The TLV Manual

Elad Shahar
elad@weizmann.ac.il

Weizmann Institute

Additional Material: Ittai Balaban
balaban@cs.nyu.edu

New York University

March 26, 2005

Contents

1 Introduction **2**
1.1 TLV-BASIC . . . 2
1.2 Notation . . . 3
1.3 Variable Ordering . . . 3
1.4 Beginning to work with TLV . . . 3
1.5 Command line options . . . 5
1.6 New in TLV 4.0 . . . 7

I Input Languages of TLV **8**

2 TLV-BASIC **9**
2.1 Introduction . . . 9
2.1.1 System Variables and Dynamic Variables . . . 9
2.1.2 Creating and Deleting Variables . . . 9
2.1.3 Very Basic Commands . . . 10
2.2 Examples . . . 11
2.2.1 Using TLV-BASIC for proof scripts . . . 11
2.2.2 Using TLV-BASIC for verification rules . . . 12
2.3 Data types . . . 13
2.4 Predefined Dynamic Variables . . . 14
2.5 Expressions . . . 15
2.5.1 Logical operators . . . 16
2.5.2 Numeric operators . . . 16
2.5.3 Conditional Expressions: a ? b : c and `case` . . . 16
2.5.4 Checking for validity, contradictions and equality . . . 17
2.5.5 Value sets . . . 18
2.5.6 Set operators . . . 18
2.5.7 Variable sets . . . 18
2.5.8 Satisfing assignments . . . 20
2.5.9 Constructor loop expressions . . . 20
2.5.10 next, prime and unprime . . . 21
2.5.11 Systems and Transitions: sys_num, succ, pred . . . 21
2.5.12 Assigning and Obtaining System Variable Values . . . 22

2.5.13 Other operators: size, exist . . . 22
2.6 Basic Commands: Let, Print, Load and Quit . . . 22
2.7 Procedures and Functions . . . 23
2.7.1 Default Values . . . 24
2.7.2 Local Variables . . . 25
2.7.3 Parse Tree Parametrs . . . 26
2.7.4 By-name Parameters . . . 26
2.8 Compound Statements . . . 27
2.8.1 While and If . . . 27
2.8.2 Boolean Conditions . . . 27
2.8.3 For and Fix: additional loop statements . . . 28
2.9 Parse Tree Commands . . . 29
2.10 Status Commands . . . 32
2.10.1 Performance . . . 32
2.10.2 Memory Use and Garbage Collection . . . 33
2.11 Reordering commands . . . 33
2.12 Input Commands: read_num, read_string . . . 33
2.13 Output Commands . . . 35
2.13.1 Logging Output . . . 35
2.13.2 Textual Output of OBDDs . . . 35
2.13.3 Dotty Output of OBDDs . . . 37
2.14 Records . . . 38
2.15 Pitfalls . . . 38
2.16 Parsing Temporal Logic . . . 39
2.16.1 Propositional linear Temporal Logic (PTL) . . . 40
2.16.2 Propositional branching Temporal Logic (CTL) . . . 42

3 The SMV Input Language **43**
3.1 Introduction . . . 43
3.2 Clarifications . . . 43
3.2.1 Asynchronous Composition . . . 43
3.2.2 Integer Range Types in Variable Declaration . . . 45
3.2.3 case Expressions . . . 45
3.2.4 Scope of Module Parameters . . . 45
3.2.5 Modules as Records . . . 46
3.2.6 Translating ASSIGN to TRANS, and vice versa . . . 46
3.3 Changes to the SMV input language . . . 46
3.3.1 FAIRNESS . . . 46
3.3.2 Systems . . . 47
3.3.3 kind of . . . 47
3.3.4 COMPOSED . . . 47
3.3.5 OWNED . . . 49
3.3.6 Using system variables as indexes of arrays . . . 49
3.3.7 Constructor Loops . . . 50
3.3.8 Conditional Statements . . . 51
3.3.9 default in . . . 51

4 The SPL Input Language **52**
4.1 Overview . . . 52
4.2 Example — Peterson's Algorithm for Mutual Exculsion . . . 52
4.3 The Implementation of SPL in TLV . . . 55

