@ =
          state_vals_to_s_exps(@state_charts_list.comp_vals@);

@i @state_def.parent_implies_children@ =
          @state_def.parent_as_S_exp@->
                    Implies(@state_def.children_as_S_exps@->And());

@i @state_def.children_imply_parent@ =
          many_op_one(@state_def.children_as_S_exps@,
                    IMPLIES_SIGN,
                    @state_def.parent_as_S_exp@);

@i @state_def.incompatible_siblings@ = Singleton(true_S_ptr);

@i @state_def.conditionals_so_far@ =
          @state_charts_list.conditionals_so_far@;
```

⟨*state_def:OR_ARRAY STATE_NAME '[' INTEGER ']' ':' state_charts_list END OR_ARRAY*⟩≡

```
@i @state_charts_list.ancestor@ = @STATE_NAME.0.str@;

@i @state_def.code@ = rsml.get_state_code();

@i @state_def.val@ = new state_value(@STATE_NAME.0.str@,
                                     OrArrayState,
                                     @state_def.code@);

@e state_def.glob : state_def.val state_charts_list.glob
                    state_charts_list.comp_vals
                    state_def.code;
```

```
        @state_def.glob@ =
                glob.set_state_code(@state_def.val@) +
                glob.set_child_of(@state_charts_list.comp_vals@,
                                  @state_def.code@);

@i @state_charts_list.inherited_vals@ = (StateValueList *)0;

@i @state_def.parent_as_S_exp@ =  @STATE_NAME.as_S_exp@;

@i @state_def.children_as_S_exps@ =
        state_vals_to_s_exps(@state_charts_list.comp_vals@);

@i @state_def.parent_implies_children@ =
        @state_def.parent_as_S_exp@->
                Implies(@state_def.children_as_S_exps@->Or());

@i @state_def.children_imply_parent@ =
        many_op_one(@state_def.children_as_S_exps@,
                IMPLIES_SIGN,
                @state_def.parent_as_S_exp@);

@i @state_def.incompatible_siblings@ =
        mutual_exclusion(@state_def.children_as_S_exps@);

@i @state_def.conditionals_so_far@ =
        @state_charts_list.conditionals_so_far@;
```

⟨*state_def:AND_ARRAY STATE_NAME '[' INTEGER ']' ':' state_charts_list END AND_ARRAY*⟩≡

```
@i @state_charts_list.ancestor@ = @STATE_NAME.0.str@;

@i @state_def.code@ = rsml.get_state_code();

@i @state_def.val@ = new state_value(@STATE_NAME.0.str@,
                                     AndArrayState,
                                     @state_def.code@);

@e state_def.glob : state_def.val state_charts_list.glob
                    state_charts_list.comp_vals
                    state_def.code;
        @state_def.glob@ =
```

```
                    glob.set_state_code(@state_def.val@) +
                    glob.set_child_of(@state_charts_list.comp_vals@,
                                      @state_def.code@);

@i @state_charts_list.inherited_vals@ = (StateValueList *)0;

@i @state_def.parent_as_S_exp@ =  @STATE_NAME.as_S_exp@;

@i @state_def.children_as_S_exps@ =
        state_vals_to_s_exps(@state_charts_list.comp_vals@);

@i @state_def.parent_implies_children@ =
        @state_def.parent_as_S_exp@->
                Implies(@state_def.children_as_S_exps@->And());

@i @state_def.children_imply_parent@ =
        many_op_one(@state_def.children_as_S_exps@,
                IMPLIES_SIGN,
                @state_def.parent_as_S_exp@);

@i @state_def.incompatible_siblings@ = Singleton(true_S_ptr);

@i @state_def.conditionals_so_far@ =
        @state_charts_list.conditionals_so_far@;
```

⟨state_def:ATOMIC STATE_NAME ':'⟩≡

```
@i @state_def.code@ = rsml.get_state_code();

@i @state_def.val@ = new state_value(@STATE_NAME.0.str@,
                                     AtomicState,
                                     @state_def.code@);

@i @state_def.glob@ = glob.set_state_code(@state_def.val@);

@i @state_def.parent_as_S_exp@ =  @STATE_NAME.as_S_exp@;

@i @state_def.children_as_S_exps@ = new S_exp_list;

@i @state_def.parent_implies_children@ = true_S_ptr;

@i @state_def.children_imply_parent@ = Singleton(true_S_ptr);
```

```
@i @state_def.incompatible_siblings@ = Singleton(true_S_ptr);

@i @state_def.conditionals_so_far@ = EmptyStateValueList;
```

⟨*state_def:CONDITIONAL STATE_NAME ':'*⟩≡

```
@i @state_def.code@ = rsml.get_state_code();

@i @state_def.val@ = new state_value(@STATE_NAME.0.str@,
                                     ConditionalState,
                                     @state_def.code@);

@i @state_def.glob@ = glob.set_state_code(@state_def.val@);

@i @state_def.parent_as_S_exp@ = @STATE_NAME.as_S_exp@;

@i @state_def.children_as_S_exps@ = new S_exp_list;

@i @state_def.parent_implies_children@ = true_S_ptr;

@i @state_def.children_imply_parent@ = Singleton(true_S_ptr);

@i @state_def.incompatible_siblings@ = Singleton(true_S_ptr);

@i @state_def.conditionals_so_far@ =
        new StateValueList(*@state_def.val@);
```

⟨*state_charts_list:state_def*⟩≡

```
@i @state_def.ancestor@ = @state_charts_list.ancestor@;

@i @state_charts_list.comp_vals@ =
         new StateValueList(*@state_def.val@,
                            @state_charts_list.inherited_vals@);

@e state_charts_list.glob : state_def.glob;
        @state_charts_list.glob@ = 0;

@i @state_charts_list.conditionals_so_far@ =
        @state_def.conditionals_so_far@;
```

⟨*state_charts_list:state_charts_list state_def*⟩≡

```
@i @state_def.ancestor@ = @state_charts_list.0.ancestor@;

@i @state_charts_list.1.ancestor@ = @state_charts_list.0.ancestor@;

@i @state_charts_list.1.inherited_vals@ =
        new SimpleList<state_value>(*@state_def.val@,
                        @state_charts_list.inherited_vals@);

@i @state_charts_list.comp_vals@ =
        @state_charts_list.1.comp_vals@;

@e state_charts_list.0.glob : state_def.glob
        state_charts_list.1.glob;
        @state_charts_list.glob@ = 0;

@i @state_charts_list.0.conditionals_so_far@ =
        @state_charts_list.1.conditionals_so_far@->
                Append(@state_def.conditionals_so_far@);
```

⟨*state_list:STATE_NAME*⟩≡

```
@i @state_list.comp_exps@ =
        new S_exp_list(@STATE_NAME.as_S_exp@,
                        @state_list.inherited_exps@);
```

⟨*state_list:state_list ',' STATE_NAME*⟩≡

```
@i @state_list.0.comp_exps@ = @state_list.1.comp_exps@;
@i @state_list.1.inherited_exps@ =
        new S_exp_list(@STATE_NAME.as_S_exp@,
                        @state_list.0.inherited_exps@);
```

CURRENTLY UNIMPLEMENTED

⟨*simple_state_path:STATE_NAME*⟩≡

⟨*simple_state_path:simple_state_path '.' STATE_NAME*⟩≡

⟨*state_path:STATE_NAME '[' parameter ']'*⟩≡

⟨*state_path:STATE_NAME '[' parameter ']' '.' simple_state_path* ⟩≡

⟨*state_path:simple_state_path '.' STATE_NAME '[' parameter ']'* ⟩≡

⟨*state_path:simple_state_path '.' STATE_NAME '[' parameter ']' '.' simple_state_path* ⟩≡

⟨*state_path:simple_state_path* ⟩≡

⟨*state_path_ref:PREV '(' REAL ')' state_path* ⟩≡

⟨*state_path_ref:state_path* ⟩≡

7.3.3 Bodies of auxiliary functions

⟨*Bodies*⟩≡

7.4 Attributes for constants

The `val` of a `REAL` is a `scaled_integer`. The `val` of a `CONSTANT_NAME` is a `Str`. Both attributes are computed during the first pass. To generate a declaration for the constant and the defining equation we don't need any more information.

⟨*constant_def:CONSTANT CONSTANT_NAME ':' VALUE ':' REAL END CONSTANT* ⟩≡

⟨*constant_def:CONSTANT CONSTANT_NAME ':' VALUE ':' '-' REAL END CONSTANT* ⟩≡

The `exp` of a `constant_ref` is a pointer to an s-expression representing the constant as a 0-ary function.

⟨*constant_ref:CONSTANT_NAME* ⟩≡

```
@i @constant_ref.exp@ = @CONSTANT_NAME.as_S_exp@;
```

⟨*constant_def_list:*⟩≡

⟨*constant_def_list:constant_def_list constant_def* ⟩≡

7.5 Attributes for variables

⟨*in_variable_def:IN_VARIABLE IN_VAR_NAME ':' TYPE ':' NUMERIC expected_min expected_max min_gran max_gran END IN_VARIABLE* ⟩≡

⟨*out_variable_def:OUT_VARIABLE OUT_VAR_NAME ':' TYPE ':' NUMERIC expected_min expected_max min_gran max_gran ASSIGNMENT ':' expression TRIGGER ':' event_ref END OUT_VARIABLE* ⟩≡

⟨*expected_min:*⟩≡

⟨*expected_min:EXPECTED_MIN ':' REAL* ⟩≡

⟨*expected_max:*⟩≡

⟨*expected_max:EXPECTED_MAX ':' REAL* ⟩≡

⟨*min_gran:*⟩≡

⟨*min_gran:MIN_GRAN ':' REAL* ⟩≡

⟨*max_gran:*⟩≡

⟨*max_gran:MAX_GRAN ':' REAL* ⟩≡

⟨*in_variable_ref:IN_VAR_NAME* ⟩≡

```
@i @in_variable_ref.str@ = @IN_VAR_NAME.str@;
@i @in_variable_ref.as_S_exp@ = @IN_VAR_NAME.as_S_exp@;
```

⟨*out_variable_ref:OUT_VAR_NAME* ⟩≡
```
@i @out_variable_ref.str@ = @OUT_VAR_NAME.str@;
@i @out_variable_ref.as_S_exp@ = @OUT_VAR_NAME.as_S_exp@;
```

⟨*in_variable_ass:IN_VAR_NAME* ⟩≡
```
@i @in_variable_ass.str@ = @IN_VAR_NAME.str@;
```

⟨*out_variable_ass:OUT_VAR_NAME* ⟩≡
```
@i @out_variable_ass.str@ = @OUT_VAR_NAME.str@;
```

⟨*in_variable_def_list:*⟩≡

⟨*in_variable_def_list:in_variable_def_list in_variable_def*
⟩≡

⟨*out_variable_def_list:*⟩≡

⟨*out_variable_def_list:out_variable_def_list out_variable_def*
⟩≡

⟨*in_variable_list:*⟩≡
```
@i @in_variable_list.computed_val@ = EmptyStrList;
```

⟨*in_variable_list:in_variable_ass* ⟩≡
```
@i @in_variable_list.computed_val@ =
        new StrList(@in_variable_ass.str@);
```

⟨*in_variable_list:in_variable_list ',' in_variable_ass* ⟩≡
```
@i @in_variable_list.0.computed_val@ =
        @in_variable_list.1.computed_val@;
@i @in_variable_list.1.inherited_val@ =
        new StrList(@in_variable_ass.str@,
                    @in_variable_list.0.inherited_val@);
```

⟨*variable:out_variable_ref* ⟩≡
```
@i @variable.str@ = @out_variable_ref.str@;
@i @variable.as_S_exp@ = @out_variable_ref.as_S_exp@;
```

⟨*variable:in_variable_ref*⟩≡

```
@i @variable.str@ = @in_variable_ref.str@;
@i @variable.as_S_exp@ = @in_variable_ref.as_S_exp@;
```

NOTICE that the attributes for this instance of `variable_ref` are dummied out.

⟨*variable_ref:PREV '(' REAL ')' in_variable_ref*⟩≡

```
@i @variable_ref.str@ = DummyStr;
@i @variable_ref.as_S_exp@ = DummyFormula;
```

⟨*variable_ref:in_variable_ref*⟩≡

```
@i @variable_ref.str@ = @in_variable_ref.str@;
@i @variable_ref.as_S_exp@ = @in_variable_ref.as_S_exp@;
```

7.6 Attributes for events

Boilerplate.

⟨*event_def_list:*⟩≡

```
@i @event_def_list.computed_events@ =
        @event_def_list.inherited_events@;
```

⟨*event_def_list:event_def_list event_def*⟩≡

```
@i @event_def_list.0.computed_events@ =
        @event_def_list.1.computed_events@;

@i @event_def_list.1.inherited_events@ =
        new EventValueList(*@event_def.val@,
                @event_def_list.0.inherited_events@);
```

⟨*event_def:EVENT EVENT_NAME STATE ';'*⟩≡

```
@i @event_def.code@ = rsml.get_event_code();

@i @event_def.val@ = new event_value(@EVENT_NAME.str@,StateEvent);
```

⟨*event_def:EVENT EVENT_NAME VARIABLE ';'*⟩≡

```
@i @event_def.code@ = rsml.get_event_code();

@i @event_def.val@ = new event_value(@EVENT_NAME.str@,VariableEvent);
```

⟨*event_def:EVENT EVENT_NAME INTERFACE ';'*⟩≡

```
@i @event_def.code@ = rsml.get_event_code();

@i @event_def.val@ = new event_value(@EVENT_NAME.str@,InterfaceEvent);
```

⟨*event_ref:EVENT_NAME*⟩≡

```
@i @event_ref.str@ = @EVENT_NAME.str@;
```

⟨*event_list:*⟩≡

```
@i @event_list.computed_events@ =
        @event_list.inherited_events@;
```

⟨*event_list:event_ref*⟩≡

```
@i @event_list.computed_events@ =
        new StrList(@event_ref.str@,
                @event_list.inherited_events@);
```

⟨*event_list:event_list ',' event_ref*⟩≡

```
@i @event_list.0.computed_events@ =
        @event_list.1.computed_events@;

@i @event_list.1.inherited_events@ =
        new StrList(@event_ref.str@,
                @event_list.0.inherited_events@);
```

7.7 Attributes for interfaces

⟨*in_interface_def_list:*⟩≡

⟨*in_interface_def_list:in_interface_def_list in_interface_def*
⟩≡

⟨*in_interface_def:IN_INTERFACE IN_INTERFACE_NAME ':' SOURCE
':' component_location TRIGGER ':' RECEIVE '(' in_variable_list
')' selection_condition ACTION ':' event_list END IN_INTERFACE*
⟩≡

```
@i @event_list.inherited_events@ = EmptyStrList;

@i @in_variable_list.inherited_val@ = EmptyStrList;
```

⟨*out_interface_def_list:*⟩≡

⟨*out_interface_def_list:out_interface_def_list out_interface_def*
⟩≡

⟨*out_interface_def:OUT_INTERFACE OUT_INTERFACE_NAME ':'
DESTINATION ':' component_location TRIGGER ':' event_ref
selection_condition ACTION ':' SEND '(' parameter_list ')'
END OUT_INTERFACE* ⟩≡

```
@i @parameter_list.inherited_val@ = new S_exp_list;
```

⟨*out_interface_def:OUT_INTERFACE OUT_INTERFACE_NAME ':'
DESTINATION ':' component_location ',' IN_INTERFACE_NAME
TRIGGER ':' event_ref selection_condition ACTION ':' SEND
'(' parameter_list ')' END OUT_INTERFACE* ⟩≡

```
@i @parameter_list.inherited_val@ = new S_exp_list;
```

⟨*selection_condition:*⟩≡

⟨*selection_condition:SELECTION ':' condition* ⟩≡

7.8 Attributes for macros

⟨*macro_def_list:*⟩≡

⟨*macro_def_list:macro_def_list macro_def* ⟩≡

⟨*macro_def:MACRO MACRO_NAME '(' local_var_list ')' ':' condition END MACRO* ⟩≡

```
        @i @local_var_list.inherited_val@ = new S_exp_list;
```

7.9 Attributes for functions

⟨*function_def_list:*⟩≡

⟨*function_def_list:function_def_list function_def* ⟩≡

⟨*function_def:FUNCTION FUNCTION_NAME '(' local_var_list ')' ':' RETURN ':' CASE case_list END CASE END FUNCTION* ⟩≡

```
        @i @local_var_list.inherited_val@ = new S_exp_list;
        @i @case_list.inherited_val@ = EmptyCaseList;

        @i @function_def.val@ =
                new function_definition(@FUNCTION_NAME.str@,
                        @local_var_list.computed_val@,
                        @case_list.computed_val@);
```

⟨*case_list:case* ⟩≡

```
        @i @case_list.computed_val@ =
                new CaseList(@case.val@);
```

⟨*case_list:case_list case* ⟩≡

```
        @i @case_list.0.computed_val@ = @case_list.1.computed_val@;
        @i @case_list.1.inherited_val@ =
                new CaseList(@case.val@,
                        @case_list.0.inherited_val@);
```

⟨*case:expression IF pred_ref ';'*⟩≡

```
        @i @case.val@ = new rsml_case(@pred_ref.fml@, @expression.val@);
```

7.10 Attributes for transitions and busses

⟨*Include files*⟩+≡

```
#include "formula_table_utilities.h"
#include "state_value_utilities.h"
#include "s_list_table_utilities.h"
```

Synthesis of `computed_etests` is boiler plate. Given

```
t_or_tbd_list0 = t_or_tbd_list1    t_or_tbd
```

there are two basic "flows":

- `t_or_tbd_list0.computed_etest  <-- t_or_tbd_list1.computed_etest`

  That is, all `t_or_tbd_list` nodes have the *same* computed etest, which ultimately flows up from the last item in the list.

- `t_or_tbd_list0.inherited_etest  --> t_or_tbd.inherited_etest`

  `t_or_tbd.inherited_etest + local_info --> t_or_tbd.computed_etest`

  `t_or_tbd.computed_etest --> t_or_tbd_list1.inherited_etest`

  That is, a `t_or_tbd` node takes the inherited etest of its parent, adds the local information from its declaration (the event/condition pair), and hands the result, as inherited information, to its sibling `t_or_tbd_list1`.

When we turn the corner, the empty production synthesizes up the same `EventTests` it inherits. Synthesis of `computed_ctests` is similar.

⟨*transition_or_transition_bus_def_list:*⟩≡

```
        @i @transition_or_transition_bus_def_list.computed_etests@ =
              @transition_or_transition_bus_def_list.inherited_etests@;

        @i @transition_or_transition_bus_def_list.computed_ctests@ =
              @transition_or_transition_bus_def_list.inherited_ctests@;
```

Here are the two flows in the bread-and-butter step described above (for the case in which the `t_or_tbd` turns out to be a `transition_def`).

⟨*transition_or_transition_bus_def_list:transition_or_transition_bus_def_list transition_or_transition_bus_def*⟩≡

```
@i @transition_or_transition_bus_def_list.0.computed_etests@ =
        @transition_or_transition_bus_def_list.1.computed_etests@;

@i @transition_or_transition_bus_def_list.1.inherited_etests@ =
    @transition_or_transition_bus_def.computed_etests@;

@i @transition_or_transition_bus_def.inherited_etests@ =
        @transition_or_transition_bus_def_list.0.inherited_etests@;

@i @transition_or_transition_bus_def_list.0.computed_ctests@ =
        @transition_or_transition_bus_def_list.1.computed_ctests@;

@i @transition_or_transition_bus_def_list.1.inherited_ctests@ =
    @transition_or_transition_bus_def.computed_ctests@;

@i @transition_or_transition_bus_def.inherited_ctests@ =
        @transition_or_transition_bus_def_list.0.inherited_ctests@;
```

In computing the `ctest` we want to add a formula only if the source state of the transition is a conditional state. In applying the **prepend** operation that qualification is applied automatically: the keys of the `s_list_table` consist only of the names of conditional states, and if we supply `prepend` with a different key the result is a no-op.

FLAW? Can we get in trouble here? Both `DisjoinToETests` and `AddToSTable` work by side-effect.

⟨*transition_or_transition_bus_def:transition_def*⟩≡

```
@i @transition_or_transition_bus_def.computed_etests@ =
        DisjoinToETests(@transition_def.trigger_name@,
                @transition_def.fml@,
                @transition_or_transition_bus_def.inherited_etests@);

@i @transition_or_transition_bus_def.computed_ctests@ =
        AddToSTable(@transition_def.from_name@,
                @transition_def.fml@,
                @transition_or_transition_bus_def.inherited_ctests@);
```

For the case in which the `t_or_tbd` turns out to be a `transition_bus_def` we have to go down another couple of layers, so this layer just pipes inherited and computed etests between parent and child.

⟨*transition_or_transition_bus_def:transition_bus_def*⟩≡

```
@i @transition_bus_def.inherited_etests@ =
       @transition_or_transition_bus_def.inherited_etests@;

@i @transition_or_transition_bus_def.computed_etests@ =
       @transition_bus_def.computed_etests@;

@i @transition_bus_def.inherited_ctests@ =
       @transition_or_transition_bus_def.inherited_ctests@;

@i @transition_or_transition_bus_def.computed_ctests@ =
       @transition_bus_def.computed_ctests@;
```

A transition_bus_def also pipes its etests between the parent and the child transition_def_list.

⟨*transition_bus_def:TRANSITIONBUS TRANSITIONBUS_NAME ':' transition_def_list END TRANSITIONBUS* ⟩≡

```
@i @transition_def_list.inherited_etests@ =
       @transition_bus_def.inherited_etests@;
@i @transition_bus_def.computed_etests@ =
       @transition_def_list.computed_etests@;

@i @transition_def_list.inherited_ctests@ =
       @transition_bus_def.inherited_ctests@;
@i @transition_bus_def.computed_ctests@ =
       @transition_def_list.computed_ctests@;
```

We use the same boilerplate to gather up the values in a transition_def_list as we did in a t_or_tbd_list.

⟨*transition_def_list:*⟩≡

```
@i @transition_def_list.computed_etests@ =
       @transition_def_list.inherited_etests@;

@i @transition_def_list.computed_ctests@ =
       @transition_def_list.inherited_ctests@;
```

⟨*transition_def_list:transition_def_list transition_def*
⟩≡

```
@i @transition_def_list.0.computed_etests@ =
       @transition_def_list.1.computed_etests@;
```

```
@i @transition_def_list.1.inherited_etests@ =
        DisjoinToETests(@transition_def.trigger_name@,
                @transition_def.fml@,
                @transition_def_list.0.inherited_etests@);

@i @transition_def_list.0.computed_ctests@ =
        @transition_def_list.1.computed_ctests@;

@i @transition_def_list.1.inherited_ctests@ =
        AddToSTable(@transition_def.from_name@,
                @transition_def.fml@,
                @transition_def_list.0.inherited_ctests@);
```

We bottom out with an individual **transition_def**, which supplies the material for the event/condition pair that will be inserted into the formula table. To get **transition_def.fml**, we conjoin the assertion "I am in the source state" with the **fml** defining the transition's condition.

FLAW: The **transition_maybe_action** and its use in defining **transition_def.val** are dummied out.

⟨*transition_def:TRANSITION TRANSITION_NAME FROM STATE_NAME TO STATE_NAME ':' transition_maybe_trigger transition_maybe_cond transition_maybe_action END TRANSITION* ⟩≡

```
@i @transition_def.code@ = rsml.get_transition_code();

@i @transition_def.fml@ =
        @STATE_NAME.0.as_S_exp@->And(@transition_maybe_cond.fml@);

@i @transition_def.trigger_name@ = @transition_maybe_trigger.str@;

@i @transition_def.from_name@ = @STATE_NAME.0.str@;

@e transition_def.from_code: transition_def.state_def_done
    STATE_NAME.0.str;

    @transition_def.from_code@ =
        glob.code_of_state(@STATE_NAME.0.str@);

@i @transition_def.to_name@ = @STATE_NAME.1.str@;

@e transition_def.to_code: transition_def.state_def_done
```

```
    STATE_NAME.1.str;

    @transition_def.to_code@ =
        glob.code_of_state(@STATE_NAME.1.str@);

@e transition_def.leaves_code: transition_def.state_def_done
     transition_def.from_code transition_def.to_code;

     @transition_def.leaves_code@ =
        glob.leaves(@transition_def.from_code@,
                    @transition_def.to_code@);

@e transition_def.leaves_name: transition_def.state_def_done
     transition_def.leaves_code;

    @transition_def.leaves_name@ =
        glob.state_at(@transition_def.leaves_code@)->str();

@i @transition_def.actions_names@ =
        @transition_maybe_action.strlist@;

@i @transition_def.val@ =
    new transition_value(@TRANSITION_NAME.str@,
                         @transition_def.trigger_name@,
                         @transition_maybe_cond.fml@,
                         @transition_def.from_code@,
                         @STATE_NAME.0.str@,
                         @transition_def.to_code@,
                         @STATE_NAME.1.str@,
                         @transition_def.leaves_code@,
                         @transition_def.leaves_name@,
                         @transition_def.actions_names@,
                         @transition_def.code@);

@e transition_def.glob: transition_def.state_def_done
         transition_def.val;
    @transition_def.glob@ =
        glob.set_transition_code(@transition_def.val@);
```

We model the absence of a triggering event as a transition triggered by the unique event whose name is "".

⟨*transition_maybe_trigger:*⟩≡

```
@i @transition_maybe_trigger.str@ = NullEventName;
```

The obvious thing.

⟨*transition_maybe_trigger:TRIGGER ':' event_ref* ⟩≡

```
@i @transition_maybe_trigger.str@ = @event_ref.str@;
```

We are not currently modeling timeout events, so this becomes a dummy named "?", so it can't become confused with a declared event or with the **NullEvent**.

⟨*transition_maybe_trigger:TRIGGER ':' TIMEOUT '(' simple_expression ',' expression ')'* ⟩≡

```
@i @transition_maybe_trigger.str@ = DummyEventName;
```

CHECK THIS OUT: I'm assuming that if there's no explicit condition, the condition defaults to TRUE.

⟨*transition_maybe_cond:*⟩≡

```
@i @transition_maybe_cond.fml@ =  true_S_ptr;
```

⟨*transition_maybe_cond:CONDITION ':' condition* ⟩≡

```
@i @transition_maybe_cond.fml@ = @condition.fml@;
```

The only attribute for actions lists their names.

⟨*transition_maybe_action:*⟩≡

```
@i @transition_maybe_action.strlist@ = EmptyStrList;
```

⟨*transition_maybe_action:ACTION ':' event_list* ⟩≡

```
@i @event_list.inherited_events@ = EmptyStrList;
@i @transition_maybe_action.strlist@ =
        @event_list.computed_events@;
```

7.11 Attributes for predicates

⟨*pred_ref:predicate* ⟩≡
```
@i @pred_ref.fml@ = @predicate.fml@;
```

⟨*predicate:neg_predicate* ⟩≡
```
@i @predicate.fml@ = @neg_predicate.fml@;
```

⟨*predicate:macro_ref* ⟩≡
```
@i @predicate.fml@ = DummyFormula;
```

⟨*predicate:math_predicate* ⟩≡
```
@i @predicate.fml@ = @math_predicate.fml@;
```

⟨*predicate:state_predicate* ⟩≡
```
@i @predicate.fml@ = @state_predicate.fml@;
```

⟨*predicate:forall_predicate* ⟩≡
```
@i @predicate.fml@ = @forall_predicate.fml@;
```

⟨*predicate:exists_predicate* ⟩≡
```
@i @predicate.fml@ = @exists_predicate.fml@;
```

⟨*predicate:TRUE_TOKEN* ⟩≡
```
@i @predicate.fml@ = true_S_ptr;
```

⟨*predicate:FALSE_TOKEN* ⟩≡
```
@i @predicate.fml@ = false_S_ptr;
```

⟨*neg_predicate:NOT pred_ref* ⟩≡
```
@i @neg_predicate.fml@ = @pred_ref.fml@->Not();
```

IGNORING MACROS.

⟨*macro_ref:MACRO_NAME '(' parameter_list ')'* ⟩≡