II TLV Modules **56**

5 Utilities **57**

6 Array Utilities **58**

7 Transition System Module (TS) **60**
7.1 Types of Transition Systems . . . 60
7.2 Initialization . . . 61
7.3 The Current Transition System . . . 61
7.4 Obtain Number of Systems . . . 61
7.5 Creating Transition Systems . . . 62
7.6 Manipulating Components of Transition Systems . . . 62
7.6.1 I, V . . . 62
7.6.2 T . . . 63
7.6.3 Fairness . . . 63
7.6.4 Owned . . . 64
7.7 Operations on Transition systems . . . 64
7.8 Successors and Predecessors . . . 65
7.9 Refute . . . 65
7.10 Restriction . . . 66

8 Simulations **67**
8.1 Creating . . . 67
8.2 Printing . . . 68
8.3 Annotation with Temporal Formula . . . 69

9 Simple Model Checking **71**
9.1 Deadlocks . . . 71
9.2 Invariance . . . 72
9.3 Response . . . 73

10 Model Checking **74**
10.1 Automata as Specification . . . 74
10.2 Propositional Linear Temporal Logic (LTL) . . . 75
10.2.1 Validity and Satisfiability . . . 75
10.2.2 Model Checking . . . 76
10.3 CTL . . . 77
10.4 CTL* . . . 77
10.4.1 Fair Computation Structures . . . 77
10.4.2 The Logic CTL* . . . 78

10.4.3 Model Checking . . . 79

11 Deductive Rules **80**
11.1 Invariance . . . 80
11.2 Response . . . 81

12 Floyd's Method **82**

13 Abstraction **83**

Bibliography **83**

Index **84**

III Appendixes **88**

A SPL: Syntax **89**
A.1 Simple Statements . . . 89
A.2 Schematic Statements . . . 90
A.3 Compound Statements . . . 91
A.4 Programs . . . 92

B SPL: Semantics **94**
B.1 System Variables . . . 94
B.2 Initial Condition . . . 94
B.3 Transitions . . . 95
B.4 The Justice Set . . . 100
B.5 The Compassion Set . . . 100

C Error Messages of the SPL Compiler **101**

Chapter 1

Introduction

TLV is a flexible environment for verification of finite state systems. TLV (Temporal Logic Verifier) reads programs written in the SMV input language [1] [2] or in SPL [3], translates them to OBDDs [4] and then enters an interactive mode where OBDDs can be manipulated. The interactive mode includes a high level programming language, called TLV-BASIC, which also understands SMV expressions.

In contrast to the SMV system, all proofs in TLV are done interactively. Verification is performed by running procedures written in TLV-BASIC. By writing programs in TLV-BASIC, TLV can be extended to support both deductive and algorithmic verfication.

The (TLV) system described here has been constructed on top of the CMU SMV system which supports verification of CTL specifications of finite-state systems. This support has been removed. However, it is possible to re-implement most of it within the TLV environment.

1.1 TLV-BASIC

TLV-BASICis named so because like the BASIC programming language, it accepts the same syntax for user commands (which are executed immediately) and for programs that may be prepared in a file off-line and invoked by a command. TLV-BASIC is a structured language, containing no *go-to*'s but using *while* and conditional statements.

The TLV-BASIC language is used for three purposes:

- Temporal verification rules, such as the basic invariance rule BINV, as well as algorithms for model-checking, are written as TLV-BASIC procedures. There are standard rules file containing these procedures and other utilities, such as simulation of steps of the system, which are automatically loaded when the system is started. However, the user may modify the rules, or prepare new ones.
- For each particular system to be verified, the user usually prepares a *proof script* file which contains definitions of the assertions used in order to verify the required property.
- The interactive dialog with the user is carried out in TLV-BASIC. It lets the user manipulate OBDDs, load files, define and run procedures.

Expressions in TLV-BASIC are quantifier-free assertion, obeying the SMV syntax for state-formulas, and represented internally by an OBDD.

You can view OBDDs as sets, functions or expressions in propositional logic - depending on the context.

1.2 Notation

The prompt of TLV is '>>', therefore, anything which is on a line which starts with '>>' has been entered by the user. Lines which do not start with one of these prompts have been printed by TLV.

1.3 Variable Ordering

The following section is mostly copied from the SMV man page:

TLV uses Boolean Decision Diagrams (BDDs) to represent functions, transitions and state valuations. A BDD is a decision tree, in which variables always appear in the same order as the tree is traversed from root to leaf.

The efficiency of BDDs is obtained by always combining isomorphic subtrees, and by eliminating redundant decision nodes in the tree. The degree storage efficiency obtained in this way is closely related to the variable ordering. The present version of the program has no built-in heuristics for selecting the variable ordering. Instead, the variables appear in the BDDs in the same order in which they are declared in the program. This means that variables declared in the same module are grouped together, which is generally a good practice, but this alone is not generally sufficient to obtain good performance in the evaluation.

Usually, the variable ordering must be adjusted by hand, in an ad hoc way. A good heuristic is to arrange the ordering so that variables which often appear close to each other in formulas are close together in the order of declaration, and global variables should appear first in the program. The number of BDD nodes currently in use is printed on standard error each time the program performs garbage collection, if verbose-level is greater than zero. An important number to look at is the number of BDD nodes representing the transition relation. It is very important to minimize this number.

TLV uses a simple automatic variable reordering algorithm (adopted from smv2.4.4) which can be used if you do not want to bother to reorder the variables by hand. It goes over all variables of the program, tries to put each variable in all possible positions and finds the position which minimizes the total number of bdd nodes. The variable is then moved to that location.

1.4 Beginning to work with TLV

A schematic view of the interaction with TLV is presented in Fig. 1.1. TLV first reads a system written in the SPL or SMV languages. If the system is written in SPL then TLV internally invokes a compiler which translates the file to the SMV language, and then reads the translated file.

Next, TLV attempts to load a rules file which defines standard procedures and verification rules using TLV-BASIC.

If a proof file is specified in the command line then it is loaded. A proof file contains further definitions and rules in TLV-BASIC.

At this point, the user takes control The user can enter TLV commands interactively. The user can also load proof files, execute internal TLV commands, or run the various routines which were defined previously in the rules file or any proof file which was loaded.

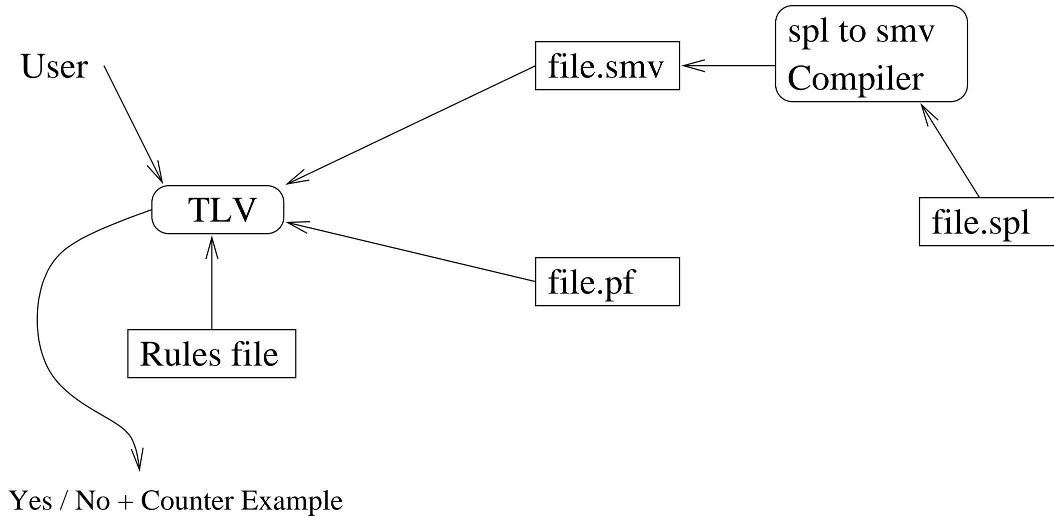

Figure 1.1: Working with TLV

To work properly, TLV has to know the directory where the default rules file is. In order to do that put the following in your .cshrc file

```
setenv TLV_PATH ~elad/TLV/
```

This variable is also used to determine the location of other TLV-BASIC files and example files in the SMV or SPL languages, which come with the system.

If you plan to work with SPL then you should also add this directory to your path variable, since it also contains the SPL compiler.

The file "Rules.tlv" is the default rules file which is loaded if if the environment variable `TLV_RULES` is not set. If the current directory from which you are running tlv contains a file named Rules.tlv, then it will be loaded, and not the one in `$TLV_PATH` . This way you can override the default rules file. Setting the `TLV_RULES` environment variable will cause that file to be loaded instead of the Rules.tlv . Again, the current directory is always searched first.