```
@i @parameter_list.inherited_val@ = new S_exp_list;
```

⟨*math_predicate:expression boolean_math_operator expression*
⟩≡

```
@i @math_predicate.fml@ =
        new Application(@boolean_math_operator.str@,
                Doubleton(@expression.0.val@,
                          @expression.1.val@));
```

⟨*math_predicate:simple_expression EQ_ONE_OF " parameter_list*
" ⟩≡

```
@i @parameter_list.inherited_val@ = new S_exp_list;
@i @math_predicate.fml@ =
        one_op_many(@simple_expression.val@,
                    EQ_SIGN,
                    @parameter_list.computed_val@)->Or();
```

⟨*boolean_math_operator:'='* ⟩≡

```
@i @boolean_math_operator.str@ = new Str("=");
```

⟨*boolean_math_operator:'¿'* ⟩≡

```
@i @boolean_math_operator.str@ = new Str("REAL_>");
```

⟨*boolean_math_operator:'¡'* ⟩≡

```
@i @boolean_math_operator.str@ = new Str("REAL_<");
```

⟨*boolean_math_operator:LESS_OR_EQUAL* ⟩≡

```
@i @boolean_math_operator.str@ = new Str("REAL_<=");
```

⟨*boolean_math_operator:GREATER_OR_EQUAL* ⟩≡

```
@i @boolean_math_operator.str@ = new Str("REAL_>=");
```

⟨*boolean_math_operator:NOT_EQUAL* ⟩≡

```
@i @boolean_math_operator.str@ = new Str("REAL_/=");
```

⟨*state_predicate:simple_state_path IN_STATE STATE_NAME* ⟩≡

```
@i @state_predicate.fml@ = @STATE_NAME.as_S_exp@;
```

Boilerplate: `state_list` inherits the empty list. It computes the list of all states in the sequence (as an `S_exp_list`), and we OR them together (that being my current, tenuous, understanding of the semantics).

⟨*state_predicate:simple_state_path IN_ONE_OF '' state_list ''*⟩≡

```
@i @state_list.inherited_exps@ = new S_exp_list;
@i @state_predicate.fml@ = @state_list.comp_exps@->Or() ;
```

IGNORING QUANTIFIERS.

⟨*forall_predicate:FORALL LOCAL_VAR_NAME '[' constant_ref ',' constant_ref ']' ',' pred_ref*⟩≡

```
@i @forall_predicate.fml@ = DummyFormula;
```

⟨*exists_predicate:EXISTS LOCAL_VAR_NAME '[' constant_ref ',' constant_ref ']' ',' pred_ref*⟩≡

```
@i @exists_predicate.fml@ = DummyFormula;
```

7.12 Attributes for conditions

⟨*condition:FORALL LOCAL_VAR_NAME '[' constant_ref ',' constant_ref ']' ',' condition*⟩≡

```
@i @condition.fml@ = DummyFormula;
```

⟨*condition:EXISTS LOCAL_VAR_NAME '[' constant_ref ',' constant_ref ']' ',' condition*⟩≡

```
@i @condition.fml@ = DummyFormula;
```

⟨*condition:TABLE row_list END TABLE*⟩≡

```
@i @condition.fml@ = @row_list.fml_list@->Or();
```

⟨*condition:TRUE_TOKEN*⟩≡

```
@i @condition.fml@ = true_S_ptr;
```

⟨*condition:FALSE_TOKEN*⟩≡

```
@i @condition.fml@ = false_S_ptr;
```

⟨*row_list:pred_ref ':' truth_value_list ';'* ⟩≡
```
        @i @row_list.fml_list@ =
                formula_row(@pred_ref.fml@, @truth_value_list.tv_list@);
```

⟨*row_list:row_list pred_ref ':' truth_value_list ';'* ⟩≡
```
        @i @row_list.0.fml_list@ =
                ptwise_And(@row_list.1.fml_list@,
                        formula_row(@pred_ref.fml@,
                                        @truth_value_list.tv_list@));
```

⟨*truth_value_list:truth_value* ⟩≡
```
        @i @truth_value_list.tv_list@ = new TruthValueList(@truth_value.val@);
```

⟨*truth_value_list:truth_value truth_value_list* ⟩≡
```
        @i @truth_value_list.0.tv_list@ =
                new TruthValueList(@truth_value.val@,
                                @truth_value_list.1.tv_list@);
```

⟨*truth_value:'T'* ⟩≡
```
        @i @truth_value.val@ = tv_true;
```

⟨*truth_value:'F'* ⟩≡
```
        @i @truth_value.val@ = tv_false;
```

⟨*truth_value:'.'* ⟩≡
```
        @i @truth_value.val@ = tv_dont_care;
```

7.13 Attributes for expressions

Note: I'm dummying out the various TIME expressions.

⟨*Ox Macros*⟩≡
```
  @macro BinaryAp(op,)
        @i @expression.0.val@ =
                new Application(new Str(op),
        new S_exp_list(@expression.1.val@,Singleton(@expression.2.val@)));
```

```
@end

@macro UnaryAp(op,)
        @i @expression.0.val@ =
                new Application(new Str(op),
                        Singleton(@expression.1.val@));
@end
```

⟨*expression:expression '*' expression* ⟩≡

```
        BinaryAp("REAL_*",)
```

⟨*expression:expression '/' expression* ⟩≡

```
        BinaryAp("REAL_/",)
```

⟨*expression:expression '+' expression* ⟩≡

```
        BinaryAp("REAL_+",)
```

⟨*expression:expression '-' expression* ⟩≡

```
        BinaryAp("REAL_-",)
```

⟨*expression:'-' expression %prec UMINUS* ⟩≡

```
        UnaryAp("REAL_NEG",)
```

⟨*expression:'(' expression ')'* ⟩≡

```
        @i @expression.0.val@ = @expression.1.val@;
```

⟨*expression:simple_expression* ⟩≡

```
        @i @expression.val@ = @simple_expression.val@;
```

⟨*simple_expression:variable_ref* ⟩≡

```
        @i @simple_expression.val@ = @variable_ref.as_S_exp@;
```

⟨*simple_expression:function_ref* ⟩≡

```
        @i @simple_expression.val@ = @function_ref.val@;
```

⟨*simple_expression:constant_ref*⟩≡

```
@i @simple_expression.val@ = @constant_ref.exp@;
```

⟨*simple_expression:LOCAL_VAR_NAME*⟩≡

```
@i @simple_expression.val@ = @LOCAL_VAR_NAME.as_S_exp@;
```

⟨*simple_expression:TIME*⟩≡

```
@i @simple_expression.val@ = DummyFormula;
```

⟨*simple_expression:TIME '(' event_ref ')'*⟩≡

```
@i @simple_expression.val@ = DummyFormula;
```

⟨*simple_expression:TIME '(' PREV '(' REAL ')' event_ref ')'*⟩≡

```
@i @simple_expression.val@ = DummyFormula;
```

⟨*simple_expression:TIME '(' variable_ref ')'*⟩≡

```
@i @simple_expression.val@ = DummyFormula;
```

⟨*simple_expression:TIME '(' ENTERED STATE_NAME ')'*⟩≡

```
@i @simple_expression.val@ = DummyFormula;
```

⟨*function_ref:FUNCTION_NAME '(' parameter_list ')'*⟩≡

```
@i @function_ref.val@ =
        new Application(@FUNCTION_NAME.str@,
                @parameter_list.computed_val@);
@i @parameter_list.inherited_val@ = new S_exp_list;
```

⟨*parameter_list:*⟩≡

```
@i @parameter_list.computed_val@ =
        @parameter_list.inherited_val@;
```

⟨*parameter_list:parameter*⟩≡

```
@i @parameter_list.computed_val@ =
        Singleton(@parameter.val@);
```

⟨*parameter_list:parameter_list ',' parameter* ⟩≡
```
@i @parameter_list.0.computed_val@ =
        @parameter_list.1.computed_val@;

@i @parameter_list.1.inherited_val@ =
        new S_exp_list(@parameter.val@,
                @parameter_list.0.inherited_val@);
```

⟨*parameter:expression* ⟩≡
```
@i @parameter.val@ = @expression.val@;
```

⟨*local_var_list:*⟩≡
```
@i @local_var_list.computed_val@ = @local_var_list.inherited_val@;
```

⟨*local_var_list:LOCAL_VAR_NAME* ⟩≡
```
@i @local_var_list.computed_val@ =
        Singleton(@LOCAL_VAR_NAME.as_S_exp@);
```

⟨*local_var_list:local_var_list ',' LOCAL_VAR_NAME* ⟩≡
```
@i @local_var_list.0.computed_val@ =
        @local_var_list.1.computed_val@;

@i @local_var_list.1.inherited_val@ =
        new S_exp_list(@LOCAL_VAR_NAME.as_S_exp@,
                @local_var_list.0.inherited_val@);
```

7.14 Attributes for names

The basic attribute, assigning a `Str` to the name, is computed during lexical analysis. (See `grammar.l`.)

⟨*STATE_NAME:NAME* ⟩≡

⟨*SYSTEM_NAME:NAME* ⟩≡

⟨*COMPONENT_NAME:NAME* ⟩≡

⟨*EVENT_NAME:NAME* ⟩≡

⟨*OUT_INTERFACE_NAME:NAME* ⟩≡

⟨*IN_INTERFACE_NAME:NAME* ⟩≡

⟨*CONSTANT_NAME:NAME* ⟩≡

⟨*IN_VAR_NAME:NAME* ⟩≡

⟨*OUT_VAR_NAME:NAME* ⟩≡

⟨*FUNCTION_NAME:NAME* ⟩≡

⟨*LOCAL_VAR_NAME:NAME* ⟩≡

⟨*MACRO_NAME:NAME* ⟩≡

⟨*TRANSITION_NAME:NAME* ⟩≡

⟨*TRANSITIONBUS_NAME:NAME* ⟩≡

Chapter 8

EVES

This chapter contains a partly filled in template for integrating the EVES prover. We define two traversals in order to experiment with two different ways of encoding the state hierarchy into EVES.

8.1 Currently implemented

In the E translation we currently generate the following EVES theory:

- Unconditional states are declared as boolean constants, and axioms define the hierarchy relation (in particular, the "in state" and "in one of" predicates).
- RSML constants are declared and axioms define their initial values.
- RSML in variables are declared as unspecified real-valued constants.

  Note that RSML variables should really be modeled as real-valued functions of time.
- The definitions of RSML functions are entered as axioms.

Out variables are ignored. Timeout events are ignored, and predicates referring to the timed properties of variables are dummied to the formula "false." (It might be more informative to dummy them as distinct unspecified constants with suitable mnemonic names.)

We generate candidate theorems corresponding to the following tests:

- The table defining each rsml function is complete and consistent.
- The conditions for exiting from each conditional state are complete and consistent.
- For each event, the conditions for transitions triggered by the event are complete.

8.2 Preliminaries

The traversal for the EVES translation is **eves2** (for historical reasons—the other EVES translation proved a failure).

⟨*Globals*⟩+≡
```
int eves2 = 0;
```

⟨*Traversals*⟩+≡
```
@traversal @preorder eves2
```

The type of the global state for the translation:

⟨*Include files*⟩+≡
```
#include "eves2_state.h"
```

In **eves2**, rather than do lots of work generating axiom names that are both unique and mnemonic we'll just distinguish them with an integer suffix. Calls to the `my_eves2_state.suffix` method will successively return the strings `"__1"`, `"__2"`, etc.

⟨*Globals*⟩+≡
```
eves2_state my_eves2_state;
```

The command line argument choosing **eves2** is **E**:

⟨*Switches for traversals*⟩+≡
```
case 'E' : eves2 = 1; break;
```

Basic classes needed to define the attributes and the traversal actions define s-expressions and manipulations on them:

⟨*Include files*⟩+≡
```
#include "obj.h"
#include "list.h"
#include "str.h"
#include "s_exp.h"
#include "s_exp_utilities.h"
#include "eves_state.h"
#include "misc_utilities.h"
#include "state_value.h"
```

8.3 Attribute declarations

Currently, there are no special eves attributes.

8.4 Auxiliary functions

The operation `eves_skeleton2` generates skeletons for noweb files that will be used to accumulate an eves theory. It will be called if the `evess` flag is set.

⟨*Function headers*⟩≡

```
void eves2_skeleton();
```

8.4.1 Printing axioms

`print_eves_axiom` prints an EVES rule which will be put into the noweb module named `module_name`; the name of the rule is `name`, its variable list is `vbls`, and its body is `formula`.

⟨*Function headers*⟩+≡

```
void print_eves_axiom(char *kind, char *module_name, Str *name,
                      S_exp_list *vbls,
                      S_exp *formula, FILE *f);
```

⟨*C Macros*⟩+≡

```
#define print_axiom(module_name, name, vbls, formula,f) \
        print_eves_axiom("axiom", module_name, name, vbls, formula, f)

#define print_frule(module_name, name, vbls, formula,f) \
        print_eves_axiom("frule", module_name, name, vbls, formula, f)

#define print_grule(module_name, name, vbls, formula,f) \
        print_eves_axiom("grule", module_name, name, vbls, formula, f)

#define print_rule(module_name, name, vbls, formula,f) \
        print_eves_axiom("rule", module_name, name, vbls, formula, f)
```

We need an analogous operation for a list of formulas. The limitations of `print_eves_family`: all formulas have to be of the same `kind` (axiom, frule, grule, etc.), must be inserted into the same `module_name`, and all must have the same collection of top-level variables (the value of `vbls`). The names assigned to

these axioms will be things like "name_stub__36", "name_stub__37", etc. You'll have no control over which integer; but all will be distinct.

⟨*Function headers*⟩+≡

```
void print_eves_family(char *kind, char *module_name, Str *name_stub,
                       S_exp_list *vbls,
                       S_exp_list *fml_list, FILE *f);
```

⟨*C Macros*⟩+≡

```
#define print_axiom_family(module_name, name_stub, vbls, fml_list,f) \
        print_eves_family("axiom",module_name, name_stub, vbls, fml_list,f)

#define print_frule_family(module_name, name_stub, vbls, fml_list,f) \
        print_eves_family("frule",module_name, name_stub, vbls, fml_list,f)

#define print_grule_family(module_name, name_stub, vbls, fml_list,f) \
        print_eves_family("grule",module_name, name_stub, vbls, fml_list,f)

#define print_rule_family(module_name, name_stub, vbls, fml_list,f) \
        print_eves_family("rule",module_name, name_stub, vbls, fml_list,f)
```

And an analogous operation for printing formula tables. Right now I don't know how rich the type of formula tables will ultimately become, so I'll just define an operation that prints the tautology check for formula tables. The tautology check for event Foo will be printed as an Eves axiom named something like "Foo-events-not-ignored__17".

A first step toward generalizing this is eves_table_check, which takes a Str (a key in some way identifying the source of the formulas) and an S_exp_list. It prints out the check that the sequence of formulas is exclusive and exhaustive.

⟨*Include files*⟩+≡

```
#include "formula_table.h"
```

⟨*Function headers*⟩+≡

```
void eves_taut_check(Str *event_name, S_exp *fml);
void eves_table_check(Str *key, S_exp_list *fml_list);
```

All the names for the EVES formulas are generated in the same way. The expected use of the macro EvesCharName is to replace x with a string literal. The expected use of the macro EvesStrName is to replace x with a Str.

⟨*C Macros*⟩+≡
```
#define EvesStrName(x) \
        x->append(new Str(my_eves2_state.suffix()))
#define EvesCharName(x) EvesStrName((new Str(x)))
```

8.4.2 Printing function declarations

`print_eves_function_stub` prints an EVES function-stub in the named module, with the given name and list of variables.

⟨*Function headers*⟩+≡
```
void print_eves_function_stub(char *module_name, Str *name,
                              S_exp_list *vbls, FILE *f);
```

If `name` points to `"Foo"`, then this function generates the eves declaration `(function-stub Foo ())`.

⟨*Function headers*⟩+≡
```
void declare_in_state(Str *name);
```

8.4.3 Printing tabular definitions

⟨*Include files*⟩+≡
```
#include "function_definition.h"
#include "function_definition_utilities.h"
```

Given the `CaseList` parameter `(c1,v1),(c2,v2),(c3,v3)`, this prints the following family of frules:

```
c1 -> name(vbls) = v1
c2 -> name(vbls) = v2
c3 -> name(vbls) = v3
```

Each rule is given a name of the form `tabular-def-of-name__37`.

⟨*Function headers*⟩+≡
```
void print_tabular_definition(char *module_name, Str *name,
                              S_exp_list *vbls, CaseList *c_list,FILE *f);
```

8.4.4 Loading external EVES theories

To avoid yet another layer of m4 macros on top of this one, we adopt the following clumsy way to define the theories to be loaded by the theory we're building. To load the theory `real` we say

⟨*eves load theories*⟩≡
```
printf("(load real)\n");
```

Another piece of the preamble, the location of the EVES library, is hardwired by the following:

⟨*eves-library*⟩≡
```
printf("(set-library \"/home/projects2/FMI/EVES/translator/eves_rsml_library/\")\n");
```

8.4.5 Commonly used C macros

⟨*C Macros*⟩+≡
```
#define EVES_REAL new Application(new Str("real"))
#define EVES_BOOL new Application(new Str("bool"))
```

8.4.6 Auxiliary operations for the eves2 traversal

The alternative or rules are illustrated as follows. First, a 2-child or-state. Given `parent` is the OR of `child1` and `child2` we want frules as follows:

```
(not child1) -> parent = child2
(not child2) -> parent = child1
```

These would be named, respectively, `child1->parent=child2` and `child2->parent=child1`.

In general, we do things recursively. For example, suppose `parent` is an OR of `child1`, `child2`, and `child3`. We then declare a function stub `parent~1` and add the frules

```
(not child1) -> parent = parent~1
(not parent~1) -> parent = child1

(not child2) -> parent~1 = child3
(not child3) -> parent~1 = child2
```

Etc.

First, we implement an auxiliary operation `two_child_or_rule` that simply generates the two frules of the two-child case; and then a recursive operation `alternative_or_rules` that does the general case. The extra parameters `stub` and `counter` are used to generate names for the auxiliary function stubs (`stub~1`, `stub~2`, ...) that are declared along the way. Logically, the precise choice of `stub` is irrelevant (except for avoiding name clashes)—the intended use is that the top-level call to `alternative_or_rules` uses the `Str` value of `parent` as its `stub`.

The output goes into module "alternative-or frules".

⟨*Function headers*⟩+≡
```
void two_child_or_rule(S_exp *parent, S_exp *child1, S_exp *child2);

void alternative_or_rules(S_exp *parent, S_exp_list *chilluns,
        Str *stub, int counter);
```

8.5 Attribute definitions

Currently, all attributes are defined in `attribute-definitions.nw`.

8.6 Traversal actions

Note: I'm no longer maintaining the `eves` traversal—and everything associated with it should ultimately be deleted.

8.6.1 For `system` productions

⟨*system:SYSTEM SYSTEM_NAME ':' component_list END SYSTEM SECTION_SEPARATOR* ⟩+≡
```
@eves2 if (eves2) { eves2_skeleton();
                  }
```

8.6.2 For the `component` productions

⟨*component_list:*⟩+≡

⟨*component_list:component_list component_def* ⟩+≡

⟨*component_def:COMPONENT COMPONENT_NAME ':' END COMPONENT* ⟩+≡

NOTE: I'm no longer bothering with the `eves` traversal.

To the module for defining the state hierarchy we add the axiom saying that the top-most state of a component (regarded as a Boolean) is always true.

We generate the tautology check for state events and the table check for conditional states.

⟨*component_def:long_case* ⟩+≡

```
@eves2 if (eves2) {
        print_rule("eves-rules defining state hierarchy",
                   EvesCharName("top-level-state"),
                   new S_exp_list,
                   @component_def.in_top_state@,
                   stdout);
        @transition_or_transition_bus_def_list.computed_etests@->
                do_it(eves_taut_check);
        @transition_or_transition_bus_def_list.computed_ctests@->
                do_it(eves_table_check);
}
```

⟨*component_location:COMPONENT_NAME* ⟩+≡

⟨*component_location:EXTERNAL* ⟩+≡

8.6.3 For the state_def productions

For the case of an OR-state both traversals do the following things:

- Declare the state name as a new constant.
- Generate the child-parent and the incompatible-or-sibling frules.

The **eves2** traversal generates the alternative-or-rules.

⟨*state_def:OR_STATE_TOKEN STATE_NAME DEFAULT STATE_NAME ':' state_charts_list END OR_STATE_TOKEN* ⟩+≡

```
@eves2 if (eves2) {
        declare_in_state(@STATE_NAME.0.str@);

        print_frule_family("child-parent frules",
                new Str("child-implies-parent"),
                new S_exp_list,
                @state_def.children_imply_parent@,
                stdout);

        print_frule_family("incompatible-or-sib frules",
                new Str("incompatible-or-sibs"),
                new S_exp_list,
                @state_def.incompatible_siblings@,
                stdout);
```

```
alternative_or_rules(@state_def.parent_as_S_exp@,
        @state_def.children_as_S_exps@,
        @STATE_NAME.str@,
        1);
}
```

In the case of an AND-state we traversals:

- Declare the state name as a new constant.
- Generate the child-parent and the parent-implies-children rules.

⟨*state_def:AND_STATE_TOKEN STATE_NAME ':' state_charts_list END AND_STATE_TOKEN* ⟩+≡

```
@eves2 if (eves2) {
        declare_in_state(@STATE_NAME.str@);

        print_frule_family("child-parent frules",
                new Str("child-implies-parent"),
                new S_exp_list,
                @state_def.children_imply_parent@,
                stdout);

        print_frule("parent-implies-children frules",
                EvesCharName("parent-implies-children"),
                new S_exp_list,
                @state_def.parent_implies_children@,
                stdout);

        }
```

OR-arrays of states are not supported.

⟨*state_def:OR_ARRAY STATE_NAME '[' INTEGER ']' ':' state_charts_list END OR_ARRAY* ⟩+≡

AND-arrays of states are not supported.

⟨*state_def:AND_ARRAY STATE_NAME '[' INTEGER ']' ':' state_charts_list END AND_ARRAY* ⟩+≡

We declare the names of atomic states as constants.

⟨*state_def:ATOMIC STATE_NAME ':'* ⟩+≡

```
@eves2 if (eves2) {
        declare_in_state(@STATE_NAME.0.str@);
        }
```

Conditional states do not figure in the hierarchy, as they're not real states – you're never actually in one, so we declare nothing and generate nothing.

⟨*state_def:CONDITIONAL STATE_NAME ':'* ⟩+≡

State charts lists are not supported.

⟨*state_charts_list:state_def* ⟩+≡

⟨*state_charts_list:state_charts_list state_def* ⟩+≡

8.6.4 For the constants productions

Declare the RSML constant as an EVES constant and add its defining equation as a rewrite rule.

Note 1: Once we set of a suitable real-number theory in EVES, we will presumably want to add the assertion that the constant really does belong to the type real.

Note 2: For the puporses of automatic checking we will presumably want to rewrite constants to their literal values—but when checking fails, this may not be the most helpful thing.

⟨*constant_def:CONSTANT CONSTANT_NAME ':' VALUE ':' REAL END CONSTANT* ⟩+≡

```
@eves2 if (eves2) {
        print_eves_function_stub(
                "rsml-constant decls",
                @CONSTANT_NAME.str@,
                new S_exp_list,
                stdout);


        print_rule(
                "eves-rules for rsml constants",
```

```
                EvesStrName(
                        (new Str("def-constant-"))->
                        append(@CONSTANT_NAME.str@)),
                new S_exp_list,
                @CONSTANT_NAME.as_S_exp@->
                    Equals(new RealLiteral(@REAL.val@)),
                stdout);

}
```

⟨*constant_def:CONSTANT CONSTANT_NAME ':' VALUE ':' '-' REAL END CONSTANT*⟩+≡

```
@eves2 if (eves2) {

        print_eves_function_stub(
                "rsml-constant decls",
                @CONSTANT_NAME.str@,
                new S_exp_list,
                stdout);

        print_rule(
                "eves-rules for rsml constants",
                EvesStrName(
                        (new Str("def-constant-"))->
                        append(@CONSTANT_NAME.str@)),
                new S_exp_list,
                @CONSTANT_NAME.as_S_exp@->
                    Equals(new Application(
                        new Str("-"),
                        new S_exp_list(new RealLiteral(@REAL.val@))
                        )
                    ),
                stdout);

}
```

⟨*constant_ref:CONSTANT_NAME*⟩+≡

⟨*constant_def_list:*⟩+≡

8.6.5 For the variables productions

For now, the only actions we need to take are to declare in variables and out variables as unspecified constants. (As elsewhere noted, these variables are logical constants properly modeled as maps from time to real values; but for now I'll just make them unspecified real-valued constants.)

⟨*in_variable_def:IN_VARIABLE IN_VAR_NAME ':' TYPE ':' NUMERIC expected_min expected_max min_gran max_gran END IN_VARIABLE* ⟩+≡

```
        @eves2 if (eves2) {
                print_eves_function_stub(
                        "eves-function-declarations",
                        @IN_VAR_NAME.str@,
                        new S_exp_list,
                        stdout);

                print_grule(
                        "eves-axioms for model",
                        EvesStrName(
                                (new Str("type-of-"))->
                                append(@IN_VAR_NAME.str@)),
                        new S_exp_list,
                        (new Application(new Str("type-of"),
                                Singleton(@IN_VAR_NAME.as_S_exp@)))->
                                        Equals(EVES_REAL),
                        stdout);
        }
```

⟨*out_variable_def:OUT_VARIABLE OUT_VAR_NAME ':' TYPE ':' NUMERIC expected_min expected_max min_gran max_gran ASSIGNMENT ':' expression TRIGGER ':' event_ref END OUT_VARIABLE* ⟩+≡

```
        @eves2 if (eves2) {

                print_eves_function_stub(
                        "eves-function-declarations",
                        @OUT_VAR_NAME.str@,
                        new S_exp_list,
```

```
                stdout);

        print_grule(
                "eves-axioms for model",
                EvesStrName(
                        (new Str("type-of-"))->
                        append(@OUT_VAR_NAME.str@)),
                new S_exp_list,
                (new Application(new Str("type-of"),
                        Singleton(@OUT_VAR_NAME.as_S_exp@)))->
                                Equals(EVES_REAL),
                stdout);


}
```

⟨*expected_min:*⟩+≡

⟨*expected_min:EXPECTED_MIN ':' REAL* ⟩+≡

⟨*expected_max:*⟩+≡

⟨*expected_max:EXPECTED_MAX ':' REAL* ⟩+≡

⟨*min_gran:*⟩+≡

⟨*min_gran:MIN_GRAN ':' REAL* ⟩+≡

⟨*max_gran:*⟩+≡

⟨*max_gran:MAX_GRAN ':' REAL* ⟩+≡

⟨*in_variable_ref:IN_VAR_NAME* ⟩+≡

⟨*out_variable_ref:OUT_VAR_NAME* ⟩+≡

⟨*in_variable_ass:IN_VAR_NAME* ⟩+≡

⟨*out_variable_ass:OUT_VAR_NAME* ⟩+≡

⟨*in_variable_def_list:*⟩+≡

⟨*in_variable_def_list:in_variable_def_list in_variable_def*
⟩+≡

⟨*out_variable_def_list:*⟩+≡

⟨*out_variable_def_list:out_variable_def_list out_variable_def*
⟩+≡

⟨*in_variable_list:*⟩+≡

⟨*in_variable_list:in_variable_ass* ⟩+≡

⟨*in_variable_list:in_variable_list ',' in_variable_ass* ⟩+≡

⟨*variable:out_variable_ref* ⟩+≡

⟨*variable:in_variable_ref* ⟩+≡

⟨*variable_ref:PREV '(' REAL ')' in_variable_ref* ⟩+≡

⟨*variable_ref:in_variable_ref* ⟩+≡

8.6.6 For the events productions

This code is for diagnostics and testing.

⟨*Include files*⟩+≡