For example, the following will cause TLV to first search for the file Fast_Rules.tlv in the current directory, and the in `~elad/TLV/` :

```
setenv TLV_PATH ~elad/TLV/
setenv TLV_RULES Fast_Rules.tlv
```

The following are some pointers for the TLV novice:

- Exiting TLV can be done with the `<CTRL-D>` key.
- Many keywords start with a capital letter. Letter case **is** important.
- Sometimes after TLV prints an error message, you will not get a prompt. TLV has a problem with parsing. When an error occurs, recovery is not always immediate. However it is always possible to reach to the prompt if you enter enough "END" or ";" characters which will get TLV out of some internal erronous parsing states.

  In general, you can execute new commands only when you get the `>>` prompt.
- TLV uses gnu readline, so the arrow and delete keys work as one would expect. Furthermore, using the up and down arrows switches between all previously entered commands.

1.5 Command line options

The command for running TLV is:

```
% tlv [options] [autoload-file] input-file
```

If the autoload-file parameter exists it is loaded just before entering interactive mode. This file should contain code in tlv-basic.

The input-file is a program written in the SMV input language (with suffix ".smv") or in SPL (with suffix ".spl") .

After the files are read the program enters interactive mode. TLV does not run the program. It asks the user for further instructions. This is indicated by the prompt `>>` .

Note that once you are in TLV you cannot load a new file in the smv input language or in SPL. Only programs in TLV-BASIC can be loaded. If you want to verify a new program then you should exit TLV and rerun it with the file name of the new smv program as a command line parameter.

The command line options are:

- **-i** input-order
  The variable ordering is read from file input-order . This is most useful in combination with the **-o** option: The variable ordering can be written to a file using the -o option, the file can be inspected and reordered by the user, then read back in using the -i option. See *variable ordering*.
- **-o** output-order
  The variable ordering is written to file output-order, after parsing. No evaluation occurs when the -o option is used.
- **-v** verbose-level
  A large amount of gibberish printed on the standard error. Setting verbose-level to 1 should give you all the information you need. Setting the verbose-level higher than 2 has the same affect as 2.
- **-reorder** dynamic-variable-reordering
  The dynamic variable reordering algorithm will work with this option. Every time when the garbage collection routine is called with the total BDD size large enough, *dynamic-reordering* tries to change the variable order in order to reduce the total bdd node number.

- **-reorderbits** bits-for-dynamic-variable-reordering
  This option gives the limit for the number of bits of the variable to be reordered. The reorder routine will skip the variables that exceeds this limit. The default value is 10.

- **-reordersize** starting-size-for-dynamic-variable-reordering
  This option gives the minimal total bdd node number that the reorder routine will start working. Current default value is 5000.

- **-gc** parameter
  do garbage collection when number of allocated is above parameter, To force garbage collection all the time use -gc 0 .

- **-disj n**
  Determines the amount of transitions which are generated per system. The total transition relation of the system is the disjunction of all the individual transitions. This command option changes the size of the arrays _t[], _d[] and _pres[]. More transitions means that each transition is smaller, and this can have a signficant impact on performance.

  The parameter, n, can be 0, 1, or 2. When n = 0, a single transition is generated per system. When n = 1, (the default) the transitions of modules which have interleaved execution (processes) are exported to the interactive environment as separate transitions. There will be one transition for each process. Thus, in the `mux-sem.smv` example, the system will generate two transitions, one corresponding to each process.

  When n = 2, the transitions are generated differently, such that there may be more transitions overall. If there are several synchronous modules, each of which having processes, then a transition is generated by taking one process from each of the synchronous modules.

- **-gui**
  This flag is intended to be given only by a gui which wraps TLV. The intent is to implement a gui using expectTk.

  There is a TLV command called “gui”. Invoking this command, followed by a tcltk command string as a parameter would print the command in such a way that the gui would detect the commands and attempt to execute them. This way, the user can execute commands, or write procedures in TLV-BASIC which change the gui.

  Without the flag, the command “gui” will not output anything to the screen. With the flag, the command will output the tcltk commands such that the gui will detect them and execute them. This way, if you write a procedure with gui commands, you can run the procedure, even without the gui, and the gui command will not produce clutter on your screen.