```
#include "event_value_utilities.h"
```

⟨*event_def_list:*⟩+≡

```
        @eves2 if (eves2) {
/*
                printf("event_def_list with empty child\n");
                printf("Inherited\n");
                print_EventValueList(@event_def_list.inherited_events@);
                printf("Computed\n");
                print_EventValueList(@event_def_list.computed_events@);
*/
        }
```

⟨*event_def_list:event_def_list event_def*⟩+≡

```
        @eves2 if (eves2) {
        }
```

⟨*event_def:EVENT EVENT_NAME STATE ';'*⟩+≡

```
        @eves2 if (eves2) {
        }
```

⟨*event_def:EVENT EVENT_NAME VARIABLE ';'*⟩+≡

```
        @eves2 if (eves2) {
        }
```

⟨*event_def:EVENT EVENT_NAME INTERFACE ';'*⟩+≡

```
        @eves2 if (eves2) {
        }
```

⟨*event_ref:EVENT_NAME*⟩+≡

```
        @eves2 if (eves2) {
        }
```

⟨*event_list:*⟩+≡

```
@eves2 if (eves2) {
}
```

⟨*event_list:event_ref*⟩+≡

```
@eves2 if (eves2) {
}
```

⟨*event_list:event_list ',' event_ref*⟩+≡

```
@eves2 if (eves2) {
}
```

8.6.7 For the `interfaces` productions

8.6.8 For the `macros` productions

8.6.9 For the `functions` productions

⟨*function_def_list:*⟩+≡

```
@eves2 if (eves2) {
}
```

⟨*function_def_list:function_def_list function_def*⟩+≡

```
@eves2 if (eves2) {
}
```

⟨*function_def:FUNCTION FUNCTION_NAME '(' local_var_list ')' ':' RETURN ':' CASE case_list END CASE END FUNCTION*⟩+≡

```
@eves2 if (eves2) {
        print_eves_function_stub(
                "eves-rsml function decls",
                @FUNCTION_NAME.str@,
                @local_var_list.computed_val@,
                stdout);

        print_tabular_definition(
```

```
                "eves-rsml function definitions",
                @FUNCTION_NAME.str@,
                @local_var_list.computed_val@,
                @case_list.computed_val@,
                stdout);

        print_axiom(
                "rsml-functions well-defined",
                EvesStrName(
                        @FUNCTION_NAME.str@->
                        append(new Str("-cases-exclusive"))),
                @local_var_list.computed_val@,
                mutual_exclusion(
                        condition_list(@case_list.computed_val@)
                        )->And(),
                stdout);

        print_axiom(
                "rsml-functions well-defined",
                EvesStrName(
                        @FUNCTION_NAME.str@->
                        append(new Str("-cases-exhaustive"))),
                @local_var_list.computed_val@,
                condition_list(@case_list.computed_val@)->Or(),
                stdout);


}
```

⟨*case_list:case* ⟩+≡

```
@eves2 if (eves2) {
}
```

⟨*case_list:case_list case* ⟩+≡

```
@eves2 if (eves2) {
}
```

⟨*case:expression IF pred_ref ';'* ⟩+≡

```
@eves2 if (eves2) {
}
```

8.6.10 For the `transition busses` productions

8.6.11 For the `transitions` productions

8.6.12 For the `predicate` productions

8.6.13 For the `conditions` productions

8.6.14 For the `expressions` productions

8.7 Bodies of auxiliary functions

Because this text is being run through notangle any module brackets which are to appear in the notangled code must be escaped with @-signs.

8.7.1 eves2_skeleton

The skeleton is generated in a somewhat roundabout way: We use notangle to store the contents of module "eves skeleton" in a file called **`_eves_skeleton`**. Then **`eves2_skeleton`** uses the function **`grab`** to read this file and return it as part of the act of generating the skeleton. The reason for this deviousness is to keep the documentation for the skeleton under control of noweb. The alternative would be to fill the body of **`eves2_skeleton`** with an eye-glazing accumulation of **`printf`** or **`fprintf`** statements.

The top-level outline:

⟨*eves skeleton*⟩≡

```
<<*>>=
<<eves-library>>
<<eves-load theories>>
<<eves-function-declarations>>
<<eves-axioms for model>>
<<eves-candidate theorems>>
```

EVES function declarations

⟨*eves skeleton*⟩+≡

```
<<eves-function-declarations>>=
<<in-state decls>>
<<rsml-constant decls>>
<<eves-rsml function decls>>
```

Axioms

⟨*eves skeleton*⟩+≡
```
<<eves-axioms for model>>=
<<eves-rules defining state hierarchy>>
<<eves-rules for rsml constants>>
<<eves-rsml function definitions>>
```

Axioms: state hierarchy

⟨*eves skeleton*⟩+≡
```
<<eves-rules defining state hierarchy>>=
<<child-parent frules>>
<<parent-implies-children frules>>
<<incompatible-or-sib frules>>
<<alternative-or frules>>
```

Candidate theorems: events not ignored, functions well-defined, conditional states well-defined

⟨*eves skeleton*⟩+≡
```
<<eves-candidate theorems>>=
<<events-not-ignored>>
<<conditional-states-well-defined>>
<<rsml-functions well-defined>>
```

Boilerplate:

⟨*eves skeleton*⟩+≡
```
<<in-state decls>>=
<<rsml-constant decls>>=
<<eves-rsml function decls>>=
<<eves-rules for rsml constants>>=
<<eves-rsml function definitions>>=
<<child-parent frules>>=
<<parent-implies-children frules>>=
<<incompatible-or-sib frules>>=
<<alternative-or frules>>=
<<events-not-ignored>>=
<<conditional-states-well-defined>>=
<<rsml-functions well-defined>>=
```

⟨*Include files*⟩+≡
```
#include "skeleton_utilities.h"
```

⟨*Bodies*⟩+≡
```
void eves2_skeleton(){

  grab("_eves_skeleton",stdout);

// Setting library

  printf("\n<<eves-library>>=\n");
```
⟨*eves-library*⟩
```
// Loading theories

  printf("\n<<eves-load theories>>=\n");
```
⟨*eves load theories*⟩
```
};
```

8.7.2 print_eves_axiom

⟨*Bodies*⟩+≡
```
void print_eves_axiom(char *kind, char *module_name, Str *name,
                      S_exp_list *vbls, S_exp *formula, FILE *f){

  fprintf(f,"\n<<%s>>=\n", module_name);
  fprintf(f,"(%s %s (", kind, name->str);
  if (! vbls->isEmpty()) {vbls->eves_print(f);}
  fprintf(f,")\n\t");
  formula->eves_print(f);
  fprintf(f,")\n");
};
```

8.7.3 print_eves_family

⟨*Bodies*⟩+≡
```
void print_eves_family(char *kind, char *module_name, Str *name_stub,
                       S_exp_list *vbls,
                       S_exp_list *fml_list, FILE *f){

  while (! fml_list->isEmpty() && fml_list != NULL ){
    print_eves_axiom(kind, module_name,
                     EvesStrName(name_stub),
                     vbls, fml_list->head(), f);
```

```
    fml_list = fml_list->tail();
  }
};
```

8.7.4 eves_taut_check

⟨*Bodies*⟩+≡

```
void eves_taut_check(Str* event_name, S_exp* fml){

  Str *name_stub = new Str("events-not-ignored-");

  print_axiom("events-not-ignored",
              EvesStrName((name_stub->append(event_name))),
              new S_exp_list, fml, stdout);
};
```

8.7.5 eves_table_check

⟨*Include files*⟩+≡

```
#include "s_exp_utilities.h"
```

⟨*Bodies*⟩+≡

```
void eves_table_check(Str *key, S_exp_list *fml_list){

  Str *exhaustive = new Str("cond-state-exhaustive-");
  Str *exclusive = new Str("cond-state-exclusive-");

  print_axiom("conditional-states-well-defined",
              EvesStrName((exhaustive->append(key))),
               new S_exp_list,
               (new S_exp_list)->Or(),
/*  fml_list->Or(), */
               stdout);

  print_axiom("conditional-states-well-defined",
              EvesStrName((exclusive->append(key))),
               new S_exp_list, mutual_exclusion(fml_list)->And(), stdout);

};
```

8.7.6 print_eves_function_stub

⟨*Bodies*⟩+≡
```
void print_eves_function_stub(char *module_name, Str *name,
                      S_exp_list *vbls, FILE *f){

  fprintf(f,"\n<<%s>>=\n", module_name);
  fprintf(f,"(function-stub %s (",name->str);
  vbls->eves_print(f);
  fprintf(f,"))\n", name->str);
};
```

8.7.7 declare_in_state

⟨*Bodies*⟩+≡
```
void declare_in_state(Str *name){
  print_eves_function_stub("in-state decls", name,
                          new S_exp_list, stdout);
};
```

8.7.8 print_tabular_definition

⟨*Bodies*⟩+≡
```
void print_tabular_definition(char *module_name, Str *name,
                      S_exp_list *vbls, CaseList *c_list,FILE *f){

  Str *name_stub = (new Str("tabular-def-of-"))->append(name);

  print_frule_family(module_name,
                     name_stub,
                     vbls,
                     as_S_exp_list(c_list,
                                   new Application(name,vbls)),
                     f);
};
```

8.7.9 declare_disjoin

⟨*Bodies*⟩+≡

```
void declare_disjoin(int arity){

  S_exp_list *arguments = new S_exp_list();
  Str *disjoin = new Str(decorate("disjoin",arity));
  Str *argn;

  for (int i = arity; i >=1; i--){
    argn = new Str(decorate("arg",i));
    arguments = new S_exp_list(new Variable(argn),arguments);
  }

  print_eves_function_stub("disjoin operators", disjoin,
        arguments, stdout);
};
```

8.7.10 two_child_or_rule

Recall that EQ_SIGN has been defined as a pointer to Str("=")

⟨*Bodies*⟩+≡

```
void two_child_or_rule(S_exp *parent, S_exp *child1, S_exp *child2){

   Str *arrow = new Str("->");
   Str *not = new Str("not-");
   Str *first_name = not->append(child1->PConnName())->
                     append(arrow)->
                     append(parent->PConnName())->append(EQ_SIGN)->
                     append(child2->PConnName());

   Str *second_name = not->append(child2->PConnName())->
                      append(arrow)->
                      append(parent->PConnName())->append(EQ_SIGN)->
                      append(child1->PConnName());

   print_frule("alternative-or frules",
               first_name,
               new S_exp_list,
               child1->Not()->Implies(parent->Equals(child2)),
               stdout);

   print_frule("alternative-or frules",
               second_name,
```

```
                new S_exp_list,
                child2->Not()->Implies(parent->Equals(child1)),
                stdout);
};
```

8.7.11 alternative_or_rules

⟨*Bodies*⟩+≡

```
void alternative_or_rules(S_exp *parent, S_exp_list *chilluns,
                          Str *stub, int counter){

  int num_chilluns = chilluns->length();

  S_exp *child1 = chilluns->head();
  Str *child1_name = child1->PConnName();
  Str *parent_name = parent->PConnName();

  if (num_chilluns == 1)
     {print_rule("alternative-or frules",
                 parent_name->append(EQ_SIGN)->append(child1_name),
                 new S_exp_list,
                 parent->Equals(child1),
                 stdout);
      return; }

  S_exp_list *tail = chilluns->tail();
  S_exp *child2 = tail->head();
  Str *child2_name = child2->PConnName();

  if (num_chilluns == 2)
     {two_child_or_rule(parent,child1,child2);}
  else
     {Str *next_fn_name =
        new Str(decorate(stub->append("~")->str,counter));
      S_exp *next_fn = new Application(next_fn_name);

      print_eves_function_stub("alternative-or frules",
                               next_fn_name,
                               new S_exp_list,
                               stdout);
      two_child_or_rule(parent,child1,next_fn);
      alternative_or_rules(next_fn,tail,stub,counter+1);}
};
```

Chapter 9

PVS

This chapter contains a partly filled in template for integrating the PVS prover.

9.1 Currently implemented

In the p translation we currently generate the following PVS theory:

- Unconditional states are declared as boolean constants, and axioms define the hierarchy relation (in particular, the "in state" and "in one of" predicates).

The following theory is generated for EVES in the E traversal, and should be immediate PVS:

- RSML constants are declared and axioms define their initial values.
- RSML in variables are declared as unspecified real-valued constants.

  Note that RSML variables should really be modeled as real-valued functions of time.
- The definitions of RSML functions are entered as axioms.

Out variables are ignored. Timeout events are ignored, and predicates referring to the timed properties of variables are dummied to the formula "false." (It might be more informative to dummy them as distinct unspecified constants with suitable mnemonic names.)

We generate candidate theorems corresponding to the following tests:

- The conditions for exiting from each conditional state are complete and consistent.
- For each event, the conditions for transitions triggered by the event are complete.

The E traversal also generates the assertion saying that the table defining each rsml function is complete and consistent.

9.2 Preliminaries

The flag coding the name of the PVS traversal is `pvs`.

⟨*Globals*⟩+≡
```
int pvs = 0;
```

`pvs` is declared as a traversal by an Ox traversal declaration:

⟨*Traversals*⟩+≡
```
@traversal @preorder pvs
```

The type of the global state for the translation:

⟨*Include files*⟩+≡
```
#include "pvs_state.h"
```

The variable representing the state is `pvs_state`:

⟨*Globals*⟩+≡
```
pvs_state my_pvs_state;
```

The command line argument choosing this traversal is *p*

⟨*Switches for traversals*⟩+≡
```
case 'p' : pvs = 1; break;
```

Basic classes needed to define the attributes and the traversal actions:

⟨*Include files*⟩+≡
```
#include "pvs_types.h"
```

9.3 Attribute declarations

All the "basic" attributes are declared in **`attribute-declarations.nw`**. This section contains the declarations of applications specific to a particular translation. The attributes defined for the grammar symbol "BAR" go in the module "BAR attributes". The list of all possible attribute modules can be found in the file `attribute_modules.nw`.

9.4 Auxiliary functions

9.4.1 Printing constant declarations

PVS constant declarations can, in general, be pretty complicated, and there are many equivalent forms for providing the same definition. For example,

```
g(x:int):int = x+1
```

is equivalent to

```
g : [int -> int]
g: AXIOM g = (LAMBDA (x:int): x+1)
```

and also equivalent to

```
x: VAR int
g(x):int = x+1
```

For present purposes I'll "normalize" all definitions into the second form, so that the declaration itself, which occupies the first line, simply associates a name with a type expression (and, for now, the type of type expressions will be dummied out to `Str`. Thus we have

⟨*Function headers*⟩+≡
```
void print_pvs_const_decl(char *module_name, Str *name,
                          PVS_type_exp *type_exp, FILE *f);
```

This puts the declaration `name : type_exp` into the noweb module "module_name."

Following the pattern used for EVES, we specialize the preceding function to

⟨*Function headers*⟩+≡
```
void pvs_declare_in_state(Str *name);
```

which declares `*name` as a boolean constant and puts its declaration into the noweb module "pvs-in-state decls".

9.4.2 Printing formulas

`print_pvs_formula` prints a PVS formula which will be put into the noweb module named `module_name`; the name of the rule is `name` and its body is `formula`.

⟨*Function headers*⟩+≡
```
void print_pvs_formula(char *kind, char *module_name, Str *name,
                       S_exp *formula, FILE *f);
```

⟨*C Macros*⟩+≡
```
#define print_pvs_axiom(module_name, name,formula,f) \
        print_pvs_formula("axiom",module_name, name,formula,f)

#define print_pvs_theorem(module_name, name,formula,f) \
        print_pvs_formula("theorem",module_name, name,formula,f)
```

As in the EVES case, `print_pvs_family` is the analogous operation for list of formulas. Limitations (as in the EVES case): all formulas of the same `kind` (theorem, axiom, etc.) and all inserted into the same `module_name`. The names will be things like `"name_stub__36"`, `"name_stub__37"`, etc. You'll have no control over which integer; but all will be distinct.

⟨*Function headers*⟩+≡
```
void print_pvs_family(char *kind, char *module_name, Str *name_stub,
                      S_exp_list *fml_list, FILE *f);
```

⟨*C Macros*⟩+≡
```
#define print_pvs_axiom_family(module_name, name_stub, fml_list,f) \
        print_pvs_family("axiom",module_name, name_stub, fml_list,f)

#define print_pvs_theorem_family(module_name, name_stub, fml_list,f) \
        print_pvs_family("theorem",module_name, name_stub, fml_list,f)
```

And an analogous operation for printing formula tables. Right now I don't know how rich the type of formula tables will ultimately become, so I'll just define an operation that prints the tautology check for formula tables. The tautology check for event `Foo` will be printed as a PVS theorem named something like `"Foo_tautology_check__17"`.

⟨*Include files*⟩+≡
```
#include "formula_table.h"
```

⟨*Function headers*⟩+≡
```
void pvs_taut_check(Str* event_name, S_exp* fml);
```

All the names for the PVS formulas are generated in the same way. The expected use of the macro `EvesCharName` is to replace `x` with a string literal. The expected use of the macro `EvesStrName` is to replace `x` with a `Str`.

⟨*C Macros*⟩+≡
```
#define PvsStrName(x) \
        x->append(new Str(my_pvs_state.suffix()))
#define PvsCharName(x) PvsStrName((new Str(x)))
```

9.4.3 PVS skeleton

The skeleton of a noweb file whose tangling will accumulate the translation:

⟨*Function headers*⟩+≡
```
void pvs_skeleton();
```

9.5 Attribute definitions

Currently, all attributes are defined in `attribute-definitions.nw`.

9.6 Traversal actions

9.6.1 For the production system

The system name is the name of the top-level PVS theory.

⟨*system:SYSTEM SYSTEM_NAME ':' component_list END SYSTEM SECTION_SEPARATOR* ⟩+≡
```
@pvs if (pvs) {

        pvs_skeleton();

        printf("\n<<system name>>=\n");
        @SYSTEM_NAME.str@->print(stdout);
        }
```

9.6.2 For the component productions

⟨*component_list:*⟩+≡

⟨*component_list:component_list component_def* ⟩+≡

To the module for defining the state hierarchy we add the axiom saying that the top-most state of a component (regarded as a Boolean) is always true.

We generate the tautology check for state events.

STILL TO D0: Generate the table check for conditional states.

⟨*component_def:COMPONENT COMPONENT_NAME ':' END COMPONENT* ⟩+≡

⟨*component_def:long_case* ⟩+≡

```
@pvs if (pvs) {
        print_pvs_axiom(
                "pvs-rules-defining-state-hierarchy",
                PvsCharName("in_top_state"),
                @component_def.in_top_state@,
                stdout);

        @transition_or_transition_bus_def_list.computed_etests@->
                do_it(pvs_taut_check);
}
```

⟨*component_location:COMPONENT_NAME* ⟩+≡

⟨*component_location:EXTERNAL* ⟩+≡

9.6.3 For the states productions

⟨*state_def:OR_STATE_TOKEN STATE_NAME DEFAULT STATE_NAME ':' state_charts_list END OR_STATE_TOKEN* ⟩+≡

```
@pvs if (pvs) {
        pvs_declare_in_state(@STATE_NAME.0.str@);

        print_pvs_axiom_family("pvs-child-parent rules",
                new Str("child_implies_parent"),
                @state_def.children_imply_parent@,
                stdout);

        print_pvs_axiom("pvs-parent-implies-children rules",
                PvsCharName("or_parent_implies_children"),
                @state_def.parent_implies_children@,
```

```
        stdout);

print_pvs_axiom_family("pvs-incompatible-or-sib rules",
        new Str("incompatible_or_sibs"),
        @state_def.incompatible_siblings@,
        stdout);

printf("\n");

}
```

In the case of an AND-state we do the following things:

- Declare the state name as a new constant.
- Generate the child-parent and the parent-implies-children rules.

⟨*state_def:AND_STATE_TOKEN STATE_NAME ':' state_charts_list END AND_STATE_TOKEN* ⟩+≡

```
@pvs if (pvs) {

        pvs_declare_in_state(@STATE_NAME.0.str@);

        print_pvs_axiom_family("pvs-child-parent rules",
                new Str("child_implies_parent"),
                @state_def.children_imply_parent@,
                stdout);

        print_pvs_axiom("pvs-parent-implies-children rules",
                PvsCharName("and_parent_implies_children"),
                @state_def.parent_implies_children@,
                stdout);

        }
```

OR-arrays of states are not supported.

⟨*state_def:OR_ARRAY STATE_NAME '[' INTEGER ']' ':' state_charts_list END OR_ARRAY* ⟩+≡

AND-arrays of states are not supported.

⟨*state_def:AND_ARRAY STATE_NAME '[' INTEGER ']' ':' state_charts_list END AND_ARRAY*⟩+≡

We declare the names of atomic states as constants.

⟨*state_def:ATOMIC STATE_NAME ':'*⟩+≡

```
        @pvs if (pvs) {
                pvs_declare_in_state(@STATE_NAME.0.str@);
                }
```

Conditional states do not figure in the hierarchy, as they're not real states – you're never actually in one, so we declare nothing and generate nothing.

⟨*state_def:CONDITIONAL STATE_NAME ':'*⟩+≡

State charts lists are not supported.

⟨*state_charts_list:state_def*⟩+≡

⟨*state_charts_list:state_charts_list state_def*⟩+≡

9.6.4 For the `constants` productions

NOTE: constant declarations and definitions are done for EVES.

9.6.5 For the `variables` productions

NOTE: variable declarations are done for EVES.

9.6.6 For the `events` productions

9.6.7 For the `interfaces` productions

9.6.8 For the `macros` productions

9.6.9 For the `functions` productions

NOTE: RSML function declarations and definitions are done for EVES.

9.6.10 For the transition busses productions

9.6.11 For the transitions productions

9.6.12 For the predicate productions

9.6.13 For the conditions productions

9.6.14 For the expressions productions

9.7 Bodies of auxiliary functions

Because this text is being run through notangle the module brackets which are to appear in the printed code must be escaped with @-signs.

9.7.1 pvs_const_decl

⟨*Bodies*⟩+≡

```
void print_pvs_const_decl(char *module_name, Str *name,
                          PVS_type_exp *type_exp, FILE *f){
  fprintf(f,"\n<<%s>>=\n", module_name);
  fprintf(f,"%s : ", name->str);
  type_exp->print(f);
  fprintf(f,"\n");
};
```

9.7.2 pvs_declare_in_state

⟨*Bodies*⟩+≡

```
void pvs_declare_in_state(Str *name){
  print_pvs_const_decl("pvs-in-state decls",name,PVS_bool,stdout);
};
```

9.7.3 print_pvs_formula

⟨*Bodies*⟩+≡

```
void print_pvs_formula(char *kind, char *module_name, Str *name,
                       S_exp *formula, FILE *f){

  fprintf(f,"\n<<%s>>=\n", module_name);
  name->print(f);
  fprintf(f,": %s ", kind);
  formula->pvs_print(f);
```

```
  fprintf(f,"\n");
};
```

9.7.4 print_pvs_family

⟨*Bodies*⟩+≡

```
void print_pvs_family(char *kind, char *module_name, Str *name_stub,
                       S_exp_list *fml_list, FILE *f){
  while (! fml_list->isEmpty() && fml_list != NULL ){
    print_pvs_axiom(module_name,
                    PvsStrName(name_stub),
                     fml_list->head(), f);
    fml_list = fml_list->tail();
  }

};
```

9.7.5 pvs_taut_check

⟨*Bodies*⟩+≡

```
void pvs_taut_check(Str* event_name, S_exp* fml){

  Str *name_stub = new Str("tautology_check_");

  print_pvs_theorem("pvs-events-not-ignored",
              PvsStrName((name_stub->append(event_name))),
              fml, stdout);

};
```

9.7.6 pvs_skeleton

Notangling will put module "pvs skeleton" into the file `_pvs_skeleton`.

Top-level structure:

⟨*pvs skeleton*⟩≡

```

<<*>>=
<<system name>> : THEORY
<<pvs-exportings>>
BEGIN
```

```
<<pvs-assumings>>
<<pvs-theory part>>
END <<system name>>
```

The only part currently populated is the theory part: type declarations, constant declarations, def declarations, and formulas

⟨*pvs skeleton*⟩+≡

```
<<pvs-theory part>>=
<<pvs-type-decl>>
<<pvs-const-decl>>
<<pvs-def-decl>>
<<pvs-formula>>
```

Formulas define the state hierarchy and candidate theorems

⟨*pvs skeleton*⟩+≡

```
<<pvs-formula>>=
<<pvs-rules-defining-state-hierarchy>>
<<pvs-candidate-theorems>>
```

Defining the state hieraracby:

⟨*pvs skeleton*⟩+≡

```
<<pvs-rules-defining-state-hierarchy>>=
<<pvs-child-parent rules>>
<<pvs-parent-implies-children rules>>
<<pvs-incompatible-or-sib rules>>
```

Candidate theorems

⟨*pvs skeleton*⟩+≡

```
<<pvs-candidate-theorems>>=
<<pvs-events-not-ignored>>
```

Boilerplate

⟨*pvs skeleton*⟩+≡

```
<<pvs-exportings>>=
<<pvs-assumings>>=
```

```
<<pvs-type-decl>>=
<<pvs-const-decl>>=
<<pvs-in-state decls>>
<<pvs-in-state decls>>=
<<pvs-def-decl>>=
<<pvs-events-not-ignored>>=
```

⟨*Bodies*⟩+≡

```
void pvs_skeleton(){

  grab("_pvs_skeleton",stdout);

};
```

Chapter 10

SPIN

Invoke the SPIN traversal with a command line that looks like this

```
translator -is <file_name> | notangle
```

This code will look for the body of the driver in the file `_driver_body` and will send the SPIN model of `file_name` to standard out. If the name of the component is `Foo`, each line of `_driver_body` must look like

```
In_Channel_Foo ! <exp>, <exp>, ...;
```

Unfortunately, there is currently no way to know the types and meanings of the `<exp>`s without running the translator once and inspecting the definition of the channel type `In_Channel_Foo`. It will be simple enough to provie more felicitous arrangements.

10.1 The strategy

10.2 Preliminaries

The usual preliminaries:
The flag coding the name of the traversal is `spin`

⟨*Globals*⟩+≡
```
int spin = 0;
```

`spin` is declared as a traversal by an Ox traversal declaration:

⟨*Traversals*⟩+≡
```
@traversal @postorder spin
```

Note that this is a postorder traversal. We will note the places in which postorder is essential.

The type of the global state for the translation:

⟨*Include files*⟩+≡
```
#include "spin_state.h"
```

The variable representing the state is `spin_state`:

⟨*Globals*⟩+≡
```
spin_state my_spin_state;
```

The command line argument choosing this traversal is *?*

⟨*Switches for traversals*⟩+≡
```
case 's' : spin = 1; break;
```

IMPORTANT NOTE: as things now stand it is necessary to do traversal `i` before doing traversal `s`.

Basic classes needed to define the attributes and the traversal actions:

⟨*Include files*⟩+≡

10.3 Attribute declarations

All the "basic" attributes are declared in `attribute-declarations.nw`. This section contains the declarations of applications specific to defining the SPIN model.

`in_channel_name` is the name of the channel through which events and values arrive at the component's in interface; and `in_type_name` is the name of the type of values sent on this channel.

⟨*component_def attributes*⟩+≡
```
    Str *in_type_name;
    Str *in_channel_name;
```

10.4 Auxiliary functions

⟨*Function headers*⟩+≡
```
void spin_skeleton();
```

10.5 Attribute definitions

⟨*component_def:COMPONENT COMPONENT_NAME ':' END COMPONENT* ⟩+≡

```
@i @component_def.in_type_name@ =
       @COMPONENT_NAME.str@->prepend("In_Interface_");

@i @component_def.in_channel_name@ =
       @COMPONENT_NAME.str@->prepend("In_Channel_");
```

⟨*component_def:long_case* ⟩+≡

```
@i @component_def.in_type_name@ =
       @COMPONENT_NAME.str@->prepend("In_Interface_");

@i @component_def.in_channel_name@ =
       @COMPONENT_NAME.str@->prepend("In_Channel_");
```

10.6 Traversal actions

⟨*Include files*⟩+≡

```
#include "c_style.h"
```

10.6.1 For the production system

⟨*system:SYSTEM SYSTEM_NAME ':' component_list END SYSTEM SECTION_SEPARATOR* ⟩+≡

```
@spin if (spin) { spin_skeleton(); }
```

10.6.2 For the component productions

Within the `long_case` we generate the name of the type and channel for the interface of the (currently, the unique) component. We also generate the declaration of the process representing the component, and the "init" statement that runs that process. Note: the modules "name of interface type", and "name of interface channel", "name of component," and "name of top level state" are inserted at several places in the text. It's essential, therefore, that those modules are written to only by these `print_spin_code` statements.

⟨*component_def:COMPONENT COMPONENT_NAME ':' END COMPONENT* ⟩+≡

⟨*component_def:long_case* ⟩+≡

```
@spin if (spin) {
        print_spin_code("name of interface type",
                        @component_def.in_type_name@,
                        stdout);

        print_spin_code("name of interface channel",
                        @component_def.in_channel_name@,
                        stdout);

        print_spin_code("name of top level state",
                        @state_def.val@->str(),
                        stdout);

        print_spin_code("name of component",
                        @COMPONENT_NAME.str@,
                        stdout);

        }
```

⟨*component_location:COMPONENT_NAME* ⟩+≡

⟨*component_location:EXTERNAL* ⟩+≡

10.6.3 For the states productions

For each state Foo we generate SPIN to define the integer code for the state, the "leave" macro, and the "enter" macro. In the SPIN code, the macros for children must occur before those for parents—which is automatic if we generate them in a postorder traversal. We also generate the "set parent" definitions, and in this case the macros for parents must occur before those for children, which will automatically result from a preorder traversal.

The integer code definition for state Foo is

```
#define Foo i
```

where i is the unique integer code for Foo. It's obtained by the following macro.

⟨*Ox Macros*⟩+≡

```
@macro SpinStateCode()
```

```
    print_spin_code("define states",
                    define_int_code(@STATE_NAME.0.str@,
                                    @state_def.code@),
                    stdout);
  @end
```

If Foo has a parent, Bar, then the "set parent" definition for Foo is

```
#define set_parent_Foo    state[Bar]=TRUE; set_parent_Bar;
```

If Foo does not have a parent, we generate

```
#define set_parent_Foo    state[Foo]=TRUE;
```

[This actually adds a redundant definition. What we should do, instead, is to have the children inherit an attribute telling them whether their parent is top-level or not. In that case, we could altogether avoid defining set_parent_Foo for top-level Foo.]

⟨*Ox Macros*⟩+≡

```
  @macro SpinSetParent()
    print_set_parent(@STATE_NAME.0.str@,
                     @state_def.ancestor@,
                     stdout);
  @end
```

The "leave" macro does the following: For each non-conditional state A with non-conditional children B, C, D, we generate the SPIN definition:

```
#define leave_A     state[A]=FALSE; leave_B; leave_C; leave_D;
```

which is done by the following macro:

⟨*Ox Macros*⟩+≡

```
  @macro SpinLeavingCode()
    print_leave_definition(
          @STATE_NAME.0.str@,
          UnCondStatesToStrList(@state_charts_list.comp_vals@),
          stdout);
  @end
```

The "enter" definition is slightly more complex. Entering any non-conditional state means, first of all, doing "set parent" and then the following:

- Entering an atomic state sets it to true.

- Entering an AND-state sets it to true and enters all its children.
- Entering an OR-state sets it to true and enters its default state. (I don't think it's necessary also to explicitly leave all its siblings—that should come for free.)

These must be generated by a postorder traversal.

As previously noted conditional states are not supported: "Entering" a conditional state doesn't actually land us in the state—rather, it immediately results in a transition (based on the condition) to another state. Semantic problems include the possibility of an infinite loop among conditional transitions.

⟨*state_def:OR_STATE_TOKEN STATE_NAME DEFAULT STATE_NAME ':' state_charts_list END OR_STATE_TOKEN* ⟩+≡

```
@spin if (spin) {
       SpinStateCode()
       SpinLeavingCode()
       print_enter_or(@STATE_NAME.0.str@,
                      @STATE_NAME.1.str@,
                      stdout);
       }

@spin @revorder(1) if (spin) {
       SpinSetParent()
       }
```

⟨*state_def:AND_STATE_TOKEN STATE_NAME ':' state_charts_list END AND_STATE_TOKEN* ⟩+≡

```
@spin if (spin) {
       SpinStateCode()
       SpinLeavingCode()
       print_enter_and(@STATE_NAME.str@,
                       @state_charts_list.comp_vals@,
                       stdout);
       }

@spin @revorder(1) if (spin) {
       SpinSetParent()
       }
```

⟨*state_def:OR_ARRAY STATE_NAME '[' INTEGER ']' ':' state_charts_list END OR_ARRAY* ⟩+≡

```
@spin if (spin) {
        printf("\n/******OR-arrays not supported*******/\n");
        }
```

⟨*state_def:AND_ARRAY STATE_NAME '[' INTEGER ']' ':' state_charts_list END AND_ARRAY*⟩+≡

```
@spin if (spin) {
        printf("\n/******AND-arrays not supported*******/\n");
        }
```

⟨*state_def:ATOMIC STATE_NAME ':'*⟩+≡

```
@spin if (spin) {
        SpinStateCode()
        print_leave_definition(
                @STATE_NAME.0.str@,
                EmptyStrList,
                stdout);
        print_enter_atomic(@STATE_NAME.str@,stdout);
        }

@spin @revorder(1) if (spin) {
        SpinSetParent()
        }
```

⟨*state_def:CONDITIONAL STATE_NAME ':'*⟩+≡

```
@spin if (spin) {
        printf("\n/******Conditional states not supported*******/\n");
        }
```

⟨*state_charts_list:state_def*⟩+≡

⟨*state_charts_list:state_charts_list state_def*⟩+≡

10.6.4 For the constants productions

WARNING: The way we up constant declarations is a real kludge. Constants are declared as integers; and things work only if the real literal is of the form `xxx.0`.

⟨*constant_def:CONSTANT CONSTANT_NAME ':' VALUE ':' REAL END CONSTANT* ⟩+≡

```
@spin if (spin) {
        print_spin_code("define constants",
                        define_int_code(@CONSTANT_NAME.str@,
                                @REAL.val@->integer_part()),
                        stdout);
        }
```

⟨*constant_def:CONSTANT CONSTANT_NAME ':' VALUE ':' '-' REAL END CONSTANT* ⟩+≡

```
@spin if (spin) {
        print_spin_code("define constants",
                        define_int_code(@CONSTANT_NAME.str@,
                                -@REAL.val@->integer_part()),
                        stdout);
        }
```

⟨*constant_ref:CONSTANT_NAME* ⟩+≡

⟨*constant_def_list:*⟩+≡

⟨*constant_def_list:constant_def_list constant_def* ⟩+≡

10.6.5 For the `variables` productions

For now, we don't model out variables or interfaces. We model an in interface as a channel of size 1, on which enter events of the `In_Interface` type. This type is a struct with two kinds of components: booleans, modeling the arrival of events at the interface; bytes, containing the values received through the interface:

```
typedef Component_In_Interface {
        bool interface1;
        bool interface2;
        ...
        byte in_var1;
```

```
        byte in_var2;
        ...
};

chan Component_In_Channel = [1] of {Component_In_Interface};
```

Thus, the declaration of each in variable and of each in interface enters one component into this struct, by entering it into the "component of in interface" module. We will declare a local variable, `input` that gets these input values, via a `? input` command.

We also declare local variables, corresponding exactly to the RSML input variables, which are set by receipt of inputs:

```
        int in_var1;
        int in_var2;
        ...
```

That is, `in_var1` will be updated from `input.in_var1`. In the current, simplified, semantics this extra layer is unnecessary (since we could work directly with `input.in_var1`), but it may add some flexibility later in modeling variables as functions of time. We enter these into the "declare in variables" module.

⟨*in_variable_def:IN_VARIABLE IN_VAR_NAME ':' TYPE ':' NUMERIC expected_min expected_max min_gran max_gran END IN_VARIABLE*⟩+≡

```
  @spin if (spin) {
        print_spin_code("component of in interface",
                        @IN_VAR_NAME.str@->prepend("\tbyte ")->append(";\n"),
                        stdout);

        print_spin_code("declare in variables",
                        @IN_VAR_NAME.str@->prepend("\tint ")->append(";\n"),
                        stdout);
        }
```

⟨*out_variable_def:OUT_VARIABLE OUT_VAR_NAME ':' TYPE ':' NUMERIC expected_min expected_max min_gran max_gran ASSIGNMENT ':' expression TRIGGER ':' event_ref END OUT_VARIABLE* ⟩+≡

⟨*expected_min:*⟩+≡

⟨*expected_min:EXPECTED_MIN ':' REAL* ⟩+≡

⟨*expected_max:*⟩+≡

⟨*expected_max:EXPECTED_MAX ':' REAL* ⟩+≡

⟨*min_gran:*⟩+≡

⟨*min_gran:MIN_GRAN ':' REAL* ⟩+≡

⟨*max_gran:*⟩+≡

⟨*max_gran:MAX_GRAN ':' REAL* ⟩+≡

⟨*in_variable_ref:IN_VAR_NAME* ⟩+≡

⟨*out_variable_ref:OUT_VAR_NAME* ⟩+≡

⟨*in_variable_ass:IN_VAR_NAME* ⟩+≡

⟨*out_variable_ass:OUT_VAR_NAME* ⟩+≡

⟨*in_variable_def_list:*⟩+≡

⟨*in_variable_def_list:in_variable_def_list in_variable_def*
⟩+≡

⟨*out_variable_def_list:*⟩+≡

⟨*out_variable_def_list:out_variable_def_list out_variable_def*
⟩+≡

⟨*in_variable_list:*⟩+≡

⟨*in_variable_list:in_variable_ass* ⟩+≡

⟨*in_variable_list:in_variable_list ',' in_variable_ass* ⟩+≡

⟨*variable:out_variable_ref* ⟩+≡

⟨*variable:in_variable_ref* ⟩+≡

⟨*variable_ref:PREV '(' REAL ')' in_variable_ref* ⟩+≡

10.6.6 For the events productions

We generate an integer code for each event; in addition, for each state event, we generate one clause of the test for whether to break out of the microloop.

⟨*Ox Macros*⟩+≡
```
@macro SpinEventCode()
  print_spin_code("define events",
                 define_int_code(@EVENT_NAME.0.str@,
                                 @event_def.code@),
                 stdout);
@end
```

⟨*event_def_list:*⟩+≡

⟨*event_def_list:event_def_list event_def* ⟩+≡

If any new state event is generated in a pass of the microloop we must set the variable `continue_microloop`:

```
if
:: event[EVENT_NAME] -> continue_microloop = TRUE; break
:: else -> skip
fi;
```

The point of the `break` is that once we've found one such there's no need to test any of the others.

⟨*event_def:EVENT EVENT_NAME STATE ';'* ⟩+≡

```
@spin if (spin) {
        SpinEventCode()

        fprintf(stdout,"\n<<check next state event>>=\n");
        fprintf(stdout,"event[");
        @EVENT_NAME.str@->print(stdout);
        fprintf(stdout,"] ||");
        }
```

⟨*event_def:EVENT EVENT_NAME VARIABLE ';'* ⟩+≡

```
@spin if (spin) {
        SpinEventCode()
        }
```

⟨*event_def:EVENT EVENT_NAME INTERFACE ';'* ⟩+≡

```
@spin if (spin) {
        SpinEventCode()
        }
```

⟨*event_ref:EVENT_NAME* ⟩+≡

⟨*event_list:*⟩+≡

⟨*event_list:event_ref* ⟩+≡

⟨*event_list:event_list ',' event_ref* ⟩+≡

10.6.7 For the interfaces productions

For the stuff put int "component of interface" see section 10.6.5. We compile the interface declaration

```
IN_INTERFACE TempSensor :
        SOURCE   : EXTERNAL
        TRIGGER  : RECEIVE (Temp_Status, Temperature)
```

```
        SELECTION : TRUE
        ACTION    : Temp_Report_Event
END IN_INTERFACE
```

into one clause of an "if" statement:

```
:: input.TempSensor && TRUE ->
        event[Temp_Report_Event] = TRUE;
        Temperature = input.Temperature;
        Temp_Status = input.Temp_Status;
```

This code goes into the "receive inputs" module, which will itself be surrounding by the `if ... fi;` brackets. There is no "else" clause: our assumption is that for each input, precisely one of the boolean flags will be set.

NOTE: For now, the `SELECTION` will always be dummied to `TRUE`.

⟨*in_interface_def_list:*⟩+≡

⟨*in_interface_def_list:in_interface_def_list in_interface_def*
⟩+≡

⟨*in_interface_def:IN_INTERFACE IN_INTERFACE_NAME ':' SOURCE ':' component_location TRIGGER ':' RECEIVE '(' in_variable_list ')' selection_condition ACTION ':' event_list END IN_INTERFACE*
⟩+≡

```
@spin if (spin) {
        print_spin_code("component of in interface",
                        @IN_INTERFACE_NAME.str@->prepend("\tbool ")
                                ->append(";\n"),
                        stdout);

        spin_receive_inputs(@IN_INTERFACE_NAME.str@,
                         @in_variable_list.computed_val@,
                         true_S_ptr,
                         @event_list.computed_events@,
                         stdout);

        }
```

⟨*out_interface_def_list:*⟩+≡

⟨*out_interface_def_list:out_interface_def_list out_interface_def*
⟩+≡

⟨*out_interface_def:OUT_INTERFACE OUT_INTERFACE_NAME ':' DESTINATION ':' component_location TRIGGER ':' event_ref selection_condition ACTION ':' SEND '(' parameter_list ')' END OUT_INTERFACE* ⟩+≡

⟨*out_interface_def:OUT_INTERFACE OUT_INTERFACE_NAME ':' DESTINATION ':' component_location ',' IN_INTERFACE_NAME TRIGGER ':' event_ref selection_condition ACTION ':' SEND '(' parameter_list ')' END OUT_INTERFACE* ⟩+≡

⟨*selection_condition:*⟩+≡

⟨*selection_condition:SELECTION ':' condition* ⟩+≡

10.6.8 For the `macros` productions

10.6.9 For the `functions` productions

10.6.10 For the `transition busses` productions

10.6.11 For the `transitions` productions

We generate an integer code for each transition and define the effect of each transition. We also generate the set of assignment statements that sets the `trans_enabled` array at the top of each microloop.

⟨*Ox Macros*⟩+≡
```
@macro SpinTransitionCode()
  print_spin_code("define transitions",
                  define_int_code(@TRANSITION_NAME.0.str@,
                                  @transition_def.code@),
                  stdout);
@end
```

Suppose that transition `tr` is

- triggered by `Trig`

- under condition Sitch
- goes from OutOf
- goes to InTo
- actually leaves Leaving
- and generates actions a1, a2, ...

We generate the following sequence of definitions:

```
#define cond_tr Sitch
#define enabled_tr state[OutOf] && event[Trig] && cond_tr
#define take_tr selected[tr]=FALSE; leave_Leaving; enter_InTo;
#define action_tr event[Trig] = FALSE;
                  event[action1]=TRUE; event[action2]=TRUE; ...;
```

In module "update trans_enabled" we enter statements of the form:

```
trans_enabled[trans] = enabled_trans;
```

for each transition trans.

In module "make allowed transitions" we enter statements of the form:

```
if
:: selected[t1] -> take_t1; action_t1
:: else ->skip
fi;
```

for each transition trans.

⟨*transition_def_list:*⟩+≡

⟨*transition_def_list:transition_def_list transition_def*
⟩+≡

⟨*transition_def:TRANSITION TRANSITION_NAME FROM STATE_NAME TO STATE_NAME ':' transition_maybe_trigger transition_maybe_cond transition_maybe_action END TRANSITION* ⟩+≡

```
@spin if (spin) {
        SpinTransitionCode()
        print_transition_def(
                @TRANSITION_NAME.str@,
                @transition_def.trigger_name@,
```

```
        @transition_def.fml@,
        @transition_def.actions_names@,
        @transition_def.to_name@,
        @transition_def.from_name@,
        @transition_def.leaves_name@,
        stdout);

print_spin_code(
        "update trans_enabled",
        select1(TransEnabled,@TRANSITION_NAME.str@)->
                append(" = ")->
                append(PREFIX_ENABLED(@TRANSITION_NAME.str@))->
                append(";"),
        stdout);

print_spin_code(
        "make allowed transitions",
        select1(Selected,@TRANSITION_NAME.str@)->
                prepend("if\n:: ")->
                append(" -> ")->
                append(COMPOSE2(PREFIX_TAKE(@TRANSITION_NAME.str@),
                                PREFIX_ACTION(@TRANSITION_NAME.str@)))->
                append("\n:: else -> skip\nfi;"),
        stdout);
}
```

⟨*transition_maybe_trigger:*⟩+≡

⟨*transition_maybe_trigger:TRIGGER ':' event_ref*⟩+≡

⟨*transition_maybe_trigger:TRIGGER ':' TIMEOUT '(' simple_expression ',' expression ')'*⟩+≡

⟨*transition_maybe_cond:*⟩+≡

⟨*transition_maybe_cond:CONDITION ':' condition*⟩+≡

⟨*transition_maybe_action:*⟩+≡

⟨*transition_maybe_action:ACTION ':' event_list* ⟩+≡

10.6.12 For the predicate productions

10.6.13 For the conditions productions

10.6.14 For the expressions productions

10.7 Bodies of auxiliary functions

Because this text is being run through notangle the module brackets which are to appear in the printed code must be escaped with @-signs.

NOTE: Currently assuming that the number of events to be counted is the total number of events—but we may want separate arrays for separate kinds of events.

NOTE: Right now only one chunk of skeleton text is stored in a separate file and simply "grabbed"; it would probably make life easier to maintain more of the text that way.

⟨*Include files*⟩+≡

```
#include "skeleton_utilities.h"
```

The text of module "spin skeleton" provides the top-level organization of the SPIN code. It will be notangled into the file `_spin_skeleton`, and retrieved by `grab` inside the body of `spin_skeleton`. NOTE: The initial blank line in the first definition of "spin skeleton" is crucial—it guarantees that the start symbol occurs at the beginning of a line.

⟨*spin skeleton*⟩≡

```

<<*>>=

#define TRUE 1
#define FALSE 0
#define TIME int

<<set up state array>>
<<define states>>
<<define set parent>>
<<define leave actions>>
<<define conditional transitions>>
<<define enter actions>>
<<define constants>>
<<define interface type>>
<<set up event array>>
```

```
<<define events>>
<<define rsml functions>>
<<define rsml macros>>
<<set up transition arrays>>
<<define transitions>>
<<define main spin process>>
<<define driver>>
<<run process>>
```

Declare the type and channel of the In interface

⟨*spin skeleton*⟩+≡

```
<<define interface type>>=
typedef <<name of interface type>> {
<<component of in interface>>
};

chan <<name of interface channel>> = [1] of {<<name of interface type>>};
```

Define the main SPIN process

⟨*spin skeleton*⟩+≡

```
<<define main spin process>>=
proctype <<name of component>> () {
        int i;
        bool continue_microloop;
        <<name of interface type>> input;

        <<declare in variables>>

        <<initialize compat>>

        enter_<<name of top level state>>;

end: do /* Label makes this an OK end-state */
::
        <<name of interface channel>> ? input;

atomic{do  /* Top of microstep loop */
::

if
<<receive inputs>>
```

```
fi;

<<update trans_enabled>>

i = 0;
do
:: i < transition_count ->

        if
        :: trans_enabled[i] && ! selected[i] ->
            int j = 0;
            selected[i] = TRUE;
            do
                ::
                if
                :: j < transition_count ->
                    if
                    :: selected[j] && ! compatible(i,j) ->
                        selected[i] = FALSE; break
                    :: else -> skip
                    fi;
                :: else -> skip
                fi;
                j++;
            od;

        :: else -> skip
        fi;
        i++;

:: else -> break
od;

<<make allowed transitions>>

<<check event consumption>>

i = 0;
do
      :: i < transition_count -> trans_enabled[i]=FALSE; i++
      :: else -> break;
od;

continue_microloop =
```

```
<<check next state event>>
FALSE;

/* The break in this next statement quits the microloop*/
if
:: continue_microloop = FALSE -> break
:: else -> skip
fi;

od; /* end of micro step loop */
}

<<output actions>>

od /* end of macro loop */
}
```

Define the driver

⟨*spin skeleton*⟩+≡

```
<<define driver>>=
proctype Driver () {
<<get driver body>>
}
```

Define the initial process

⟨*spin skeleton*⟩+≡

```
<<run process>>=
init { run <<name of component>> ();
       run Driver()}
```

Boilerplate

⟨*spin skeleton*⟩+≡

```
<<define set parent>>=
<<define states>>=
<<define leave actions>>=
<<define conditional transitions>>=
<<define enter actions>>=
<<define constants>>=
<<define events>>=
```

```
<<define transitions>>=
<<define rsml functions>>=
<<define rsml macros>>=
<<component of in interface>>=
<<name of interface type>>=
<<name of interface channel>>=
<<name of top level state>>=
<<receive inputs>>=
<<update trans_enabled>>=
<<find maximal enabled set>>=
<<make allowed transitions>>=
<<check event consumption>>=
<<check next state event>>=
<<output actions>>=
<<run process>>=
<<declare in variables>>=
```

⟨*Bodies*⟩+≡

```
void spin_skeleton(){

  grab("_spin_skeleton",stdout);

// Setting up state definitions; note that SPIN doesn't like
// arrays of size 0

  printf("\n<<set up state array>>=\n");
  if ( glob.number_of_states() > 0 )
    { printf("#define state_count %d\n",glob.number_of_states());
      printf("bool state[state_count];\n");
    }

// Setting up event definitions

  printf("\n<<set up event array>>=\n");
  if ( glob.number_of_events() > 0 )
    { printf("#define event_count %d\n",glob.number_of_events());
      printf("bool event[event_count];");
    }

// Setting up transition definitions

  printf("\n<<set up transition arrays>>=\n");
```

```
  if ( glob.number_of_events() > 0 )
    { printf("#define transition_count %d\n",
              glob.number_of_transitions());
      printf("bool trans_enabled[transition_count];\n");
      printf("bool selected[transition_count];\n");
      printf("#define t_count_squared %d\n",
              glob.number_of_transitions() * glob.number_of_transitions());
      printf("bool compat[t_count_squared];\n");
      printf("#define compatible(i,j) compat[i * transition_count + j]\n");
    }

// Initializing compat

  printf("\n<<initialize compat>>=\n");
  spin_initialize_compat(glob,stdout);

// The body of the driver

  printf("\n<<get driver body>>=\n");
  grab("_driver_body",stdout);

};
```

Chapter 11

Parsing RSML

```
/*****************************************************************
 *
 * This is a hacked up version of the tokens.lex file from the U. of
 * Washington.  I've stripped out all their symbol table
 * manipulations, replacing them with simple-minded entries into a
 * hash table.  And I've added Ox code that will set the Str attribute
 * of each of the many kinds of names.
 *
 *****************************************************************/

/* Lexical analysis.  Lexical analysis is broken up into two passes; the
   first creates the system object, all component objects, and some of the
   objects found in the system.  The second pass is a normal tokenizing
   pass for the parser.  The object created during the first pass are
   initialized, and other objects are created and initialized during the
   second pass.   */

/*
 * States:
 *
 *    Pass #1: entering things into symbol table
 *    ==============================================
 *    <F0>: Looking for declaration keyword
 *    <F1>: Reading particular declaration keyword
 *    <F2>: Reading ending declaration keyword
 *
 *    Pass #2: Normal tokenizing
 *    ==============================================
 *    <P> : Normal tokenizing
 *
```

```
 */

%{
#include <string.h>
#include <stdio.h>
#include <search.h>
#include "configuration.h"
#include "y.tab.h"
#include "generated_include_file.h"
#include "table.h"
#include "s_exp_utilities.h"
#include "rsml_state.h"

#define display_table name_table.display


/*------------------ variable declarations -----------------------------*/


/* Temporary token used during the first pass to remember the declaration
   type so it can be assigned to the symbol. */

static int declToken;

extern table name_table;     // Stands in for symbol table
                             // Declared in load.cc,
                             // so it persists between invocations
                             // of yylex and yyparse.

extern rsml_state rsml;     // Statistics gathered during parsing.
                            // Also declared in load.cc.