- **-c cache-size**
  Set the size of the cache for BDD operations, such as OR, AND, XOR, quantification, succ and pred. There is a tradeoff here between performance and memory. Up to a point, a larger cache can speed up operations by orders of magnitude. Each cache entry uses 16 bytes, so a quarter million entries use about four megabytes, which is reasonable if you have about 12 megabytes of real memory available. Virtual memory is of practically no use.

  The size should be a prime or nearly prime number. Default is 522713. Some suggested values for the -c parameter: 16381, 65521, 262063, 522713, 1046429

- **-m mini-cache-size**
  Sets the size of the portion of the cache for BDD operations which is used by the less expensive (non- iterative) BDD operations. It should be a prime or nearly prime number not larger than the cache-size. The default is 522713, same as the default cache-size.

- **-k key-table-size**
  Set the size of the key table for BDD nodes. It should be a prime or nearly prime number, and should be at least 1/10 the number of BDD nodes in use at any given time. The default is 126727, which should be enough for most applications.

1.6 New in TLV 4.0

The following have been added to TLV 4.0 :

- Documentation of command line options for bdd cache sizes. See above.
- The conditional expression `a?b:c` , from the C programming language. See section 2.5.2.
- It is possible to use a program variable as an index to an array, including assignments. See section 3.3.6.
- Constructor loops, similar to Cadence SMV (sections 2.5.9, 3.3.4, 3.3.7). There are also constructor loops for building expressions.
- Conditional statements (section 3.3.8).
- The `default ... in ...` construct, also similar to Cadence SMV (section 3.3.9).
- Functions is_true, is_false and is_equal for easier testing in TLV-Basic (section 2.5.4).
- TLV-Basic looping constructs, For and Fix (section 2.8.3).
- Routines for manipulating arrays (section 6).
- Default parameters can be specified for TLV-Basic functions and procedures.
- It is now possible to refer to the formal parameters of modules, from the interactive environment. For example, in the following program, you can refer to the name `a.c` in the interactive environment, and it will be equal to `b`.

```
MODULE main
VAR
  b : boolean;
  a : user(b);

MODULE user(c)
ASSIGN
  next(c) := !next(c);
```

See the appropriate sections for more information.

Part I

Input Languages of TLV

Chapter 2

TLV-BASIC

2.1 Introduction

The TLV-BASIC language is used to program rules, model-checking algorithms, and compute assertions. The main (and only) data structure is a function with boolean arguments and integer range. As such, it can represent integers, booleans (a function with range $\{0, 1\}$), assertions, transitions, state valuations and sets of variables which are represented as boolean functions. The underlying implementation is an OBDD, which is manipulated using the SMV OBDD library. Expressions in the language are constructed out of integer constants and variables to which we apply integer operations, integer comparisons, and all the boolean and quantifying operators available in the SMV language.

2.1.1 System Variables and Dynamic Variables

The TLV system distinguishes between two types of variables. The *system variables* are the variables defined in the SMV or SPL input files. They represent the system variables for the program or circuit to be verified. For example, after reading file `mux-sem.smv` of Fig. 3.1, TLV recognizes the system variables `y`, `proc[1].loc` and `proc[2].loc`. In fact we have two copies of the system variables in order to be able to express the transition relations as assertions on two copies of the system variables. The second copy of each variable can be referred to by enclosing it with next(). For example, an expression stating that the variable `y` preserves its value in the next state can be written as : `y = next(y)`.

The other type of variables are *dynamic variables*, which the TLV system manipulates and assigns. We are interested in expressing functions, assertions and relations where the domain is the state space of the program. These functions, assertions and relations are stored in dynamic variables. Dynamic variables can be viewed as containers of finite state expressions over the system variables.

2.1.2 Creating and Deleting Variables

Dynamic variables are created mainly by assigning values them using the "Let" command. No TLV statement should assign values to a system variable.

Assume, for example, that after reading the file `mux-sem.smv` into TLV, the user issues the TLV command

```
>> Let x := y = 1 ;
```

This statement assigns to the dynamic variable `x` the assertion `y = 1`. Thus, references to system variables within TLV-expression are always interpreted symbolically. That is, the system retains `y = 1` as a symbolic assertion and does not attempt to evaluate it.