/* These are used as local variables when inserting or searching for
/* entries in the hash table. */

static TABLE_DATA *data_found;
static int *token_ptr;

#define set_token(token) {declToken = token; token_ptr = new TABLE_DATA; \
                          *token_ptr = declToken; }

#define MAX_LINE_LENGTH 255
int         parser_passNum;

/* Ox seemed to be having trouble with this as a local declaration. */

%}
```

```
/*---------------------- GRAMMAR DECLARATIONS --------------------------*/

%START IDENT Z
%x F0 F1 F2 P P3

/* Replace INTEGER with REAL - Vivek 4/2 */

IDENT           [a-z_A-Z][a-z_A-Z0-9]*
REAL_LIT         ([0-9][0-9_]*([.][0-9_]+)?([Ee][-+]?[0-9_]+)?)|\
([0-9][0-9_]*#[0-9a-fA-F_]+([.][0-9a-fA-F_]+)?#([Ee][-+]?[0-9_]+)?)
STRING_LIT      \"([^"]*(\"\")*)*\"
CHAR_LIT        \'[^\n]\'

/* For setting the Str attribute of the various kinds of names */

/* I killed the stuff, just after branching on parser_passNum, that
swallowed up comments. */

/*------------------------ START OF GRAMMAR -----------------------------*/
%%
   if (parser_passNum == 0) {
       BEGIN(F0);
   } else if (parser_passNum == 1) {
       BEGIN(P);
   }

<F0,F1,F2,P>"/*" { int c;
                   do {
      while ((c = yyinput()) != '*' &&
              c != EOF) { /* eat up comment */
      }
      if (c == '*') {
    while ((c = yyinput()) == '*');
  if (c == '/') {
      break;
  }
                        }
                   } while(TRUE);
                 }


<F0>###                         { yyterminate(); }
<F0>SYSTEM                      { set_token(SYSTEM_NAME);       BEGIN(F1);}
<F0>COMPONENT                   { set_token(COMPONENT_NAME);    BEGIN(F1);}

<F0>AND_STATE                   { set_token(STATE_NAME);
```

```
        rsml.bump_state_count(); BEGIN(F1);}
<F0>OR_STATE                    { set_token(STATE_NAME);
        rsml.bump_state_count(); BEGIN(F1);}
<F0>ATOMIC                      { set_token(STATE_NAME);
        rsml.bump_state_count(); BEGIN(F1);}
<F0>CONDITIONAL                 { set_token(STATE_NAME);
        rsml.bump_state_count(); BEGIN(F1);}
<F0>CONSTANT                    { set_token(CONSTANT_NAME);     BEGIN(F1);}
<F0>EVENT                       { set_token(EVENT_NAME);
        rsml.bump_event_count(); BEGIN(F1);}

<F0>IN_INTERFACE                { set_token(IN_INTERFACE_NAME);  BEGIN(F1);}
<F0>OUT_INTERFACE               { set_token(OUT_INTERFACE_NAME); BEGIN(F1);}
<F0>IN_VARIABLE                 { set_token(IN_VAR_NAME);        BEGIN(F1);}
<F0>OUT_VARIABLE                { set_token(OUT_VAR_NAME);       BEGIN(F1);}
<F0>FUNCTION                    { set_token(FUNCTION_NAME);      BEGIN(F1);}
<F0>MACRO                       { set_token(MACRO_NAME);         BEGIN(F1);}
<F0>TRANSITION                  { set_token(TRANSITION_NAME);
       rsml.bump_transition_count();    BEGIN(F1);}
<F0>TRANSITIONBUS               { set_token(TRANSITIONBUS_NAME); BEGIN(F1);}

<F0>END  { BEGIN(F2); }

<F0>[a-z_][_a-z0-9]*          ;
<F1>[a-z_][_a-z0-9]* { switch (declToken) {
 case SYSTEM_NAME:
 default:
   name_table.insert(strdup(yytext),token_ptr);
   break;
 }
BEGIN(F0);
}

<F0,F1,F2>[\n]           ;
<F0,F1,F2>[ \t]                 ; /* Eat spaces */
<F0>.                           ; /* Eat rubbish */
<F2>[a-z_][_a-z0-9]*  { BEGIN(F0); }


 /*****************************************************************
  *
  * Pass #2
  *
  *****************************************************************/

<P>SYSTEM {return(SYSTEM);}
<P>COMPONENT {return(COMPONENT);}
```

```
<P>ACTION {return(ACTION);}
<P>AND_ARRAY {return(AND_ARRAY);}
<P>AND_STATE {return(AND_STATE_TOKEN);}
<P>ASSIGNMENT {return(ASSIGNMENT);}
<P>ATOMIC {return(ATOMIC);}
<P>CASE {return(CASE);}
<P>CONDITION {return(CONDITION);}
<P>CONDITIONAL {return(CONDITIONAL);}
<P>CONSTANT {return(CONSTANT);}
<P>DESTINATION         {return(DESTINATION);}
<P>END {return(END);}
<P>ENTERED {return(ENTERED);}
<P>EVENT {return(EVENT);}
<P>EXISTS {return(EXISTS);}
<P>EXPECTED_MAX         {return(EXPECTED_MAX);}
<P>EXPECTED_MIN         {return(EXPECTED_MIN);}
<P>EXTERNAL         {return(EXTERNAL);}
<P>FALSE {return(FALSE_TOKEN);}
<P>FROM  {return(FROM);}
<P>DEFAULT                    {return(DEFAULT);}
<P>FORALL {return(FORALL);}
<P>FUNCTION {return(FUNCTION);}
<P>IF {return(IF);}
<P>INTERFACE         {return(INTERFACE);}
<P>IN_INTERFACE         {return(IN_INTERFACE);}
<P>IN_ONE_OF {return(IN_ONE_OF);}
<P>IN_STATE {return(IN_STATE);}
<P>IN_VARIABLE         {return(IN_VARIABLE);}
<P>LOAD {return(LOAD);}
<P>LOCATION {return(LOCATION);}
<P>MACRO {return(MACRO);}
<P>MAX_GRAN {return(MAX_GRAN);}
<P>MIN_GRAN {return(MIN_GRAN);}
<P>NOT {return(NOT);}
<P>NUMERIC {return(NUMERIC);}
<P>OR_STATE {return(OR_STATE_TOKEN);}
<P>OR_ARRAY         {return(OR_ARRAY);}
<P>OUT_INTERFACE         {return(OUT_INTERFACE);}
<P>OUT_VARIABLE         {return(OUT_VARIABLE);}
<P>PREV {return(PREV);}
<P>RECEIVE {return(RECEIVE);}
<P>RETURN {return(RETURN);}
<P>SELECTION {return(SELECTION);}
<P>SEND {return(SEND);}
<P>SOURCE {return(SOURCE);}
<P>STATE {return(STATE);}
```

```
<P>TABLE {return(TABLE);}
<P>THIS {return(THIS);}
<P>TIME {return(TIME);}
<P>TIMEOUT {return(TIMEOUT);}
<P>TO {return(TO);}
<P>TRANSITION         {return(TRANSITION);}
<P>TRANSITIONBUS         {return(TRANSITIONBUS);}
<P>TRIGGER {return(TRIGGER);}
<P>TRUE {return(TRUE_TOKEN);}
<P>TYPE {return(TYPE);}
<P>VALUE {return(VALUE);}
<P>VARIABLE         {return(VARIABLE);}
<P>EQ_ONE_OF {return(EQ_ONE_OF);}

<P>";" {return(';'); }
<P>"." {return('.'); }
<P>"T" {return('T'); }
<P>"F" {return('F'); }
<P>"," {return(','); }
<P>":" {return(':'); }
<P>"(" {return('('); }
<P>")" {return(')'); }
<P>"<=" {return(LESS_OR_EQUAL);}
<P>">="         {return(GREATER_OR_EQUAL);}
<P>"<" {return('<');}
<P>">" {return('>');}
<P>"!="         {return(NOT_EQUAL);}
<P>"=" {return('=');}
<P>"+"                    {return('+');}
<P>"-"                        {return('-');}
<P>"*"                        {return('*');}
<P>"/" {return('/');}
<P>"{" {return('{');}
<P>"}" {return('}');}
<P>"[" {return('[');}
<P>"]" {return(']');}
<P>[ \t]    {;}
<P>[\n]   {;}
<P>{IDENT}        {
    data_found = name_table.find(yytext);

    if (data_found == NULL) {
        return LOCAL_VAR_NAME;
@{
  @LOCAL_VAR_NAME.str@ = new Str(yytext);
          @LOCAL_VAR_NAME.as_S_exp@ = new Application(new Str(yytext));
@}
```

```
}
else {
  int tok = *data_found;
  switch (tok) {
  case(SYSTEM_NAME):
    return SYSTEM_NAME;
    @{
      @SYSTEM_NAME.str@ = new Str(yytext);
    @}
    break;
  case(COMPONENT_NAME):
    return COMPONENT_NAME;
    @{
      @COMPONENT_NAME.str@ = new Str(yytext);
    @}
    break;
  case(STATE_NAME):
    return STATE_NAME;
    @{
      @STATE_NAME.str@ = new Str(yytext);
      @STATE_NAME.as_S_exp@ = new Application(new Str(yytext));
    @}
    break;
  case(EVENT_NAME):
    return EVENT_NAME;
    @{
      @EVENT_NAME.str@ = new Str(yytext);
    @}
    break;
  case(OUT_INTERFACE_NAME):
    return OUT_INTERFACE_NAME;
    @{
      @OUT_INTERFACE_NAME.str@ = new Str(yytext);
    @}
    break;
  case(IN_INTERFACE_NAME):
    return IN_INTERFACE_NAME;
    @{
      @IN_INTERFACE_NAME.str@ = new Str(yytext);
    @}
    break;
  case(CONSTANT_NAME):
    return CONSTANT_NAME;
    @{
      @CONSTANT_NAME.str@ = new Str(yytext);
      @CONSTANT_NAME.as_S_exp@ = new Application(new Str(yytext));
    @}
```

```
        break;
      case(IN_VAR_NAME):
        return IN_VAR_NAME;
        @{
          @IN_VAR_NAME.str@ = new Str(yytext);
          @IN_VAR_NAME.as_S_exp@ = new Application(new Str(yytext));
        @}
        break;
      case(OUT_VAR_NAME):
        return OUT_VAR_NAME;
        @{
          @OUT_VAR_NAME.str@ = new Str(yytext);
          @OUT_VAR_NAME.as_S_exp@ = new Application(new Str(yytext));
        @}
        break;
      case(FUNCTION_NAME):
        return FUNCTION_NAME;
        @{
          @FUNCTION_NAME.str@ = new Str(yytext);
        @}
        break;
      case(MACRO_NAME):
        return MACRO_NAME;
        @{
          @MACRO_NAME.str@ = new Str(yytext);
        @}
        break;
      case(TRANSITION_NAME):
        return TRANSITION_NAME;
        @{
          @TRANSITION_NAME.str@ = new Str(yytext);
        @}
        break;
      case(TRANSITIONBUS_NAME):
        return TRANSITIONBUS_NAME;
        @{
          @TRANSITIONBUS_NAME.str@ = new Str(yytext);
        @}
        break;
    default:
      printf("***ERROR*** Unrecognized name ***");
    }
    }
}
```

```
<P>{REAL_LIT}         { return(REAL);
  @{
    @REAL.val@ =
                                    new scaled_integer(new Str(yytext));
  @}
                           }

<P>###                     { return(SECTION_SEPARATOR); }


%%
```

Chapter 12

Auxiliary C++ code

12.1 c_style.h

```
#ifndef _c_style_h_
#define __c_style_h_

#include <stdio.h>
#include "str.h"
#include "state_value.h"
#include "s_exp.h"
#include "global_table.h"

#define Space (new Str(" "))
#define LBracket (new Str("["))
#define RBracket (new Str("]"))

#define State new Str("state")
#define Selected new Str("selected")
#define Event new Str("event")
#define TransEnabled new Str("trans_enabled")

#define PREFIX_LEAVE(str) str->prepend("leave_")
#define PREFIX_ENTER(str) str->prepend("enter_")
#define PREFIX_SETPARENT(str) str->prepend("set_parent_")
#define PREFIX_COND(str) str->prepend("cond_")
#define PREFIX_ENABLED(str) str->prepend("enabled_")
#define PREFIX_TAKE(str) str->prepend("take_")
#define PREFIX_ACTION(str) str->prepend("action_")

// Usage of the macros: Str *str1, *str2, and s is a char*.
// COMPOSE_SEP2(this,that,"; ") is "this; that"
```

```
#define COMPOSE_SEP2(str1,str2,s) str1->append(s)->append(str2)
#define COMPOSE_SEP3(str1,str2,str3,s) \
        str1->append(s)->append(str2)->append(s)->append(str3)

#define COMPOSE2(str1,str2) COMPOSE_SEP2(str1,str2,"; ")
#define COMPOSE3(str1,str2,str3) COMPOSE_SEP3(str1,str2,str3,"; ")

// GENERAL NOTE: We take the simple minded strategy of modeling
// lines of SPIN code as Str's.  This leads to some pretty
// inefficient manipulations (lots of making new copies when
// doing appends and prepends).  In the long run it might be
// better to model SPIN code as S_exp's.

Str *sharp_define(Str *symb, Str *def);
// "#define symb def"  // notice, no concluding ";"

Str *half_define(Str *symb);
// "#define *symb "    // notice, concluding space

Str *select1(Str *arr, Str *index);
//  "arr[index]"

Str *select2(Str *arr, Str *index1, Str *index2);
//  "arr[index1][index2]"

Str *set_true(Str *lhs);
// "*lhs = TRUE"

Str *set_false(Str *lhs);
// "*lhs = FALSE"

Str *declare(Str *typ, Str *id);
// "*typ *id;"

// Assumes that i >= 0

char *decimal_string(int i);

// Expected arguments Str *symb, int code  (and int >= 0)
// Returns "#define *symb code"

#define define_int_code(symb,code) \
   sharp_define(symb, new Str(decimal_string(code)))

void print_spin_code(char *module_name, Str *c, FILE *f);
```

```
static Str *prefix_leave(Str *s);

static Str *prefix_enter(Str *s);

static Str *set_action(Str *s);

void print_leave_definition(Str *parent, StrList *children, FILE *f);

void print_set_parent(Str *child, Str *parent, FILE *f);

void print_enter_atomic(Str *name, FILE *f);
// "#define enter_name set_parent_name; state[name]=TRUE"


void print_enter_or(Str *name, Str *default_child, FILE *f);
// "#define enter_name set_parent_name; state[name]=TRUE;
//                     enter_default_child"

void print_enter_and(Str *name, StateValueList *children, FILE *f);
// #define enter_name set_parent_name; state[name]=TRUE;
//                     enter_children1; enter_children2; ..."


// Str *declare_struct(Str *name, StrList *parts);
// "typedef *name { ... parts ... };"  // separator is "\n\t"


void print_transition_def(Str *trans_name,
  Str *trigger_name,
  S_exp *fml,
  StrList *actions,
  Str *to_name,
  Str *from_name,
  Str *leaves_name,
  FILE *f);

// IMPORTANT: Right now, the condition is dummied out as
// "TRUE"

// IMPORTANT: If the trigger_name is the null event, "", or
// the dummy event "?", we currently treat the conjunct
// event[trigger_name] as TRUE.  And in the definition of
// action_trans_name we omit "event[trigger_name]=FALSE;"

//  #define cond_trans_name fml <as str, not s_exp>
//  #define enabled_trans_name state[from_name] &&
//                             event[trigger_name] &&
```

```
//                                cond_trans_name
//  #define take_trans_name selected[trans_name]=FALSE;
//                             leave_leaves_name; enter_to_name;
//  #define action_trans_name event[trigger_name]=FALSE;
//                               event[action1]=TRUE;
//                               event[action2]=TRUE; ...


// NOTE: The next three functions refer to the extern
// global variable glob, of type global_table.

static void compat_component(int i, int j, int k, FILE *f);
// "compat[i * transition_count +j] = k\n"

void spin_initialize_compat(global_tableg, FILE *f);

// for all i, generate
// "compat[i] = ... \n"

// Auxiliaries for spin_receive_inputs

static Str* set_event(Str *e_name);
static Str* set_in_variable(Str *v_name);

void spin_receive_inputs(Str *interface_name,
 StrList *in_var_list,
 S_exp *select_cond,
 StrList *actions,
 FILE *f);
// NOTE: for now, 'select_cond' is dummied to TRUE
// <<receive inputs>>=
// :: input.interface_name && 'select_cond' ->
//    event[action1] = TRUE;
//    event[action2] = TRUE;
//      ...
//    in_var_list1 = input.in_var_list1;
//    in_var_list2 = input.in_var_list2;
//      ...


void spin_set_array(char *name, char *length, char *value,  FILE *f);
// NOTE: This does not automatically deposit these in a
// noweb module.
// i = 0;
// do
//      :: i < length -> name[i]=value; i++
//      :: else -> break;
```

```
// od;

#endif
```

12.2 c_style.cc

```
#include <stdlib.h>
#include "c_style.h"
#include "strlist_utilities.h"
#include "state_value_utilities.h"

extern global_table glob;

#define DUMMY new Str("TRUE")

Str *sharp_define(Str *symb, Str *def){
  return (new Str("#define "))->
    append(symb)->append(Space)->append(def);

};

Str *half_define(Str *symb){
  return (new Str("#define "))->
    append(symb)->append(Space);

};

Str *select1(Str *arr, Str *index){
  return arr->
    append(LBracket)->append(index)->append(RBracket);
};

Str *select2(Str *arr, Str *index1, Str *index2){
  return arr->
    append(LBracket)->append(index1)->append(RBracket)->
      append(LBracket)->append(index2)->append(RBracket);

};

Str *set_true(Str *lhs){
  return lhs->append("=TRUE");
};

Str *set_false(Str *lhs){
```

```
  return lhs->append("=FALSE");
};

Str *declare(Str *typ, Str *id){
  return typ->append(Space)->append(id);
};

char *decimal_string(int i){

  char *s;
  int n = 1;
  int ten_to_n = 10;

  while ( i / ten_to_n != 0 )
    { n++;
      ten_to_n = ten_to_n * 10;
    }


  if ( i < 0 )
    { s = (char *)malloc(n+2); }
  else
    { s = (char *)malloc(n+1); }
  sprintf(s, "%d", i);
  return s;
};

// NOTE: If this stuff gets moved into a ".nw" file, all
// the "<<" and ">>" will have to be escaped with "@".

void print_spin_code(char *module_name, Str *c, FILE *f){
  fprintf(f,"\n<<%s>>=\n", module_name);
  c->print(f);

};

Str *prefix_leave(Str *s){
  return PREFIX_LEAVE(s);
};

Str *prefix_enter(Str *s){
  return PREFIX_ENTER(s);
};

void print_leave_definition(Str *parent, StrList *children, FILE *f){

// *prefix gives "#define leave_parent state[parent]=FALSE"
```

```
  Str *prefix = sharp_define(prefix_leave(parent),
     set_false(select1(State,parent)));

// If *children is (A, B, C), then *rest is
// (leave_A, leave_B, leave_C).

  StrList *rest = pointwise(children, prefix_leave);

  fprintf(f,"\n<<define leave actions>>=\n");
  (new StrList(prefix, rest))->print(f, "; ");

};

void print_set_parent(Str *child, Str *parent, FILE *f){

  Str *symbol = PREFIX_SETPARENT(child);
  Str *def_as;

  if ( parent->equals(new Str("")) )
    { def_as = set_true(select1(State,child)); }
  else
    { def_as =  COMPOSE2(set_true(select1(State,parent)),
 PREFIX_SETPARENT(parent));
    }

  print_spin_code("define set parent",
  sharp_define(symbol, def_as),
  f);


};

void print_enter_atomic(Str *name, FILE *f){

  Str *def_as = COMPOSE2(PREFIX_SETPARENT(name),
 set_true(select1(State,name)));

  print_spin_code("define enter actions",
  sharp_define(prefix_enter(name),def_as),
  f);

};

void print_enter_or(Str *name, Str *default_child, FILE *f){

  Str *def_as =
```

```
    COMPOSE3(PREFIX_SETPARENT(name),
     set_true(select1(State,name)),
     prefix_enter(default_child));

  print_spin_code("define enter actions",
  sharp_define(prefix_enter(name),def_as),
  f);

};

void print_enter_and(Str *name, StateValueList *children, FILE *f){
// #define enter_name set_parent_name; state[name]=TRUE;
//                    enter_children1; enter_children2; ..."


// *prefix gives
//     "#define enter_name set_parent_name; state[name]=TRUE"

  Str *def_prefix =
    COMPOSE2(PREFIX_SETPARENT(name),
     set_true(select1(State,name)));


  Str *prefix = sharp_define(prefix_enter(name),def_prefix);

  StrList *rest = pointwise(StateValListToStrList(children),
prefix_enter);

  fprintf(f,"\n<<define enter actions>>=\n");
  (new StrList(prefix, rest))->print(f, "; ");



};

static Str *set_action(Str *s){
  return set_true(select1(Event,s));
};

void print_transition_def(Str *trans_name,
  Str *trigger_name,
  S_exp *fml,
  StrList *actions,
  Str *to_name,
  Str *from_name,
  Str *leaves_name,
  FILE *f){
```

```
// #define cond_trans_name fml <as str, not s_exp>

  print_spin_code("define transitions",
  sharp_define(PREFIX_COND(trans_name),DUMMY),
  f);

// #define enabled_trans_name state[from_name] &&
//                              event[trigger_name] &&
//                              cond_trans_name
// unless, trigger_name is "" or "?"

  Str *do_this;
  if ( trigger_name->equals(NullEventName) ||
       trigger_name->equals(DummyEventName) )
    { do_this = COMPOSE_SEP2(select1(State,from_name),
     PREFIX_COND(trans_name),
     " && ");
    }
  else
    { do_this = COMPOSE_SEP3(select1(State,from_name),
     select1(Event,trigger_name),
     PREFIX_COND(trans_name),
     " && ");
    }

  print_spin_code("define transitions",
  sharp_define(PREFIX_ENABLED(trans_name),
       do_this),
  f);

// #define take_trans_name selected[trans_name]=FALSE;
//                         leave_leaves_name; enter_to_name;

  print_spin_code("define transitions",
  sharp_define(PREFIX_TAKE(trans_name),
       COMPOSE3(set_false(select1(Selected,
  trans_name)),
PREFIX_LEAVE(leaves_name),
PREFIX_ENTER(to_name))
       ),
  f);

// #define action_trans_name event[trigger_name]=FALSE;
//                           event[action1]=TRUE;
//                           event[action2]=TRUE; ...
//
```

```
  print_spin_code("define transitions",
  half_define(PREFIX_ACTION(trans_name)),
  f);

  StrList *do_this2 = pointwise(actions,set_action);

  if ( ! trigger_name->equals(NullEventName) &&
       ! trigger_name->equals(DummyEventName) )
    { do_this2 = new StrList(set_false(select1(Event,trigger_name)),
     pointwise(actions,set_action)); }
  else
    { do_this2 = pointwise(actions,set_action); }

  if ( do_this2 == NULL )
    { return; }
  else
    { do_this2->print(f, "; "); }


  };

// This prints "compatible(i,j) = TRUE" if transition i
// is compatible with transition j, and is otherwise a no-op.

static void compat_component(int i, int j, int k, FILE *f){

  if (k == 1)
    fprintf(f, "\tcompatible(%d,%d) = TRUE;\n", i, j);
};

void spin_initialize_compat(global_table g, FILE *f){
  fprintf(f,"d_step{\n");
  g.do_to_compatible(compat_component,f);
  fprintf(f,"}\n");
};

// Return "event[e_name]=TRUE"

static Str* set_event(Str *e_name){
  return set_true(select1(Event,e_name));
};

// Return "v_name = input.v_name"

static Str* set_in_variable(Str *v_name){
```

```
  return v_name->append(" = input.")->append(v_name);
};


void spin_receive_inputs(Str *interface_name,
 StrList *in_var_list,
 S_exp *select_cond,
 StrList *actions,
 FILE *f){

  StrList *event_assigs = pointwise(actions,set_event);
  StrList *vbl_assigs = pointwise(in_var_list,set_in_variable);

  fprintf(f,"\n<<receive inputs>>=\n");
  fprintf(f,":: input.");
  interface_name->print(f);
  fprintf(f,"  && TRUE ->\n");
  if ( event_assigs != NULL )
    { event_assigs->print(f,"; "); fprintf(f,";\n"); }
  if ( vbl_assigs != NULL )
    { vbl_assigs->print(f,"; "); fprintf(f,";\n"); }
  if ( vbl_assigs == NULL && event_assigs == NULL )
    { fprintf(f,"skip;\n"); }
};

void spin_set_array(char *name, char *length, char *value,  FILE *f){

  fprintf(f,"i = 0;\ndo\n\t:: i < %s -> %s[i]=%s; i++\n\t\
:: else -> break;\nod;\n",length,name,value);

};
```

12.3 conditions.h

```
#ifndef __conditions_h_
#define __conditions_h_ 1

#include "s_exp.h"
#include "simple_list.h"

enum TruthValue { tv_false, tv_true, tv_dont_care };

typedef SimpleList<TruthValue> TruthValueList;
```

```
// If pred is P and tv_list is "TF.FFT.", this returns a pointer
// to (P, not P, true, not P, not P, P, true).

S_exp_list *formula_row(S_exp *pred, TruthValueList *tv_list);

#endif
```

12.4 conditions.cc

```
#include "conditions.h"

S_exp_list *formula_row(S_exp *pred, TruthValueList *tv_list){
  if (tv_list == NULL)
    { return new S_exp_list; }
  else
    { switch (tv_list->head()) {
    case tv_false:

      return new S_exp_list(pred->Not(),
    formula_row(pred, tv_list->tail()));
    case tv_true:

      return new S_exp_list(pred,
    formula_row(pred, tv_list->tail()));
    case tv_dont_care:

      return new S_exp_list(true_S_ptr,
    formula_row(pred, tv_list->tail()));
    }
    }
};
```

12.5 configuration.h

```
#ifndef __configuration_h__
#define __configuration_h__ 1

// This is a temporary kludge

#define MAX_SPEC_NAMES 1000
```

```
// All-purpose kludge

#define DummyType int

#endif
```

12.6 event_value.h

```
#ifndef __event_value_h_
#define __event_value_h_ 1

#include "simple_list.h"
#include "str.h"
#include "generated_C_macros.h"

// Presumably, the kind of an event -- whether it's associated with a
// state, a variable, or an interface, will ultimately be included in
// the private data of an event_value.

// The constructor event_value(Str*, knd)) may be a mistake, as it permits
// someone to supply a NULL pointer, and therefore ruins the
// potentially desirable representation invariant of never having a
// null pointer "inside".

enum event_kind { StateEvent, VariableEvent, InterfaceEvent };

class event_value {
public:

  event_value() {st = NullEventName; knd = StateEvent; };
  event_value(Str *s, event_kind k) {st = s; knd = k; };

  Str *str() {return st;};
  event_kind kind() {return knd;};

  int operator==(event_value s){
    if (st != NULL && s.st != NULL)
      {return st->equals(s.st) && knd == s.knd; }
    else
      {return (st == NULL) && (s.st == NULL);}
  };

  void print(FILE *f);

private:
```

```
  Str *st;
  event_kind knd;
};

typedef SimpleList<event_value> EventValueList;

#define EmptyEventValueList (EventValueList *)0

#endif
```

12.7 event_value.cc

```
#include "event_value.h"

void event_value::print(FILE *f){
  fprintf(f, "event_value: str = %s, kind = %d\n",st->str,knd);

};
```

12.8 event_value_utilities.h

```
#ifndef __event_value_utilities_h_
#define __event_value_utilities_h_ 1


#include "event_value.h"
#include "str.h"

// The state events go into the StrList.

StrList *StateEventsToStrList(EventValueList *val_list);

void print_EventValueList(EventValueList *evl);

#endif
```

12.9 event_value_utilities.cc

```
#include <stdio.h>
```

```
#include "event_value_utilities.h"

StrList *StateEventsToStrList(EventValueList *val_list){

  StrList *e_list = EmptyStrList;

  EventValueList *iter = val_list;

  while (iter != NULL){
    if (iter->head().kind() == StateEvent)
      { e_list = new StrList(iter->head().str(), e_list); }
    iter = iter->tail();
  }

  return e_list;
};

void print_EventValueList(EventValueList *evl){

  while (evl != NULL) {
    evl->head().print(stdout);
    evl = evl->tail();
  }

};
```

12.10 eves2_state.h

```
#ifndef __eves2_state_h_
#define __eves2_state_h_


class eves2_state {

 public:
  eves2_state() { call_number = 0; };

// Successive calls of suffix return "__1", "__2", "__3", etc.

  char *suffix();

 private:
```

```
  int call_number;

};


#endif
```

12.11 eves2_state.cc

```
#include "eves2_state.h"
#include "misc_utilities.h"
char *eves2_state::suffix(){
  return decorate("__",++call_number);
};
```

12.12 formula_table.h

```
#ifndef __formula_table_h__
#define __formula_table_h__ 1

/* Hacked up quickie.  (See also name_table.h) */

#include "configuration.h"
#include "s_exp.h"
#include "list.h"
#include "event_value.h"

/* The only good thing about this is the provision of a */
/* "do-it-to-the-whole-table" method. */

class formula_table {

public:

// Make a formula_table with the given keys, and all associated
// formulas initially equal to *fml.  If the length of *keys is
// greater than MAX_SPEC_NAMES only the first MAX_SPEC_NAMES will be
// entered as keys.

  formula_table(StrList *keys, S_exp *fml);

  int size() { return top; };
```

```
// Return a pointer to the formula pointed to by *key (the very same
// pointer that's in the table, so there's sharing here -- which
// does not at the moment seem like a problem).  If none, return
// NULL.

  S_exp *find(Str *key);

// Replace the entry for *key with *fml, if there is an entry, and
// return 0.  If there is no entry for *key return 1.

  int update(Str *key, S_exp *fml);

// IMPORTANT: Assumes that 'this' and *table were made from the same keys
// arranged in the same sequence.

// Modifies 'this' by disjoining to its formulas, pointwise, those
// of *table.

// The only protection is that if the two tables are of different
// length it won't run off the end of either of them.

  void disjoin(formula_table *table);

// This runs through the entire table and applies the operation entry_op
// to each (key, formula) pair.

  void do_it(void (*entry_op)(Str*, S_exp*));

  void display(FILE *f);

private:

  typedef struct entry { Str *key; S_exp *fml; } ENTRY;

// We can't use any of the list templates we have, because they all
// require that the element type have a == operator.

  ENTRY stuff[MAX_SPEC_NAMES];
  int top;

};

#define EmptyFormulaTable formula_table(EmptyStrList, (S_exp *)0);

#endif
```

12.13 formula_table.cc

```
#include "formula_table.h"
#include <stdio.h>
#include <string.h>

// Silly, since we're running through the keys twice -- once to count
// them and once to insert them into the table.  But the whole thing
// is so klutzy, who cares?

formula_table::formula_table(StrList *keys, S_exp *fml){
  if (keys == NULL) {
    top = 0;
  }
  else {
    top = (keys->len() > MAX_SPEC_NAMES)? MAX_SPEC_NAMES : keys->len();
  }

  for (int i = 0; i < top ; i++) {
    stuff[i].key = keys->hd;
    keys = keys->tl;
    stuff[i].fml = fml;
  }
};

S_exp *formula_table::find(Str *key){

  if (key == NULL) {return NULL;}

  for (int i = 0; i < top ; i++) {
    if ( stuff[i].key == NULL ) { printf("NULL key in table\n"); return NULL; }
    if ( stuff[i].key->equals(key) )
      {return stuff[i].fml;}
  }
  return NULL;
};

int formula_table::update(Str *key, S_exp *fml){

  if (key == NULL) {return NULL;}

  for (int i = 0; i < top ; i++) {
    if ( stuff[i].key == NULL ) { printf("NULL key in table\n"); return NULL; }
    if ( stuff[i].key->equals(key) ) {
      stuff[i].fml = fml;
      return 0;}
  }
```

```
  return 1;
};

void formula_table::disjoin(formula_table*table){
  if ( table == NULL ) { return; }
  for (int i = 0; (i < top && i < (*table).top); i++){
    stuff[i].fml = stuff[i].fml->Or((*table).stuff[i].fml);
  }
};

void formula_table::display(FILE *f){
  printf("******* Table value *******\n");
  for (int i = 0; i < top; i++)  {
    if (stuff[i].key == NULL) {
      printf("Printing null key. Value of i is %d\n",i);
      continue;
    }
    printf("Key: %s\n", stuff[i].key->str);
    printf("Formula: ");
    stuff[i].fml->print(f);
    printf("\n");
  }
};

void formula_table::do_it(void (*entry_op)(Str*, S_exp*)){
  for (int i = 0; i < top; i++)  {
    entry_op(stuff[i].key, stuff[i].fml);
  }

};
```

12.14 formula_table_utilities.h

```
#ifndef __formula_table_utilities_h_
#define  __formula_table_utilities_h_ 1

#include "formula_table.h"
#include "event_value.h"

// This should perhaps be #define'd instead.  It returns ftab_ptr,
// after having a side effect on *ftab_ptr, updating the event.str()
// slot of the table by disjoining *fml_ptr to it.

formula_table *DisjoinToETests(Str *key, S_exp *fml_ptr,
       formula_table *ftab_ptr);
```

```
#endif
```

12.15 formula_table_utilities.cc

```
#include "formula_table_utilities.h"

formula_table *DisjoinToETests(Str *key, S_exp *fml_ptr,
       formula_table *ftab_ptr){
  ftab_ptr->update(key,
  ftab_ptr->find(key)->Or(fml_ptr));
  return ftab_ptr;
};
```

12.16 function_definition.h

```
#ifndef __function_definition_h_
#define __function_definition_h_ 1

#include <stdio.h>
#include "s_exp.h"
#include "str.h"
#include "simple_list.h"

class rsml_case {

 public:

  rsml_case(S_exp *c, S_exp *v) {condit = c; val = v; };

  S_exp *condition() { return condit; };
  S_exp *value() { return val; };

// Phoney ops included so that we can use the List template.

  int equals(rsml_case *c2) {
    printf("****DON'T CALL equals ON rsml_case****");
    return 0;
  };

  void print(FILE *f) {
```

```
    printf("****DON'T CALL print ON rsml_case ****");
  };

 private:

  S_exp *condit;
  S_exp *val;
};

typedef List<rsml_case> CaseList;
#define EmptyCaseList (CaseList *)0

class function_definition {

 public:

  function_definition(Str *n, S_exp_list *v, CaseList *c){
    nm = n;
    vbls = v;
    c_list = c;
  };


  Str *name();
  S_exp_list *variables();
  CaseList *case_list();

  void print(FILE *f) { printf("*****DUMMY*****"); };

 private:
  Str *nm;
  S_exp_list *vbls;
  CaseList *c_list;
};

#endif
```

12.17 function_definition.cc

```
#include "function_definition.h"


Str *function_definition::name() { return nm; };

S_exp_list *function_definition::variables() { return vbls; };

CaseList *function_definition::case_list() { return c_list; };
```

12.18 function_definition_utilities.h

```
#include "function_definition.h"

S_exp_list *condition_list(CaseList *c_list);

// If c is the case (b, e), then this returns a pointer
// to (IMPLIES b (= t e)).  This, after all, is how the
// cases are used.

S_exp *as_S_exp(rsml_case *c, S_exp *t);

S_exp_list *as_S_exp_list(CaseList *c_list, S_exp *t);
```

12.19 function_definition_utilities.cc

```
#include "function_definition_utilities.h"

S_exp_list *condition_list(CaseList *c_list){

  if (c_list == NULL)
    { return new S_exp_list ; }

  S_exp *tmp = c_list->hd->condition();
```

```
  if (c_list->tl == NULL)
    { return new S_exp_list(tmp) ; }
  return new S_exp_list(tmp,
condition_list(c_list->tl));
};

S_exp *as_S_exp(rsml_case *c, S_exp *t){
  return
    new Application(new Str("IMPLIES"),
    Doubleton(c->condition(),
      new Application(new Str("="),
      Doubleton(t,c->value()))));
};

S_exp_list *as_S_exp_list(CaseList *c_list, S_exp *t){

  if (c_list == NULL)
    { return new S_exp_list ; }

  rsml_case *tmp = c_list->hd;

  if (c_list->tl == NULL)
    { return new S_exp_list(as_S_exp(tmp,t)) ; }
  return new S_exp_list(as_S_exp(tmp,t),
as_S_exp_list(c_list->tl,t));
};
```

12.20 global_table.h

```
#ifndef __global_table_h_
#define __global_table_h_ 1

#include <stdio.h>
#include "state_value.h"
#include "transition_value.h"

// For now, these are the only things I need to keep track of are:
// the correspondence between state valuess and their integer codes;
// the correspondence between transition values and their integer codes;
// the tables coding leq (among states), parallel (among states),
// and compatible (among transitions).

// I'll add member functions as I need them.
```

```
class global_table {
public:

  global_table() { state_codes = (state_value **)0;
   transition_codes = (transition_value **)0;
   child_of = leq = compatible = (int **)0;
   num_states = num_transitions =
     num_events = 0;
 };


// ********** Stuff to do with states

// Allocates state_codes (uninitialized),
// child_of, leq (initialized to 0's)

  void allocate_states(int size);

  int number_of_states() { return num_states; };
  void display_state_codes(FILE *f);
  state_value *state_at(int c)  // assumes c < num_states
    { return state_codes[c]; };

  int code_of_state(Str *s); // returns -1 if s can't be
                             // found in state_codes

// Has side effect on *this.  Assumes code of state value < num_states,
// Returns dummy value of 0, always.  Puts state_value with code
// n in slot n of state_codes.

  int set_state_code(state_value *val);

// Assumes code1, code2 < num_states; leq is complete.   Returns
// code of the sup in the state hierarchy.

  int sup(int code1, int code2);

// Assumes code1, code2 < num_states (could get into an infinite
// loop otherwise); leq is complete.  Returns code of the state
// that an arrow from code_from to code_to actually leaves.

  int leaves(int code_from, int code_to);

// ******* Stuff to do with the state hierarchy

// Has side effect on *this.  Assumes parent_code < num_states, as
// are codes of all state values in *child_values.  For each code i in
```

```
// *child_values, sets child_of[i,parent_code] = 1.

  int set_child_of(StateValueList *child_values, int parent_code);
  void display_child_of(FILE *f);

// Assumes child_of is complete.  Completes leq -- the
// reflexive, transitive closure of child_of (via Warshall's algorithm.)

  int set_leq();
  void display_leq(FILE *f);

// Assumes code1, code2 < num_states; leq complete.

  int parallel(int code1, int code2);

// ********** Stuff to do with transitions

// Allocates transition_codes, compatible (uninitialized)

  void allocate_transitions(int size);
  int number_of_transitions() { return num_transitions; };

  int set_transition_code(transition_value *val);

// Assumes leq *and* transition_codes are *both* complete, and
// that compatible has been allocated and initialized to 0's.  Effect
// is to complete "compatible"; i.e., compatible[i][j] == 1 iff
// transitions i and j are compatible.

  int set_compatible();

  void display_transition_codes(FILE *f);
  void display_compatible(FILE *f);

// Do entry_op for each i, j, k such that compat[i][j]==k

  void do_to_compatible( void (*entry_op)(int, int, int, FILE*),
       FILE *f);

// ********* Stuff to do with events

  void set_event_count(int size) { num_events = size; };
  int number_of_events() { return num_events; };

// *********** Dummy

  int dummy(int i) { state_codes[i]->print(stdout); return 0; };
```

```
private:

// Rep invariant:
//    state_codes has size num_state
//    transition_codes has size num_transitions
//    child_of is num_state square
//    leq is num_state square
//    compatible is num_transitions square

// We may not really need to keep the integer values explicitly,
// since they can be read off the global rsml_state.

  int num_states;
  int num_transitions;
  int num_events;

  state_value **state_codes;
  transition_value **transition_codes;

  int **child_of;
  int **leq;
  int **compatible;

};


//


#endif
```

12.21 global_table.cc

```
#include "global_table.h"
#include "combinatorics.h"

#define TwoDArray(arr, typ)                              \
{ int counter, inner;                                    \
    arr = (typ **)malloc(size * sizeof(typ*));           \
    for (counter = 0; counter < size; counter++)         \
      { arr[counter] = (typ *)malloc(size * sizeof(typ)); } \
    for (counter = 0; counter < size; counter++)         \
```

```
      { for (inner = 0; inner < size; inner++)       \
          arr[counter][inner] = 0; }                  \
}

#define OneDArray(arr, typ)        \
    arr = (typ *)malloc(size * sizeof(typ))



// Allocates state_codes (uninitialized),
// child_of, leq (initialized to 0's)

void global_table::allocate_states(int size){

  num_states = size;

  OneDArray(state_codes, state_value*);
  TwoDArray(child_of, int);
  TwoDArray(leq, int);

};

// Allocates transition_codes, compatible (uninitialized)

void global_table::allocate_transitions(int size){

  num_transitions = size;

  OneDArray(transition_codes, transition_value*);
  TwoDArray(compatible, int);
};

int global_table::set_state_code(state_value*val){

  state_codes[val->code()] = val;
  return 0;
};

void global_table::display_state_codes(FILE*f){
  for (int i = 0; i < num_states; i++)
    {
      state_codes[i]->print(f);
      fprintf(f,"\n");
    }

};
```

```
int global_table::set_child_of(StateValueList*child_values,
       int parent_code){

  StateValueList *iter = child_values;

/* Diagnostics.

  printf("Input parent_code = %d\n",parent_code);
  printf("Input child_values = ");
  while ( iter != NULL )
    { iter->head().print(stdout);
      printf(" ");
      iter = iter->tail();
    }
  printf("\n");
*/

  while ( child_values != NULL )
    { child_of[child_values->head().code()][parent_code]= 1;
      child_values = child_values->tail();
    }
};

void global_table::display_child_of(FILE*f){
  for (int row = 0; row < num_states; row++)
    { state_codes[row]->str()->print(f);
      fprintf(f," < : ");
      for (int col = 0; col < num_states; col++)
{ if (child_of[row][col] == 1)
    { state_codes[col]->str()->print(f);
      fprintf(f," ");
    }
}
      fprintf(f,"\n");
    }

};

int global_table::set_leq(){
  trans_reflex_closure(child_of,num_states, leq);
  return 0;
};

void global_table::display_leq(FILE*f){
  for (int row = 0; row < num_states; row++)
    { state_codes[row]->str()->print(f);
      fprintf(f," <= : ");
```

```
      for (int col = 0; col < num_states; col++)
{ if (leq[row][col] == 1)
    { state_codes[col]->str()->print(f);
      fprintf(f," ");
    }
}
      fprintf(f,"\n");
    }

};

int global_table::code_of_state(Str *s){
  int i = 0;
  while ( i < num_states && ! (state_codes[i]->str()->equals(s)) ){
    i++;
  }
  if (i < num_states)
    { return i; }
  else
    { return -1; }

};

int global_table::set_transition_code(transition_value*val){
  transition_codes[val->code()] = val;
  return 0;

};

// Th "::sup" seems to tell the compiler that the sup intended is
// not this one, so it manages to find the one defined in
// combinatorics.h.  If this function gets called a lot it might
// make sense to store all its values in a table (note that,
// since it's symmetric, we only need to compute half the table).

int global_table::sup(int code1, int code2){
  return ::sup(leq, num_states, code1, code2);
};

int global_table::leaves(int code_from, int code_to){

  int s = ::sup(leq, num_states, code_from, code_to);

  if (s == code_from)
    { return code_from; }

  int i = 0;
```

```
  while ( ! (leq[code_from][i] && child_of[i][s]) )
    i++;

  return i;

};

void global_table::display_transition_codes(FILE*f){
  for (int i = 0; i < num_transitions; i++)
    {
      transition_codes[i]->print(f);
      fprintf(f,"\n");
    }
};

int global_table::parallel(int code1, int code2){
  return ( (! leq[code1][code2]) &&
   (! leq[code2][code1]) &&
   (state_at(sup(code1,code2))->kind() == AndState)
  );
};

int global_table::set_compatible(){
  int i, j;
  for (i = 0; i < num_transitions; i++)
    for (j = 0; j < i; j++)
      compatible[i][j] = compatible[j][i] =
(parallel(transition_codes[i]->source(),
  transition_codes[j]->source()) &&
 parallel(transition_codes[i]->destination(),
  transition_codes[j]->destination()));
};

void global_table::display_compatible(FILE*f){
  for (int i = 0; i < num_transitions; i++)
    {
      fprintf(f,"Transition ");
      transition_codes[i]->name()->print(f);
      fprintf(f," compat with: ");
      for (int j = 0; j < num_transitions; j++)
if (compatible[i][j])
  { transition_codes[j]->name()->print(f);
    fprintf(f, " "); }
      fprintf(f,"\n");
    }
  fprintf(f,"\n");
};
```

```
void global_table::do_to_compatible(void (*entry_op)(int, int, int, FILE*),
    FILE *f){
  for (int i = 0; i < num_transitions; i++)
    for (int j = 0; j < num_transitions; j++)
      entry_op(i,j,compatible[i][j],f);
};
```

12.22 load.h

```
/*****************************************************************
 *
 * Hacked from simmMain.H
 *
 *****************************************************************/

#ifndef _load_h
#define _load_h

int load(char *filename);

#endif
```

12.23 load.cc

```
/***********************************************************************
 *  Hacked from simmain.C -
 *
 *  Purpose: parsing
 *
 ***********************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>
#include "configuration.h"
#include "load.h"
#include "table.h"
#include "rsml_state.h"
#include "str.h"
```

```
#include "global_table.h"

extern int yylex();
extern int yyparse();

extern global_table glob;

extern int yyerror(char *);
extern FILE* yyin;
extern int parser_passNum;
extern void display_table();
extern rsml_state rsml;      // Statistics about the definition,
                             // gathered during parsing.

table name_table;    // Stands in for symbol table



//----------------------------------------------------------------------
// load is a hacked version of rsml_load --
//----------------------------------------------------------------------

int
load(char* filename)
{
    yyin = fopen(filename, "r");

    if (!yyin) {
printf("Couldn't open file\n");
return 1;
    }

    parser_passNum = 0;
    while (yylex());
    fclose(yyin);

    glob.allocate_states(rsml.state_count());
    glob.allocate_transitions(rsml.transition_count());
    glob.set_event_count(rsml.event_count());

    yyin = fopen(filename, "r");
    parser_passNum = 1;
    yyparse();    // This is what's calling Ox.

    return 0;
}
```

12.24 main.cc

```
extern char *handleCLopts(int argc, char **argv);
extern int load(char *filename);

#include "state_value.h"

main(int argc, char **argv)
{ int status;
  char *the_filename;

  the_filename = handleCLopts(argc,argv);

  status = load(the_filename);

  exit(status);
}
```

12.25 misc_utilities.cc

```
#include <stdlib.h>
#include <stdio.h>
#include "misc_utilities.h"

char *decorate(char *s, int n){

  char *decorated_s;
  int mag, ten_to_mag;

  for (mag = 1, ten_to_mag = 10; ten_to_mag <= n;
        ten_to_mag = ten_to_mag*10, mag++){;}

  decorated_s = (char *)malloc(strlen(s)+mag+1);
  sprintf(decorated_s, "%s%d", s, n);
  return decorated_s;

};


int yyerror(char *s){
```

```
 fprintf(stderr, "Oops!: %s\n", s);
 return 0;
};
```

12.26 misc_utilities.h

```
#ifndef __misc_utilities_h_
#define __misc_utilities_h_ 1

// If s is the string "foo" and n is, say, 17, this returns the string
// "foo17"

char *decorate(char *s, int n);

// Currentlt, this does nothing but post "Oops!" to standard error
// and return 0

int yyerror(char *);


#endif
```

12.27 pvs_state.h

```
#ifndef __pvs_state_h_
#define __pvs_state_h_

class pvs_state {

 public:
  pvs_state() { call_number = 0; };

// Successive calls of suffix return "__1", "__2", "__3", etc.

  char *suffix();

 private:

  int call_number;

};
```

```
#endif
```

12.28 pvs_state.cc

```
#include "pvs_state.h"
#include "misc_utilities.h"
char *pvs_state::suffix(){
  return decorate("__",++call_number);
};
```

12.29 pvs_types.h

```
#ifndef __pvs_types_h_
#define __pvs_types_h_

#include <stdio.h>
#include "str.h"

// Probably, this ought to follow the model of S_exp and have an
// abstract base class with children that represent the different
// kinds of type constructors.

class PVS_type_exp {

 public:
  PVS_type_exp(Str *name);

  void print(FILE *f);

 private:

  Str *type_name;

};

#define PVS_bool new PVS_type_exp(new Str("bool"))

#endif
```

12.30 pvs_types.cc

```
#include "pvs_types.h"

PVS_type_exp::PVS_type_exp(Str *name){
  type_name = name;
};

void PVS_type_exp::print(FILE *f){
  type_name->print(f);
};
```

12.31 rsml_state.h

```
#ifndef __rsml_state_h_
#define __rsml_state_h_ 1

#include "simple_list.h"

class rsml_state {

 public:
  rsml_state() { total_states = total_events = total_transitions =
 current_state = current_event =
 current_transition = 0; };

  int bump_state_count() { total_states++; };
  int state_count() { return total_states; };
  int get_state_code() { return current_state++; };

  int bump_event_count() { total_events++; };
  int event_count() { return total_events; };
  int get_event_code() { return current_event++; };

  int bump_transition_count() { total_transitions++; };
  int transition_count() { return total_transitions; };
  int get_transition_code() { return current_transition++; };

// We could have the "get" functions raise an error if things
// get screwed up and current is ever > total.

// We need to be able to set and read the top-level state.
// The "set" function should be private, and available only to
// one friend that's called during parsing.
```

```
  void print(FILE *f);

 private:

  int total_states;
  int total_events;
  int total_transitions;
  int current_state;
  int current_event;
  int current_transition;
};

#endif
```

12.32 rsml_state.cc

```
#include <stdio.h>
#include "rsml_state.h"

void rsml_state::print(FILE *f){
  fprintf(f,"states = %d\n",total_states);
  fprintf(f,"events = %d\n",total_events);
  fprintf(f,"transitions = %d\n",total_transitions);
};
```

12.33 scaled_integer.h

```
#ifndef __scaled_integer_h_
#define __scaled_integer_h_ 1

#include <stdlib.h>
#include "str.h"
```

```
// I don't know what parts of these things I'll ultimately
// want.  More information is available in local variables of the
// constructor, and that could be made public, too.

// NOTE: If the integer part and exponent part are too
// big this will produce nonsense.

enum Sign { neg, zero, pos };

class scaled_integer {

 public:

// This will assume that lit matches the YACC-style regular
// expression
//
//       ([0-9][0-9_]*([.][0-9_]+)?([Ee][-+]?[0-9_]+)?)
//
// RSML also permits hex expressions but I'll skip them for now.

  scaled_integer(Str *lit);

// For now I'll assume that theories will want to represent
// reals as (integer,exponent) pairs.

  int integer_part() { return int_val; };
  int signed_exp() { return signed_exp_val; };
  int absolute_exp() {
    return (signed_exp_val >= 0 ? signed_exp_val : -signed_exp_val) ;
  };

  Sign sign_of_exp() {
    return (signed_exp_val < 0 ?
    neg :
    (signed_exp_val == 0 ?
     zero:
     pos));
  };


// Returns pointer to equivalent (new) scaled_integer with the largest
// possible exponent.

  scaled_integer *normalized();

// Returns pointer to (new) equivalent scaled_integer with exponent
```

```
// "as close as possible" to signed_exp_val.  For example
//    rescale(real(10, 2), -2) becomes real(10_0000, -2)
//    rescale(real(10, 2), 3)  becomes real(1, 3)
//    rescale(real(10, 2), 4)  becomes real(1, 3), also --
// as there's no way to push the exponent above 3.

  scaled_integer *rescale(int new_exp);


 private:

  int int_val;
  int signed_exp_val;

  scaled_integer(int i, int e) { int_val = i;
 signed_exp_val = e; };

// Has a side-effect on "this", changing it to an
// equivalent form with greatest possible exponent.

  void norm();

};

#endif
```

12.34 scaled_integer.cc

```
#include "scaled_integer.h"

scaled_integer::scaled_integer(Str *lit){

  int length = lit->len();
  int s_loc = 0;
  int b_loc = 0;
  int a_loc = 0;
  int e_loc = 0;

  Sign exp_sign = pos;

// The next strings will hold, respectively, the string
// before the decimal point, the string after the decimal
// point, and the exponent string.  To each we allocate the
// maximum possible space (slightly wasteful).
```

```
  char *before_dp, *after_dp, *e;

  before_dp = (char *)(malloc(length)+1);
  after_dp = (char *)(malloc(length)+1);
  e = (char *)(malloc(length)+1);

// Discarding initial zeroes

  while (s_loc < length && lit->str[s_loc] == '0')
    { s_loc++; }

// Filling up before_dp:

  while ( s_loc < length && lit->str[s_loc] != 'E' &&
  lit->str[s_loc] != 'e' && lit->str[s_loc] != '.')

// Loop invar: s_loc <= length
//        and   before_dp[0..b_loc-1] = lit->str[0..s_loc-1] with
//                                       '_' removed
//        and   lit->str[0..s_loc-1] has no 'E', 'e', or '.'
//
// Note: It follows automatically that b_loc <= s_loc

    {
      if ( lit->str[s_loc] != '_'  )
{
  before_dp[b_loc] = lit->str[s_loc];
  b_loc++;
}
      s_loc++;
    }

// Since b_loc <= s_loc <= length, the following assignment is OK:

  before_dp[b_loc] = '\0';

// If s_loc < length, lit->str[s_loc] is 'E', 'e', or '.'

// Filling up after_dp:

  if (lit->str[s_loc] == '.')
    {
      s_loc++;
      while ( s_loc < length && lit->str[s_loc] != 'E' &&
     lit->str[s_loc] != 'e' )
{
  if ( lit->str[s_loc] != '_'  )
```

```
    {
      after_dp[a_loc] = lit->str[s_loc];
      a_loc++;
    }
  s_loc++;
}

    }

  after_dp[a_loc] = '\0';

// If s_loc < length, lit->str[s_loc] is // 'E' or 'e'.

  s_loc++;

// Filling up e.  If the next string begins with a '-'
// we change the value of exp_sign to neg.

  while ( s_loc < length )

    {
      if ( lit->str[s_loc] == '-' )
{ exp_sign = neg;
  s_loc++;
}
      else
{ if ( lit->str[s_loc] != '+' && lit->str[s_loc] != '_')
    {
      e[e_loc] = lit->str[s_loc];
      e_loc++;
    }
  s_loc++;
}
    }

  e[e_loc] = '\0';

// Tiny normalization for the special case in which after_dp
// is 0 (and final correction of exp_sign, for 0 exponent).

  if ( atoi(after_dp) == 0 )
    { int_val = atoi(before_dp);
      signed_exp_val =
(exp_sign == pos ? atoi(e) : -atoi(e));
      if (signed_exp_val == 0) { exp_sign = zero; }
    }
```

```
// Like previous case, but taking after_dp into account

  else
    { char *dum;
      dum = (char *)(malloc(length+1));
      strcpy(dum,before_dp);
      strcpy(dum + b_loc,after_dp);
      int_val = atoi(dum);
      signed_exp_val =
(exp_sign == pos ? atoi(e) : -atoi(e)) - strlen(after_dp);
      if (signed_exp_val == 0) { exp_sign = zero; }
    }

/*

  printf("Just generated the scaled integer:\n");
  printf("integer = %d, exponent = %d\n",int_val,signed_exp_val);
*/



};

void scaled_integer::norm(){
  if ( int_val == 0 )
    { signed_exp_val = 0; return; }
  if ( int_val % 10 == 0 )
    { int_val = int_val/10;
      signed_exp_val = signed_exp_val+1;
      norm(); }
};

scaled_integer *scaled_integer::normalized(){

  scaled_integer *si = new scaled_integer(int_val, signed_exp_val);
  si->norm();
  return si;

};

scaled_integer *scaled_integer::rescale(int new_exp){
  scaled_integer *si;
  si = normalized();

  int diff = si->signed_exp() - new_exp;

  if (diff <= 0) { return si; }
```

```
// Multiply the integer part of si by 10**diff

  for (int i = 1; i <= diff; i++)
    { si->int_val = si->int_val * 10; }

// Reduce the exp part of si by diff

  si->signed_exp_val = si->signed_exp_val - diff;

  return si;

};
```

12.35 s_exp.cc

```
#include "s_exp.h"

/******** APPLICATION ********/

void Application::print(FILE *f){
        fprintf(f,"(");
        fcn->print(f);
        if (! args->isEmpty()) {fprintf(f, " "); args->print(f);}
        fprintf(f, ")");
};

// eves_print simply changes a couple of the "native names" for
// operations: "/" --> "div", and inary "-" --> "negate"

void Application::eves_print(FILE *f){

  if ( fcn->equals(new Str("/")) )
      { (new Application(new Str("div"),args))->print(f);
return; }

  if ( fcn->equals(new Str("-")) && args->length() == 1 )
      { (new Application(new Str("negate"),args))->print(f);
return; }

  print(f);

};

// Auxiliaries for pvs_print is a little kludgy, since (AND x y)
```

```
// yields (AND x y), but does so in a roundabout way, going through
// (AND x (AND y)) as a way-station.