Dynamic variables are not declared like system variables. As in BASIC, dynamic variables are created, whenever they are assigned values, or mentioned as parameters of a procedure. The transition system is also represented as arrays of such dynamic variables. Even arrays of dynamic variables do not need to be declared in advance, so the assignment

```
>> Let a[1] := x /\ y;
```

Puts the propositional expression $x \wedge y$ in the dynamic variable `a[1]` .

Note that dynamic variables can be used in other expressions as well. For example the following will compute the conjunction of x,y and z:

```
>> Let a[2] := a[1] /\ z;
```

System variables are created mostly as a result of loading a file in the input language of SMV. The user can add additonal variables with the command:

```
term : type ;
```

This command creates a new system variable of the requested type. The term can be an atom or atom[index]. Type can be either boolean, or a value set of the form: { val1 , val2 , ... , valn } .

Such variables cannot be assigned by the "Let" command.

Dynamic variables can be deleted using the command `delvar`. For example:

```
>> Let k := 5;
>> Print k;
5
>> delvar k;
>> Print k;
k undefined
```

2.1.3 Very Basic Commands

Following are some of the statements available in TLV-BASIC:

- `Let` *var* `:=` *exp* — Assign the value of expression *exp* to variable *var*.
- `Call` *proc-name* ($par_1, \ldots, par_n$) — Invoke procedure *proc-name* with the given actual parameters.
- `Proc` *proc-name* ($par_1, \ldots, par_n$)`;` S `End` — Define a procedure *proc-name* with parameters $par_1, \ldots, par_n$ and body S. Paramteres are transferred by value.

```
Let n := 3;

Proc prepare();
  Let mux := TRUE;
  Let i := n;
  While (i)
    Let j := i - 1;
    While (j)
      Let mux := mux & !(proc[i].loc = 3 & proc[j].loc = 3);
      Let j := j - 1;
    End -- end loop on j
    Let i := i - 1;
  End -- end loop on i
End -- end procedure

Call prepare();
```

Figure 2.1: File `mux-sem.pf`: Proof Script of mutual exclusion for general n.

- `While (`*exp*`)` S `End` — Repeatedly execute statements S until *exp* becomes 0.
- `If (`*exp*`)` S_1 $\left[\ \texttt{else}\ S_2\right]$ `End` — If *exp* evaluates to a non-zero value, execute statements S_1. Otherwise, execute statements S_2. The `else` clause is optional.

The first two statements are the main commands that are used in an interactive mode.

2.2 Examples

An SMV version of mutual exclution with semaphores was presented in Fig. 3.1 and Peterson's algorithm for mutual exclusion, was presented in Fig. 4.1.

2.2.1 Using TLV-Basic for proof scripts

To illustrate the use of TLV-Basic for forming proof scripts, consider the family of programs that can be obtained from program MUX-SEM, by considering more than 2 processes. For example, to obtain a version of this program for 3 processes, it is only necessary to add the declaration

`proc[3] :  process user(y);`

after the declaration of `proc[2]`, and add entries for `proc[3]` to the justice and compassion lists. However, in parallel with the modification of the program, it is also necessary to modify the assertion which specifies mutual exclusion for all processes in the system. For the general case of $n > 1$ processes, this assertion can be written as

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{i-1} \neg(\texttt{proc}[\texttt{i}].\texttt{loc} = 3 \ \&\ \texttt{proc}[\texttt{j}].\texttt{loc} = 3).$$

which is computed in the file `mux-sem.pf` as presented in Fig. 2.1 (although this can be computed more succinctly using constructor loops). When we consider the same program for a different number of processes, say 4, it is only necessary to change the first statement in this file to `Let n := 4`.