S_exp *Application::unwind_binary(){

  if ( args->isEmpty() )
    { return this; }
  if ( args->tail()->isEmpty() )
    { return args->head(); }

  S_exp *unwound_tail =
    Application(fcn,args->tail()).unwind_binary();

  S_exp_list *argt_pair =
    new S_exp_list(args->head(),
   new S_exp_list(unwound_tail));

  return new Application(fcn, argt_pair);
};

// This prints multi-adic ANDs and ORs as is, and writes
// "(Foo)" as as "Foo", not as "Foo()".

void Application::pre_pvs_print(FILE *f){
  fcn->print(f);
  if ( ! args->isEmpty() )
    { fprintf(f,"(");
      args->pvs_print(f, ", ");
      fprintf(f, ")"); }
};

// This should do the trick.

void Application::pvs_print(FILE *f){

  if ( strcmp(fcn->str,"AND") == 0 ||
       strcmp(fcn->str,"OR") == 0 )
    { ((Application *)(this->unwind_binary()))->pre_pvs_print(f);}
  else
    { this->pre_pvs_print(f); }
};

int Application::equals(S_exp *t){
   if (this == t) return TRUE;
   if (!t->isApplication()) return FALSE;
   if (!fcn->equals(((Application *)t)->fcn)) return FALSE;
   if (!args->equals(((Application *)t)->args)) return FALSE;
```

```
   return TRUE;
};


/******** VARIABLE ********/

void Variable::print(FILE *f){
  name->print(f);
};

void Variable::eves_print(FILE *f){
  name->print(f);
};

void Variable::pvs_print(FILE *f){
  name->print(f);
};

int Variable::equals(S_exp *t){
   if (this == t) return TRUE;
   if (!t->isVariable()) return FALSE;
   if (!name->equals(((Variable *)t)->name)) return FALSE;
   return TRUE;
};


/******** INTEGERLITERAL ********/

void IntegerLiteral::print(FILE *f){
  if ( val < 0 )
    { (new Application(new Str("-"),
       new S_exp_list(new IntegerLiteral(-val))))
       ->print(f); }
  else
    { fprintf(f,"%d", val); }

};

void IntegerLiteral::eves_print(FILE *f){
  if ( val < 0 )
    { (new Application(new Str("negate"),
       new S_exp_list(new IntegerLiteral(-val))))
       ->print(f); }
  else
    { fprintf(f,"%d", val); }

};
```

```
void IntegerLiteral::pvs_print(FILE *fp){
    fprintf(fp,"%d", val);
};


int IntegerLiteral::equals(S_exp *t){
   if (this == t) return TRUE;
   if (!t->isConstant()) return FALSE;
   if (!((Constant *)t)->isIntegerLiteral()) return FALSE;
   if (val != (((IntegerLiteral *)t)->val)) return FALSE;
   return TRUE;
};


/******** REALLITERAL ********/

void RealLiteral::print(FILE *fp){

  S_exp *e = new IntegerLiteral(si->signed_exp());
  S_exp *i = new IntegerLiteral(si->integer_part());

  (new Application(new Str("decimal"),
   Doubleton(i, e)))->print(fp);

};

void RealLiteral::eves_print(FILE *fp){

  S_exp *e = new IntegerLiteral(si->signed_exp());
  S_exp *i = new IntegerLiteral(si->integer_part());

  (new Application(new Str("decimal"),
   Doubleton(i, e)))->eves_print(fp);

};

void RealLiteral::pvs_print(FILE *fp){

  S_exp *e = new IntegerLiteral(si->signed_exp());
  S_exp *i = new IntegerLiteral(si->integer_part());

  (new Application(new Str("decimal"),
   Doubleton(i, e)))->pvs_print(fp);

};
```

```
int RealLiteral::equals(S_exp *t){

};



/************ MISC S_EXP OPERATIONS ***************/

int S_exp::equals(S_exp *t){
  fprintf(stderr,"erroneous call of S_exp::equals\n");
  print(stdout);
  return FALSE;
};

S_exp *S_exp::And(S_exp *x){
    return new Application(
             new Str("AND"),
             new S_exp_list(this,new S_exp_list(x)));
};

S_exp *S_exp::Implies(S_exp *x){
    return new Application(
             new Str("IMPLIES"),
             new S_exp_list(this,new S_exp_list(x)));
};

S_exp *S_exp::Equals(S_exp *x){
  return new Application(
             new Str("="),
             new S_exp_list(this,new S_exp_list(x)));
};

S_exp *S_exp::Or(S_exp *x){
    return new Application(
             new Str("OR"),
             new S_exp_list(this,new S_exp_list(x)));
};

S_exp *S_exp::Not(){
  return new Application(
             new Str("NOT"),
             new S_exp_list(this));
}

Str *S_exp::PConnName(){
  return new Str("$$error$$");
};
```

```
/******** S_EXP_LIST ********/

void S_exp_list::print(FILE *f) {
  list -> print(f, "\n");
};

void S_exp_list::eves_print(FILE *f) {

  if ( this->isEmpty() )
    { return; }
  else
    { this->head()->eves_print(f); }
  if ( !this->tail()->isEmpty() )
    { fprintf(f," "); (this->tail())->eves_print(f); }

};


void S_exp_list::pvs_print(FILE *f, char *sep) {

  if ( this->isEmpty() )
    { return; }
  else
    { this->head()->pvs_print(f); }
  if ( !this->tail()->isEmpty() )
    { fprintf(f,sep); (this->tail())->pvs_print(f); }

};

void S_exp_list::pvs_print(FILE *f) {
  pvs_print(f, " ");
};

S_exp *S_exp_list::Or(){
   return new Application(new Str("OR"),this);
};

S_exp *S_exp_list::And(){
   return new Application(new Str("AND"),this);
};

S_exp_list *S_exp_list::NegateAll(){

  if (list == NULL)
    { return new S_exp_list; }
```

```
  else
    { if ( list->tl == NULL )
{ return new S_exp_list(this->head()->Not()); }
      else
{ return new S_exp_list(this->head()->Not(),
this->tail()-> NegateAll());}
    }

};
```

12.36 s_exp.h

```
#ifndef __s_exp_h_
#define __s_exp_h_ 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "obj.h"
#include "list.h"
#include "str.h"
#include "scaled_integer.h"

class S_exp {
 public:
  virtual int isConstant() { return FALSE; };
  virtual int isVariable() { return FALSE; };
  virtual int isApplication() { return FALSE; };
  virtual int isRealLiteral() { return FALSE;};

  S_exp *And(S_exp *x);
  S_exp *Implies(S_exp *x);
  S_exp *Or(S_exp *x);
  S_exp *Not();
  S_exp *Equals(S_exp *);

  virtual int equals(S_exp *);
  virtual Str *PConnName();
  virtual void print(FILE *fp) { return; };
  virtual void eves_print(FILE *fp) { return; };
  virtual void pvs_print(FILE *fp) { return; };

// Add the following function:
//
//  virtual Str *like_C() { return; }
```

```
//
// which returns a "C-style string" -- operators
// like "+" and "*" are returned infix; "AND" becomes
// "&&" (multi-adic ANDs are unwound, 0-ary and 1-ary
// become TRUE and no-op, respexly).  It's assumed that
// the input S_exp is one for which such treatment
// makes sense -- only binary uses of binary ops,
// only unary uses of unary ops.

};



class S_exp_list {

 private:
  List<S_exp> *list;
  S_exp_list(List<S_exp> *l) { list = l; };

 public:
  S_exp_list() { list = (List<S_exp> *)0; };
  S_exp_list(S_exp* v) { list =  new List<S_exp>(v); };

  /* Prepends v to the list vl */
  S_exp_list(S_exp* v, S_exp_list* vl)
      { list = new List<S_exp>(v, vl->list);};

  /* List operations */


 S_exp *head() { return list->hd ;};
  S_exp_list *tail() { return new S_exp_list(list->tl); };
  int length() { if (list == NULL) {return 0;} else {return list->len();} };
  int equals(S_exp_list *l2) { list->equals(l2->list); };
  int isEmpty() { return list == NULL ;};

     /* Returns a pointer to "this & l2".  The lists *this and
        *l2 are the same as before, but the returned list
shares things with them. */

  S_exp_list *Append(S_exp_list *l2) {
    return new S_exp_list(list->Append(l2->list));};

  /* Logical operations */

  S_exp *Or();
  S_exp *And();
```

```
  S_exp_list *NegateAll();

  /* Miscellaneous */

  // Produces a newline-separated list of s-exp forms using "native" names

  void print(FILE *);

  // Produces a space-separated list of s-exp forms using EVES names

  void eves_print(FILE *fp);

  // Produces a list of PVS forms separated by *sep

  void pvs_print(FILE *f, char *sep);

  // Produces a space-separated list of PVS forms
  void pvs_print(FILE *fp);

};


class Constant : public S_exp {
 public:
  Constant(){};

  int isConstant() { return TRUE; };
  virtual int isIntegerLiteral() {return FALSE;};
/* We don't need a separate print function -- this
   will dispatch to the literal class  */
};


class IntegerLiteral : public Constant {

 private:
  int val;  /* changed name from value to val */

 public:

  IntegerLiteral(int v) { val = v;};

  /* Accessor */
```

```
  int value() { return val; };

  /* Recognizer */

  int isIntegerLiteral() {return TRUE;};

  /* Overriding virtual functions */

  void print(FILE *f);
  void eves_print(FILE *fp);
  void pvs_print(FILE *fp);
  int equals(S_exp *t);

};

/*****************************************/
/* A lot of stuff about this RealLiteral class
/* is provisional -- in particular, representing
/* them internally as scaled integers
/*****************************************/


class RealLiteral : public Constant {

 private:

  scaled_integer *si;

 public:

  RealLiteral(Str *s) { si = new scaled_integer(s); };
  RealLiteral(scaled_integer *s) { si = s; };

  /* Recognizer */

  int isRealLiteral() {return TRUE;};

  /* Overriding virtual functions */

  void print(FILE *fp);
  void eves_print(FILE *fp);
  void pvs_print(FILE *fp);
  int equals(S_exp *t);

};
```

```
class Variable :  public S_exp {

 private:
  class Str *name;

 public:

  Variable(class Str *n) { name = n; };

  /* Accessor */

  Str *VariableName() {return name;}

  /* Recognizer */
  int isVariable() { return TRUE; };

  /* Overriding virtual functions */

  Str *PConnName() {return name; };
  void print(FILE *f);
  void eves_print(FILE *fp);
  void pvs_print(FILE *fp);
  int equals(S_exp *t);

};

class Application : public S_exp {

 private:

  class Str *fcn;
  S_exp_list *args;

// If *ap is (FOO x1 x2 x3) this returns ((FOO x1 (FOO x2 x3)).
// It's used for turning multi-adic ANDs and ORs into a sequence of
// binary applications.  The boundary cases: (FOO x1) yields x1;
// and, right now, (FOO) yields (FOO) -- but that doesn't really
// make much sense, since (AND) is true and (OR) is false.  The
// proper use of this is therefore to test initially whether the
// list of arguments is empty (in addition to testing whether the
// thing is truly an application).

  S_exp *unwind_binary();

// Auxiliary, "dumb" function -- prints ANDs and ORs as multi-adic
// operations.
```

```
    void pre_pvs_print(FILE *f);

  public:

    Application(class Str *f, S_exp_list *a) {
      fcn = f; args = a; };

    Application(class Str *f) {
      fcn = f; args = new S_exp_list(); };

    /* Accessors */

    Str *function() {return fcn;};
    S_exp_list *arguments() {return args;};

    /* Recognizer */

    int isApplication() { return TRUE; };

    /* Overriding virtual functions */

    Str *PConnName() {return fcn;};

    void print(FILE *f);
    void eves_print(FILE *fp);
    void pvs_print(FILE *fp);
    int equals(S_exp *t);

};

#define EQ_SIGN new Str("=")
#define IMPLIES_SIGN new Str("IMPLIES")

#define false_S_ptr new Application(new Str("false"))
#define true_S_ptr  new Application(new Str("true"))

#define DummyStr new Str("Dummy___")
#define DummyExpression new Application(DummyStr)

// Usage: x, y, z are pointers to S_exp

#define Singleton(x) new S_exp_list(x)

#define Doubleton(x,y) \
  new S_exp_list(x, new S_exp_list(y))
```

```
#define Tripleton(x,y,z) \
  new S_exp_list(x, Doubleton(y,z))

#endif
```

12.37 s_exp_utilities.cc

```
#include "s_exp_utilities.h"
#include "eves2_state.h"

extern eves2_state my_eves2_state;

S_exp_list *state_vals_to_s_exps(StateValueList *svals){

  S_exp_list *slist = new S_exp_list;

  while (svals != NULL){
     state_value sname = svals->head();
     if (sname.kind() != ConditionalState)
        { slist = new S_exp_list(Str_as_S_exp(sname.str(),0),slist); }
     svals = svals->tail();
  }

  return slist;

};

S_exp_list *many_op_one(S_exp_list *many, Str *op, S_exp *one){

  if ( many->isEmpty() )
    { return new S_exp_list; }
  else
    { return new S_exp_list(new Application(op,
     Doubleton(many->head(), one)),
    many_op_one(many->tail(),op,one));
    }

};

S_exp_list *one_op_many(S_exp *one, Str *op, S_exp_list *many){
```

```
  if ( many->isEmpty() )
    { return new S_exp_list; }
  else
    { return new S_exp_list(new Application(op,
     Doubleton(one, many->head())),
    many_op_one(many->tail(),op,one));
    }

};



S_exp *Str_as_S_exp(Str *s, int as_vbl){
  if (as_vbl){
    return new Variable(s);
  }
  else {
    return new Application(s);
  }


};


S_exp_list *ptwise_And(S_exp_list *l1, S_exp_list *l2){
  if (l1->isEmpty())
    { return new S_exp_list; }
  else
    { return new S_exp_list(l1->head()->And(l2->head()),
    ptwise_And(l1->tail(), l2->tail())); }
};

S_exp_list *mutual_exclusion(S_exp_list *siblings){

  /* The guard against a nullpointer may be overkill */

  if ( siblings == NULL || siblings->isEmpty() ||
       siblings->tail()->isEmpty() ) {
    return new S_exp_list;
  }
  else {
    return new S_exp_list(
      siblings->head()->Implies(siblings->tail()->NegateAll()->And()),
      mutual_exclusion(siblings->tail())
    );
  }
```

```
};
```

12.38 s_exp_utilities.h

```
#ifndef __s_exp_utilities_h_
#define __s_exp_utilities_h_ 1

#include "state_value.h"
#include "s_exp.h"

// Return a pointer to the list of formulas saying the [[siblings]]
// are mutually exclusive.  Note: If *siblings is a list of
// length 0 or 1, this returns an empty list.  This makes sense since
// the "and" of an empty list is logically true.

// The particular expression of this property may not be the best for
// all purposes: For example, given (A1, A2, A3), it returns
// (A1 -> (not A2 and not A3), A2 -> (not A3)).  For tablewise-like
// analysis we may want to include variant versions of this -- such as
// pairwise comparison.

S_exp_list *mutual_exclusion(S_exp_list *siblings);

// The function [[state_vals_to_s_exps]] translates a list of [[STATE_NAME]]s
// into the corresponding list of s-expressions, at the same time
// filtering out all the conditional state names.

S_exp_list *state_vals_to_s_exps(StateValueList *svals);

// Str_as_S_exp assumes that s points to an identifier.  The
// parameter "as_vbl" is used to determine whether the resulting
// S-expression will be a variable or a parenthesized identifier.
// This seems to be good enough -- and I don't want to write a parser.

S_exp *Str_as_S_exp(Str *s, int as_vbl);

// If many points to (fml1, fml2, ...) and one to P, then this
// returns a pointer to (fmli->P, fml2->P, ...).  If anything
// in question is empty, it returns a pointer
// to the empty S_exp_list.

S_exp_list *many_imply_one(S_exp *one, S_exp_list *many);

S_exp_list *many_op_one(S_exp_list *many, Str *op, S_exp *one);
```

```
S_exp_list *one_op_many(S_exp *one, Str *op, S_exp_list *many);

// Assumes l1 and l2 are lists of the same length.  If l1 points to
// (P1, P2, ...) and l2 to (Q1, Q2, ...) this returns a pointer to
// (P1&Q1, P2&Q2, ...).  This will dereference a nullpointer if l1 is
// longer than l2.  If l1 is shorter than l2 it will ignore the extra
// places in l2.

S_exp_list *ptwise_And(S_exp_list *l1, S_exp_list *l2);

#endif
```

12.39 simple_list.h

```
#ifndef __simple_list_h__
#define __simple_list_h__ 1

#include <stdio.h>
#include "obj.h"


template <class T>
class SimpleList {

 private:

  T       hd;
  SimpleList<T> *tl;

/* void print() {

    SimpleList<int> *iter = this;
```

```
    printf("list:");
    while (iter != NULL){
      printf(" %d ", iter->head());
      iter = iter->tail();
    }
    printf("\n");
  };
*/

 public:

  /* construct a singleton list */
  SimpleList(T t) : hd(t) {
    tl = NULL;
  };

  /* construct t::l */
  SimpleList(T t, SimpleList<T> *l) : hd(t) {
    tl = l;
  };

  /* accessors */

  T head(){
    return hd;
  }

  SimpleList<T> *tail(){
    return tl;
  }

  /* utilities */

  int len() {if (this==NULL) return 0; else return 1 + tl->len();};

   /* append the given list to this one */
   SimpleList *Append(SimpleList *l2){
    if (this==NULL) { return l2; }
    if ((this->tl)==NULL) { return new SimpleList(this->hd,l2);}
    return new SimpleList(this->hd,this->tl->Append(l2));
    };

  void do_it( void (*op)(T) ){
    if ( this == NULL )
      { return; }
    op(hd);
    tl->do_it(op);
```

```
  };

  SimpleList<T> *pointwise( T (*op)(T) ){
    if ( this == NULL )
      { return NULL; }
    return new SimpleList<T>(op(hd), tl->pointwise(op));
  };

};
#endif
```

12.40 skeleton_utilities.h

```
#ifndef __skeleton_utilities_h_
#define __skeleton_utilities_h_ 1

#include <stdio.h>

// For use in skeleton files---insert the contents of
// the file *filename into file *f.

void grab(char *filename, FILE *f);

#endif
```

12.41 skeleton_utilities.cc

```
#include "skeleton_utilities.h"

void grab(char *filename, FILE *f){

  FILE *text;
  char c;

  text = fopen(filename,"r");
  if (!text)
    { printf("Couldn't open file\n");
      return; }

  while (( c = getc(text)) != EOF) {
```

```
    putc(c,f);
  }

  fclose(text);

};
```

12.42 s_list_table.h

```
#ifndef __s_list_table_h__
#define __s_list_table_h__ 1

/* Hacked up quickie.  (See also name_table.h) */

#include "configuration.h"
#include "s_exp.h"
#include "list.h"
#include "event_value.h"

/* The only good thing about this is the provision of a */
/* "do-it-to-the-whole-table" method. */

class s_list_table {

public:

// Make a formula_table with the given keys, and all associated
// formula lists initially equal to *s.  If the length of *keys is
// greater than MAX_SPEC_NAMES only the first MAX_SPEC_NAMES will be
// entered as keys.

  s_list_table(StrList *keys, S_exp_list *s);

  int size() { return top; };

// Return a pointer to the S_exp_list pointed to by *key (the very same
// pointer that's in the table, so there's sharing here -- which
// does not at the moment seem like a problem).  If none, return
// NULL.

  S_exp_list *find(Str *key);

// Prepend *fml to the entry for *key, if there is such an entry, and
// return 0.  If there is no entry for *key return 1.  Has a side
// effect on *this.
```

```
  int prepend(Str *key, S_exp *fml);

// This runs through the entire table and applies the operation entry_op
// to each (Str*, S_exp_list*) pair.

  void do_it(void (*entry_op)(Str*, S_exp_list*));

  void display(FILE *f);

private:

  typedef struct entry { Str *key; S_exp_list *list; } ENTRY;

// We can't use any of the list templates we have, because they all
// require that the element type have a == operator.

  ENTRY stuff[MAX_SPEC_NAMES];
  int top;

};

#define EmptySListTable s_list_table(EmptyStrList, new S_exp_list);

#endif
```

12.43 s_list_table.cc

```
#include "s_list_table.h"
#include <stdio.h>
#include <string.h>

// Silly, since we're running through the keys twice -- once to count
// them and once to insert them into the table.  But the whole thing
// is so klutzy, who cares?

s_list_table:: s_list_table(StrList *keys, S_exp_list *s){
  if (keys == NULL) {
    top = 0;
  }
  else {
    top = (keys->len() > MAX_SPEC_NAMES)? MAX_SPEC_NAMES : keys->len();
  }

  for (int i = 0; i < top ; i++) {
```

```
    stuff[i].key = keys->hd;
    keys = keys->tl;
    stuff[i].list = s;
  }
};

S_exp_list *s_list_table::find(Str *key){

  if (key == NULL) {return NULL;}

  for (int i = 0; i < top ; i++) {
    if ( stuff[i].key == NULL ) { printf("NULL key in table\n"); return NULL; }
    if ( stuff[i].key->equals(key) )
      {return stuff[i].list;}
  }
  return NULL;
};

int s_list_table::prepend(Str *key, S_exp *fml){

  if (key == NULL) {return NULL;}

  for (int i = 0; i < top ; i++) {
    if ( stuff[i].key == NULL ) { printf("NULL key in table\n"); return NULL; }
    if ( stuff[i].key->equals(key) ) {
      if (stuff[i].list == NULL)
{ stuff[i].list = new S_exp_list(fml); }
      else
{ stuff[i].list = new S_exp_list(fml, stuff[i].list); }
      return 0;}
  }
  return 1;
};

void s_list_table::display(FILE *f){
  printf("******* s_list_table value *******\n");
  for (int i = 0; i < top; i++)  {
    if (stuff[i].key == NULL) {
      printf("Printing null key. Value of i is %d\n",i);
      continue;
    }
    printf("Key: %s\n", stuff[i].key->str);
    printf("Formula list: ");
    if (stuff[i].list == NULL)
      { fprintf(f, "<empty>"); }
    else
      { stuff[i].list->print(f); }
```

```
    printf("\n\n");
  }
};

void s_list_table::do_it(void (*entry_op)(Str*, S_exp_list*)){
  for (int i = 0; i < top; i++)  {
    entry_op(stuff[i].key, stuff[i].list);
  }

};
```

12.44 s_list_table_utilities.h

```
#ifndef __s_list_table_utilities_h__
#define __s_list_table_utilities_h__ 1

#include "str.h"
#include "s_exp.h"
#include "s_list_table.h"

// This returns s_tab_ptr after having a side effect on *s_tab_ptr.
// If *key is a key in *s_tab_ptr, then *fml_ptr is prepended
// to the corresponding formula sequence.  Otherwise, this
// is a no-op.

s_list_table *AddToSTable(Str *key, S_exp *fml_ptr,
  s_list_table *s_tab_ptr);


#endif
```

12.45 s_list_table_utilities.cc

```
#include "s_list_table_utilities.h"

s_list_table *AddToSTable(Str *key, S_exp *fml_ptr,
  s_list_table *s_tab_ptr){

  s_tab_ptr->prepend(key,fml_ptr);
  return s_tab_ptr;

};
```

12.46 spin_state.h

```
#ifndef __spin_state_h_
#define __spin_state_h_

class spin_state {

 public:
  spin_state() { call_number = 0; };

// Successive calls of suffix return "__1", "__2", "__3", etc.

  char *suffix();

 private:

  int call_number;

};


#endif
```

12.47 state_value.h

```
#ifndef __state_value_h_
#define __state_value_h_ 1

#include <stdio.h>
#include "str.h"
#include "simple_list.h"

enum state_kind {AtomicState, AndState, OrState,
   AndArrayState, OrArrayState, ConditionalState};

class state_value {
public:

  state_value(Str *s, state_kind k, int c) {st = s; knd = k; cd = c; };
```

```
  Str *str() {return st;};
  state_kind kind() {return knd;};
  int code() {return cd;};

  void print(FILE *f);

private:
  Str *st;
  state_kind knd;
  int cd;
};

typedef SimpleList<state_value> StateValueList;

#define EmptyStateValueList (StateValueList *)0

//

#endif
```

12.48 state_value.cc

```
#include "state_value.h"

void state_value::print(FILE *f){

  fprintf(f,"%d\t",cd);
  st->print(f);
  fprintf(f,"\t");
  switch (knd) {
    case AtomicState:
      fprintf(f,"Atomic");
      break;
    case AndState:
      fprintf(f,"And");
      break;
    case OrState:
      fprintf(f,"Or");
      break;
    case AndArrayState:
      fprintf(f,"AndArray");
      break;
    case OrArrayState:
      fprintf(f,"Or");
```

```
      break;
    case ConditionalState:
      fprintf(f,"Conditional");
      break;
    }
};
```

12.49 state_value_utilities.h

```
#ifndef __state_value_utilities_h_
#define __state_value_utilities_h_ 1


#include "state_value.h"

StrList *StateValListToStrList(StateValueList*val_list);

// This converts only the unconditional states.

StrList *UnCondStatesToStrList(StateValueList*val_list);

#endif
```

12.50 state_value_utilities.cc

```
#include <stdio.h>
#include "state_value_utilities.h"

StrList *StateValListToStrList(StateValueList*val_list){

  StrList *s_list = EmptyStrList;

  StateValueList *iter = val_list;

  while (iter != NULL){
    s_list = new StrList(iter->head().str(), s_list);
    iter = iter->tail();
  }

  return s_list;
};
```

```
StrList *UnCondStatesToStrList(StateValueList *val_list){

  StrList *s_list = EmptyStrList;

  StateValueList *iter = val_list;

  while (iter != NULL){
    if ( iter->head().kind() != ConditionalState )
      { s_list = new StrList(iter->head().str(), s_list); }
    iter = iter->tail();
  }

  return s_list;

};
```

12.51 str.h

```
#ifndef __str_h__
#define __str_h__ 1

#include <stdio.h>
#include <stdlib.h>
#include "obj.h"
#include "list.h"
class Str : public Obj {

 public:
  Str(char *s) {
    str = (char *)malloc(strlen(s)+1);
    strcpy(str, s);
  };

  char *str;

  int len() { return strlen(str); };

  void print(FILE *f) { fprintf(f,"%s",str); };

  int equals(Str *x) {return !strcmp(str,x->str);};
  int isInfix();
  int isPostfix();
```

```
  Str *append(Str *x)
    {char *dum;

     dum = (char *)(malloc (len() + x->len() + 1));
     strcpy(dum,str);
     strcpy(dum+len(),x->str);
     return new Str(dum);
    };

  Str *append(char *x)
    {char *dum;

     dum = (char *)(malloc (len() + strlen(x) + 1));
     strcpy(dum,str);
     strcpy(dum+len(),x);
     return new Str(dum);
    };

  Str *prepend(char *x)
    {char *dum;

     dum = (char *)(malloc (len() + strlen(x) + 1));
     strcpy(dum,x);
     strcpy(dum+strlen(x),str);
     return new Str(dum);
    };

};

typedef List<Str> StrList;

#define EmptyStrList (StrList *)0

#endif
```

12.52 str.cc

```
#include "str.h"

/* A dummy */
```

12.53 strlist_utilities.h

```
#ifndef __strlist_utilities_h_
#define __strlist_utilities_h_ 1


#include "str.h"

// If slist is (A,B,C) and strop(A) = A', etc.,
// then pointwise(slist,strop) is (A', B', C').
// I'm assuming that strop is "functional" in the
// sense of not changing the abstract value of what
// any existing pointer points to.

StrList *pointwise(StrList *slist, Str* (*strop)(Str*));


#endif
```

12.54 strlist_utilities.cc

```
#include "strlist_utilities.h"

StrList *pointwise(StrList *slist, Str* (*strop)(Str*)){

  if ( slist == NULL )
    { return NULL; }

  return new StrList(strop(slist->hd),
     pointwise(slist->tl,strop));

};
```

12.55 table.h

```
#ifndef __table_h__
#define __table_h__ 1

/* BRAIN DEAD.  Hacked up quickie. */

#include "configuration.h"
```

```
#define TABLE_DATA int

class table {

public:

  table() { top = 0; };
  void insert(char *key, TABLE_DATA *data);
  TABLE_DATA *find(char *key);
  void display();

private:

  typedef struct entry { char *key; TABLE_DATA *data; } ENTRY;

// We can't use any of the list templates we have, because they all
// require that the element type have a == operator.

  ENTRY stuff[MAX_SPEC_NAMES];
  int top;

};


#endif
```

12.56 table.cc

```
#include <string.h>
#include <stdio.h>
#include "table.h"


TABLE_DATA *table::find(char *key){
  for (int i = 0; i < top ; i++) {
    if (strcmp(stuff[i].key, key) == 0)
      {return stuff[i].data;}
  }
  return NULL;
};

void table::insert(char *key, TABLE_DATA *data){
```

```
    if (this->find(key) != NULL)
      { return; }
    else {
      if ( top == 1000 )
{ printf("Table full");
  return;
}
      else
{ stuff[top].data = data;
  stuff[top].key = key;
  top++;
}
    }
};

void table::display(){
  printf("****************\n");
  printf("Table = \n");
  for (int i = 0; i < top; i++) {
    printf("Key: %s, Data: %d\n", stuff[i].key, *(stuff[i].data));
  }
  printf("****************\n");
};
```

12.57 tagged_table.h

```
#ifndef __tagged_table_h__
#define __tagged_table_h__ 1

#include "configuration.h"
#include "s_exp.h"
#include "simple_list.h"

// This stuff should actually be generic in the type of
// the tags -- but the complexity of doing that (at least, the way I
// tried to do it) caused an internal compiler bug in gcc.  Instead,
// I'll just make all the tags into Str's.  The idea is that keys in
// the table will be Str's, and entries in the table will be lists of
// s-exp/tag pairs.

typedef struct tagged_fml { S_exp *fml; Str *tag; } tagged_fml ;

class tagged_s_list {

public:
```

```
// Empty tagged list

  tagged_s_list(){ list = (SimpleList<tagged_fml> *)0; };

// Singleton tagged list

  tagged_s_list(S_exp *f, Str* t) {
    tagged_fml tf;
    tf.fml = f;
    tf.tag = t;
    list = new SimpleList<tagged_fml>(tf);
  };

  tagged_s_list(tagged_fml tf) {
    list = new SimpleList<tagged_fml>(tf);
  };

// Prepend (f,t) to tg_list

  tagged_s_list(S_exp *f, Str* t, tagged_s_list *tg_list)
    { tagged_fml tf;
      tf.fml = f;
      tf.tag = t;
      list = new SimpleList<tagged_fml>(tf, tg_list->list);
    };

  tagged_s_list(tagged_fml tf, tagged_s_list *tg_list){
    list = new SimpleList<tagged_fml>(tf, tg_list->list);
  };

// Accessors

  S_exp *head_fml() { return list->head().fml; };
  Str *head_tag() { return list->head().tag; };
  tagged_s_list *tail() { return new tagged_s_list(list->tail()); };
  int isEmpty() { return list == NULL ;};

  S_exp_list *s_list()
    { if ( list == NULL )
{ return new S_exp_list; }
      else
{ return new S_exp_list(list->head().fml,
tagged_s_list(list->tail()).s_list());}
    };

  void print(FILE *f);
```

```
  void do_it( void (*op)(S_exp*, Str*) );
  void do_it( void (*op)(tagged_fml) );

private:

  SimpleList<tagged_fml> *list;

// This hidden function is used to coerce SimpleList<tagged_fml>
// to tagged_s_list.

  tagged_s_list(SimpleList<tagged_fml>*l) { list = l; };

};

// The table uses Str's as keys to tagged_s_lists's

class tagged_table {

public:

// Make a formula_table with the given keys, and empty entries.

  tagged_table(StrList *keys);

  int size() { return top; };

// Return a pointer to the tagged_list indexed by *key (the very same
// pointer that's in the table, so there's sharing here -- which
// does not at the moment seem like a problem).  If none, return
// NULL.

  tagged_s_list *find(Str *key);

// Prepend (*fml,tag) to the entry for *key, if there is such an entry, and
// return 0.  Has a side effect on *this.  If there is no entry for
// *key, or if the key is null, or if find null key in table, prints a
// warning messages and returns  1.

  int prepend(Str *key, S_exp *fml, Str *tag);

// This runs through the entire table and applies the operation entry_op
// to each (Str*, tagged_s_list*) pair.

  void do_it(void (*entry_op)(Str*, tagged_s_list*));

  void print(FILE *f);
```

```
private:

  typedef struct entry { Str *key; tagged_s_list *list; } ENTRY;

// stuff will actually be allocated as an array of entries
// having size top (a rep invariant)

  ENTRY *stuff;
  int top;

};

#endif
```

12.58 tagged_table.cc

```
#include "tagged_table.h"
#include <stdio.h>
#include <string.h>

void tagged_s_list::do_it( void (*op)(S_exp*, Str*) ){

  if ( list == NULL )
    { return; }
  op(list->head().fml, list->head().tag);
  this->tail()->do_it(op);

};

void tagged_s_list::do_it( void (*op)(tagged_fml) ){
  if ( list == NULL )
    { return; }
  op(list->head());
  this->tail()->do_it(op);

};


void tagged_s_list::print(FILE *f){
  if ( list == NULL )
    { return; };

  fprintf(f, "Tag: ");
  list->head().tag->print(f);
  fprintf(f, "; Fml: ");
  list->head().fml->print(f);
  fprintf(f,"\n");
  this->tail()->print(f);
};

tagged_table::tagged_table(StrList *keys){

  top = keys->len();

  stuff = (ENTRY *)malloc(top * sizeof(ENTRY));

  for (int i = 0; i < top ; i++) {
    stuff[i].key = keys->hd;
    keys = keys->tl;
    stuff[i].list = new tagged_s_list;
  }
```

```
};

tagged_s_list *tagged_table::find(Str *key){

  if (key == NULL)
    { printf("Searching on NULL key.  Returning empty list.\n");
      return new tagged_s_list; }

  for (int i = 0; i < top ; i++) {
    if ( stuff[i].key == NULL )
      { printf("NULL key in table.  Returning empty list.\n");
return new tagged_s_list; }

    if ( stuff[i].key->equals(key) )
      {return stuff[i].list;}
  }

// In this case, the key is not found.
  return new tagged_s_list;
};

int tagged_table::prepend(Str *key, S_exp *fml, Str *t){

  if (key == NULL)
    { printf("Asked to update NULL key.\n");
     return 1 ;}

  for (int i = 0; i < top ; i++) {
    if ( stuff[i].key == NULL )
      { printf("NULL key in table.\n");
return 1; }
    if ( stuff[i].key->equals(key) ) {
      if (stuff[i].list == NULL)
{ stuff[i].list = new tagged_s_list(fml, t); }
      else
{ stuff[i].list = new tagged_s_list(fml, t, stuff[i].list); }
      return 0;
    }
  }
  printf("Key not found.\n");
  return 1;
};


void tagged_table::print(FILE *f){
  printf("******* tagged_table value *******\n");
  for (int i = 0; i < top; i++)  {
```

```
    if (stuff[i].key == NULL) {
      printf("Printing null key. Value of i is %d\n",i);
      continue;
    }
    printf("Key: %s\n", stuff[i].key->str);
    printf("Formula list: \n");
    if (stuff[i].list == NULL)
      { fprintf(f, "<empty>"); }
    else
      { stuff[i].list->print(f); }
    printf("\n\n");
  }
};


void tagged_table::do_it(void (*entry_op)(Str*, tagged_s_list*)){
  for (int i = 0; i < top; i++)  {
    entry_op(stuff[i].key, stuff[i].list);
  }

};

// void tagged_table::do_it( void (*op)(Str*, T) ){
//   list->do_it(promote(op));
// };
```

12.59 transition_value.h

```
#ifndef __transition_value_h_
#define __transition_value_h_ 1

// This is a first approximation.  I don't know what really belongs
// in here.  The only special twist is the "leaves" data.

#include "event_value.h"
#include "str.h"
#include "s_exp.h"

// We keep the names of the actual source and destination
// and their codes (which can be looked up in
// glob).  We should probably do the same for the triggering
// event and the list of actions, but defer that for now.

// This does not automatically maintain the invariant that
// lvs is really what a transition src->dest leaves.  That
```

```
// will have to be guaranteed by the way we generate these things.

// It may not be necessary to maintain the code
// for the transition as part of the transition value; but, by
// analogy with state values we'll do so.

class transition_value {

public:

  transition_value(Str *n, Str *t, S_exp *con,
   int s, Str *s_n,
   int d, Str *d_n,
   int l, Str *l_n,
   StrList *a, int c){
    nm = n; trig = t; condit = con;
    src = s; src_nm = s_n;
    dest = d; dest_nm = d_n;
    lvs = l; lvs_nm = l_n;
    acts = a; cd = c;};

  Str *name() { return nm; }
  Str *trigger_name() { return trig; };
  int source() { return src; };
  Str *source_name() { return src_nm; };
  int destination() { return dest; };
  Str *destination_name() { return dest_nm; };
  int leaves() { return lvs; };
  Str *leaves_name() { return lvs_nm; };
  S_exp *condition() { return condit; };
  StrList *actions() { return acts; };
  int code() { return cd; };

  void print(FILE *f);

private:

  Str *nm;
  Str *trig;
  int src;
  Str *src_nm;
  int dest;
  Str *dest_nm;
  int lvs;
  Str *lvs_nm;
  S_exp *condit;
  StrList *acts;
```

```
  int cd;

};

#define DummyFormula false_S_ptr


#endif
```

12.60 transition_value.cc

```
#include <stdio.h>
#include <stdlib.h>
#include "transition_value.h"

void transition_value::print(FILE *f){
  fprintf(f,"Transition code: %d\n",cd);
  fprintf(f,"Transition name: %s\n",nm->str);
  fprintf(f,"Source (code,name): (%d,%s)\n",src,src_nm->str);
  fprintf(f,"Destination (code,name): (%d,%s)\n",dest,dest_nm->str);
  fprintf(f,"Leaving (code,name): (%d,%s)\n",lvs,lvs_nm->str);
  fprintf(f,"Trigger: ");
  trig->print(f);
  fprintf(f,"\n");
  fprintf(f,"***Not printing condition***\n");
  fprintf(f,"***Acts currently dummied out***\n\n");

};
```

12.61 combinatorics.h

```
#ifndef __combinatorics_h_
#define __combinatorics_h_ 1

// Warshall's algorithm.

void trans_closure(int **relation, int dimension, int **closure);

// Obvious variant of Warshall's algorithm.

void trans_reflex_closure(int **relation, int dimension, int **closure);
```

```
// Assume:
// relation is a square array of size dimension
// relation codes a partial order in which sups exist
// x, y < dimension.

//Return the sup of x and y.

int sup(int **relation, int dimension, int x, int y);

#endif
```

12.62 combinatorics.cc

```
#include <stdio.h>
#include "combinatorics.h"

void trans_closure(int **relation, int dimension, int **closure){

  int i, j, k;

  for (i = 0; i < dimension; i++)
    for (j = 0; j < dimension; j++)
      closure[i][j] = relation[i][j];

  for (k = 0; k < dimension; k++)
    for (i = 0; i < dimension; i++)
      for (j = 0; j < dimension; j++)
if (closure[i][j] == 0)
  { closure[i][j] = closure[i][k] && closure[k][j]; }

};

void trans_reflex_closure(int **relation, int dimension, int **closure){

  int i, j, k;

  for (i = 0; i < dimension; i++)
    for (j = 0; j < dimension; j++)
      closure[i][j] = (i==j? 1 : relation[i][j]);

  for (k = 0; k < dimension; k++)
    for (i = 0; i < dimension; i++)
      for (j = 0; j < dimension; j++)
if (closure[i][j] == 0)
  { closure[i][j] = (closure[i][k] && closure[k][j]); }
```

```
};

int sup(int **relation, int dimension, int x, int y){

  int i = 0;
  int lub_so_far;

// Because of assumption that sups exist, this will
// terminate before running off the end.

  while ( relation[x][i] != 1 || relation[y][i] != 1 )
    { i++; }

  lub_so_far = i;
  i++;

  while ( i < dimension )
    { if (relation[x][i] == 1 &&
  relation[y][i] == 1 &&
  relation[i][lub_so_far])
{ lub_so_far = i; }
      i++; }

  return lub_so_far;

};
```

Chapter 13

The Makefile

```
# $Header: /home/projects2/FMI/EVES/translator/RCS/Makefile,v1.27 1996/04/10 21:24:24 davidg Exp davidg $

CC=/usr/local/bin/g++
LOCAL_CFLAGS= -g
RSML_CFLAGS =

CFLAGS = $(LOCAL_CFLAGS) $(RSML_CFLAGS)

NOWEBFILES =  grammar-skeleton.nw \
grammar-preliminaries.nw \
initialization.nw \
attribute-declarations.nw \
grammar-productions.nw \
attribute-definitions.nw \
eves.nw \
pvs.nw \
spin.nw

# Obsolete: grammar-bodies.nw grammar-traversals.nw
# Obsolete:  eves_state.h eves_state.cc

NOWEB_DOCUMENTATION = \
grammar-skeleton.nw \
grammar-preliminaries.nw \
initialization.nw \
attribute-declarations.nw \
attribute-definitions.nw \
eves.nw \
pvs.nw \
spin.nw
```

```
AUXILIARY_SKELETON_FILES = \
_deterministic_max_enabled_set

GENERATED_SKELETON_FILES = \
_eves_skeleton \
_spin_skeleton \
_pvs_skeleton

EVES_THEORIES = \
real.ver

# Omitting grammar-productions.nw from the "documentation" variable

NOWEB_DOCUMENTATION_FIXED = $(NOWEB_DOCUMENTATION:%=%.fixed)

# The .fixed versions of the noweb files will have all _ inside
# module names escaped with a \

TEST_FILES =  test1 \
test2 \
test3 \
test3-5 \
test4

TEST_TRANSLATIONS = \
ref1 ref2 ref3 \
REF1 REF2 REF3 REF3-5 REF4 \
PVS_REF1 PVS_REF2 PVS_REF3 PVS_REF3-5 PVS_REF4


TEST_PROOFS = proofs1 \
proofs2 \
proofs3

# NOTE: For now I'm not going to be doing anything in the
# Makefile with the INHERITED_CODE -- not even keeping it
# in RCS.  I'll just change all its permissions to make it
# unwritable.

INHERITED_CODE = \
list.h \
obj.h

AUXILIARY_CODE = \
c_style.h \
c_style.cc \
```

```
conditions.h \
conditions.cc \
configuration.h \
event_value.h \
event_value.cc \
event_value_utilities.h \
event_value_utilities.cc \
eves2_state.h \
eves2_state.cc \
formula_table.h \
formula_table.cc \
formula_table_utilities.h \
formula_table_utilities.cc \
function_definition.h \
function_definition.cc \
function_definition_utilities.h \
function_definition_utilities.cc \
global_table.h \
global_table.cc \
load.h \
load.cc \
main.cc \
misc_utilities.cc \
misc_utilities.h \
pvs_state.h \
pvs_state.cc \
pvs_types.h \
pvs_types.cc \
rsml_state.h \
rsml_state.cc \
scaled_integer.h \
scaled_integer.cc \
s_exp.cc \
s_exp.h \
s_exp_utilities.cc \
s_exp_utilities.h \
simple_list.h \
skeleton_utilities.h \
skeleton_utilities.cc \
s_list_table.h \
s_list_table.cc \
s_list_table_utilities.h \
s_list_table_utilities.cc \
spin_state.h \
state_value.h \
state_value.cc \
state_value_utilities.h \
```

```
state_value_utilities.cc \
str.h \
str.cc \
strlist_utilities.h \
strlist_utilities.cc \
table.h \
table.cc \
tagged_table.h \
tagged_table.cc \
transition_value.h \
transition_value.cc \
combinatorics.h \
combinatorics.cc

RCS_CONTROLLED=  \
$(NOWEBFILES) \
$(EVES_THEORIES) \
$(TEST_FILES) \
$(TEST_TRANSLATIONS) \
$(TEST_PROOFS) \
$(AUXILIARY_CODE) \
$(AUXILIARY_SKELETON_FILES) \
$(GENERATED_SKELETON_FILES) \
Makefile \
README \
grammar.l \
kill_nils \
pretty-print \
translate \
stripped-grammar

YACC=/usr/local/src/syn/4.1/syn/etc/berkeley/yacc/yacc
YFLAGS= -d -v

LEX=/usr/local/bin/flex
LFLAGS=-i

OX=ox

OXFLAGS=-I
LDFLAGS=-lfl

MY_OBJS= \
lex.yy.o y.tab.o s_exp.o s_exp_utilities.o \
misc_utilities.o  eves2_state.o load.o main.o table.o \
formula_table.o formula_table_utilities.o event_value.o \
event_value_utilities.o conditions.o pvs_state.o pvs_types.o \
```

```
function_definition.o function_definition_utilities.o \
scaled_integer.o global_table.o state_value.o rsml_state.o \
combinatorics.o transition_value.o s_list_table.o \
state_value_utilities.o s_list_table_utilities.o \
c_style.o strlist_utilities.o skeleton_utilities.o


OBJS =  $(MY_OBJS)

translator: $(OBJS)
$(CC) -o translator $(CFLAGS) $(OBJS) $(LDFLAGS)

grammar.cc: $(NOWEBFILES) generated_include_file.h \
generated_C_macros.h $(GENERATED_SKELETON_FILES)
notangle -L"/* line %L of %F */%N" $(NOWEBFILES) > grammar.cc

oxout.y oxout.l: grammar.cc grammar.l
$(OX) $(OXFLAGS) grammar.cc grammar.l

dont_touch_this.y: oxout.y
cat oxout.y | cpif dont_touch_this.y

y.tab.c y.tab.h: dont_touch_this.y
$(YACC) $(YFLAGS)  dont_touch_this.y

checkout:
co $(RCS_CONTROLLED)

checkin: $(RCS_CONTROLLED)
ci $(RCS_CONTROLLED)

weak_checkin: $(RCS_CONTROLLED)
ci -u $(RCS_CONTROLLED)

unlocked:
rcs -u $(RCS_CONTROLLED)

clean:
rm $(MY_OBJS) translator grammar.cc oxout.y oxout.l \
y.tab.c y.tab.h lex.yy.c dont_touch_this.y \
generated_include_file.h generated_C_macros.h \


cleano:
rm $(MY_OBJS)

%.fixed: %
```

```
cat $< | generating-skeletons/fix_module_names> $@

documentation: $(NOWEB_DOCUMENTATION_FIXED) $(AUXILIARY_CODE) Makefile \
stripped-grammar grammar.l
noweave -l -n $(NOWEB_DOCUMENTATION_FIXED) > docs/grammar-cc.tex
cat docs/beginverbatim Makefile docs/endverbatim \
> docs/Makefile.tex
cat docs/beginverbatim stripped-grammar docs/endverbatim \
> docs/stripped-grammar.tex
cat docs/beginverbatim grammar.l docs/endverbatim \
> docs/grammar-l.tex
rm -f docs/aux-code.tex
for f in $(AUXILIARY_CODE); \
do \
title=`echo $$f | sed "s/_/\\\\\\_/g"` ; \
label=`echo $$f | sed -e "s/_/-/g" \
      -e "s/\./-/g"` ; \
echo "\section{" $$title "}\label{" $$label "}" >> \
docs/aux-code.tex; \
cat docs/beginverbatim >> docs/aux-code.tex; \
cat $$f >> docs/aux-code.tex ; \
cat docs/endverbatim >> docs/aux-code.tex; \
done;

# This used to depend on opts.h, but now that stuff
# has been put into grammar-skeleton.nw, so into grammar.cc

_eves_skeleton: $(NOWEBFILES)
notangle -R"eves skeleton" $(NOWEBFILES) | \
cpif _eves_skeleton

_spin_skeleton: $(NOWEBFILES)
notangle -R"spin skeleton" $(NOWEBFILES) | \
cpif _spin_skeleton

_pvs_skeleton: $(NOWEBFILES)
notangle -R"pvs skeleton" $(NOWEBFILES) | \
cpif _pvs_skeleton

generated_include_file.h : $(NOWEBFILES)
notangle -R"Include files" $(NOWEBFILES) | \
cpif generated_include_file.h

generated_C_macros.h : $(NOWEBFILES)
notangle -R"C Macros" $(NOWEBFILES) | \
cpif generated_C_macros.h
```

```
y.tab.o: y.tab.c
$(CC) -c $(CFLAGS) y.tab.c

lex.yy.c: oxout.l
$(LEX) $(LFLAGS) oxout.l

lex.yy.o: y.tab.h lex.yy.c
$(CC) -c $(CFLAGS) lex.yy.c

main.o: main.cc
$(CC) -c $(CFLAGS)  main.cc

# This should now be obsolete
# opts.o: opts.cc
#  $(CC) -c $(CFLAGS)  opts.cc

scaled_integer.o: scaled_integer.h scaled_integer.cc
$(CC) -c $(CFLAGS)  scaled_integer.cc

state_value.o: state_value.h state_value.cc
$(CC) -c $(CFLAGS)  state_value.cc

state_value_utilities.o: state_value_utilities.h state_value_utilities.cc
$(CC) -c $(CFLAGS)  state_value_utilities.cc

s_exp.o: s_exp.h s_exp.cc
$(CC) -c $(CFLAGS)  s_exp.cc

s_exp_utilities.o: s_exp_utilities.h s_exp_utilities.cc
$(CC) -c $(CFLAGS)  s_exp_utilities.cc

str.o: str.h str.cc
$(CC) -c $(CFLAGS)  str.cc

misc_utilities.o: misc_utilities.h misc_utilities.cc
$(CC) -c $(CFLAGS)  misc_utilities.cc

eves2_state.o: eves2_state.h eves2_state.cc misc_utilities.h
$(CC) -c $(CFLAGS)  eves2_state.cc

global_table.o: global_table.h global_table.cc
$(CC) -c $(CFLAGS)  global_table.cc

rsml_state.o: rsml_state.h rsml_state.cc simple_list.h
$(CC) -c $(CFLAGS)  rsml_state.cc

load.o: load.h load.cc
```

```
$(CC) -c $(CFLAGS)  load.cc

table.o: table.h table.cc
$(CC) -c $(CFLAGS)  table.cc

formula_table.o: formula_table.h formula_table.cc
 $(CC) -c $(CFLAGS)  formula_table.cc

formula_table_utilities.o: formula_table_utilities.h formula_table_utilities.cc
 $(CC) -c $(CFLAGS)  formula_table_utilities.cc

function_definition.o: function_definition.h function_definition.cc
 $(CC) -c $(CFLAGS)  function_definition.cc

function_definition_utilities.o: function_definition_utilities.h function_definition_utilities.cc
 $(CC) -c $(CFLAGS)  function_definition_utilities.cc


event_value.o: event_value.h event_value.cc generated_C_macros.h
 $(CC) -c $(CFLAGS) event_value.cc

event_value_utilities.o: event_value_utilities.h event_value_utilities.cc
 $(CC) -c $(CFLAGS) event_value_utilities.cc

conditions.o: conditions.h conditions.cc
 $(CC) -c $(CFLAGS) conditions.cc

pvs_state.o: pvs_state.h pvs_state.cc
$(CC) -c $(CFLAGS)  pvs_state.cc

pvs_types.o: pvs_types.h pvs_types.cc
$(CC) -c $(CFLAGS)  pvs_types.cc

combinatorics.o: combinatorics.h combinatorics.cc
$(CC) -c $(CFLAGS)  combinatorics.cc

transition_value.o: transition_value.h transition_value.cc
$(CC) -c $(CFLAGS)  transition_value.cc

skeleton_utilities.o: skeleton_utilities.h skeleton_utilities.cc
$(CC) -c $(CFLAGS)  skeleton_utilities.cc

s_list_table.o: s_list_table.h s_list_table.cc
$(CC) -c $(CFLAGS)  s_list_table.cc

s_list_table_utilities.o: s_list_table_utilities.h s_list_table_utilities.cc
$(CC) -c $(CFLAGS)  s_list_table_utilities.cc
```

```
tagged_table.o: tagged_table.h tagged_table.cc
$(CC) -c $(CFLAGS)  tagged_table.cc

c_style.o: c_style.h c_style.cc
$(CC) -c $(CFLAGS)  c_style.cc

strlist_utilities.o: strlist_utilities.h strlist_utilities.cc
$(CC) -c $(CFLAGS)  strlist_utilities.cc

test_scaled_integer.o: test.cc
$(CC) -c $(CFLAGS)  test_scaled_integer.cc

test_scaled_integer: test_scaled_integer.cc test_scaled_integer.o \
scaled_integer.o s_exp.o
$(CC) -o test_program $(CFLAGS) test_scaled_integer.o \
scaled_integer.o s_exp.o

test_warshall.o: test_warshall.cc
$(CC) -c $(CFLAGS)  test_warshall.cc

test_warshall: test_warshall.cc test_warshall.o
$(CC) -o test_warshall $(CFLAGS) test_warshall.o

test_tagged_table.o: test_tagged_table.cc
$(CC) -c $(CFLAGS)  test_tagged_table.cc

test_tagged_table: test_tagged_table.cc test_tagged_table.o
$(CC) -o test_tagged_table $(CFLAGS) test_tagged_table.o  \
tagged_table.o str.o s_exp.o scaled_integer.o

# DO NOT DELETE THIS LINE -- make depend depends on it.
```

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
12 April 1996

3. REPORT TYPE AND DATES COVERED
Final Report 18-Oct-95 to 12 APR-96

4. TITLE AND SUBTITLE
Cooperating Formal Methods
Prototype Code

5. FUNDING NUMBERS
N00014-95-C-0349

6. AUTHOR(S)
David Guaspari

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Odyssey Research Associates, Icn.
301 Dates Drive
Ithaca, NY 14850-1326

8. PERFORMING ORGANIZATION REPORT NUMBER
ORA-TM-96-0019
(Attachment A)

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
A - Unlimited release

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The prototype formal methods interface (FMI) can easily be linked to a variety of formal tools. This is achieved by representing the underlying specification language (RSML) in an attribute grammar and predefining a rich set of attributes that capture RSML semantics. A developer wishing to model aspects of RSML within a formal tool can access these predefined attributes directly, and may also add an arbitrary collection of new attributes. Editable WEB-style templates simplify this task. This attachment demonstrates the usefulness of our techniques by showing how the EVES and PVS theorem provers and the SPIN model checker were rapidly linked to the FMI.

14. SUBJECT TERMS
formal methods, formal specification, automated theorem proving, model checking

15. NUMBER OF PAGES
237

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	None

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102