2.2.2 Using TLV-BASIC for verification rules

Verification rules can also be written in TLV-BASIC. TLV comes with a large collection of rules, but you can program your own. The following is an example of a deductive rule which checks that for invariance:

```
-- Procedure which checks if the paramter p_ is an invariant.
Proc binv(p_);

  -- Check B1.

  -- Find counter example for first premise of binv.
  Let counter_example := ! ( _i -> p_ );

  -- If the obdd of counter_example is anything but the 0 obdd leaf
  -- then we found a counter example.
  If ( counter_example )
    Print "binv FAILED in premise B1","\n",counter_example ;
    Return ;
  Else
    Print "B1 PASSED","\n" ;
  End


  -- Check B2.

  -- Assign # of transitions in the system to an index variable k_.
  Let k_ := _tn;

  -- Loop which executes until k_ = 0.
  While(k_)

    -- Find counter example for B2 and the current transition.
    Let counter_example := _trans[k_] & p & !next(p);

    -- If the obdd of counter_example is anything but the 0 obdd leaf
    -- then we found a counter example.
    If ( counter_example )
      Print "binv FAILED in premise B2 for transition ",k_,
            "\n",counter_example ;
      Return ;
    End

    Let k_ := k_ - 1;
  End

  Print "B2 PASSED","\n" ;
```

```
End
```

Combining the use of our two script, we could verify mutual exclusion as follows:

```
$ tlv mux-sem.pf mux-sem.smv

   ( TLV loads the smv program and the proof script,
     and presents the user with the ``>>'' prompt)

>> Call binv(mux);
```

The rule "binv" is loaded automatically from file "Rules.tlv", even before the proof script is loaded, so the proof script can use any of the defined rules.

2.3 Data types

Bdds are the only basic variable type available. However they can be used for different purposes.

The bdd implementation used can have leaves with values other than 0 or 1. This allows the representaion of non boolean functions. For example, you can create "integer variables" by simply doing:

```
>> Let k := 7;
```

What happens is that k is actually a bdd which only contains a leaf with the value 7. All arithmetic operations will work on this variable so from the users point of view it does look like an integer variable.

In a similar manner functions can be defined. For example:

```
>>   Let f := (state = idle) * 3 +
              (state = entering & semaphore = on ) * 2 +
              (state = entering & semaphore = off) ;
```

Again f is a bdd with leaves ranging in this case from 0 to 3. The intent in this example was that the function f will have smaller values in states which are closer to a stituation where state = critical.

Another use of bdds is for representing sets of variables. This is useful for quantification. For example, the supplied constants _def and _vars can be used as quantifiers.

Identifiers of dynamic variablesvariables!dynamic (as opposed to system variables which are defined in the input file) have the same syntax as SMV terms. Examples of possible names for dynamic variables: a, a[1], a[1].a[1][1] .

The characters A-Z a-z "." "_" and "-" can be used. The array indexes are OBDDs which simulates an integer value: they can be either constants, dynamic variables, or expressions which use the former two. There is no command for creating an array. An array element is created by assigning to it using the "Let" command. The array elements created need not be continuous in their index numbers.

2.4 Predefined Dynamic Variables

In the interactive mode the user has access to the transition system which is stored in the following variables:

- _tn — The total number of transitions.

Transition i ($1 \leq i \leq$ _tn) resides in three arrays:

- _t[i] — The sequential component. Defines the value of the state variables in the next state as a function of the state variable and combinational variables of the current state.
- _d[i] — The combinational component. Defines the value of the combinational variables as a function of the state variables.
- _pres[i] — Preserve all other variables which are not assigned to in this transition.

The actual transition is `_t[i] & _d[i] & _pres[i]`. They are kept separate for performance considerations.

- _i — The total initial condition
- _j[i] — Array of just conditions
- _jn — The number of items in array _j
- _cp[i] — Array of compassionate subcondition
- _cq[i] — Array of compassionate subcondition
- _cn — The number of items in arrays _cp,_cq

All arrays start from index 1.

Usually there is only one system in a single SMV file. The following variables are relevant when there is more than one system:

- _sn — The number of systems in the file
- _tn[i] — The number of transitions in system i.
- _i[i] — The initial condition of system i.
- _jn[i] — The number of justice conditions in system i.
- _cn[i] — The number of compassion conditions in system i.
- _id[i] — All variables in system i are equal to their primed versions.
- _vars[i] — All variables in system i (unprimed).

If you are implementing your own rules that it might be up to you to associate each system with its corresponding transitions. If you do not use the -disj flag then this is easy, since each system has one transition, however, the -disj flag complicates things slightly. The current rules file deals with this issue in a general way with or without the flag. Read it if you need an example of how to deal with these variables directly.

Finally, if there are variables defined with a kind, as in :

```
k : 0..3 kind of xx;
```

Then for all systems, an array item with the set of variables of this kind will be created, even if that system has no variables of that kind. For example, if the file containing the decaration above has three systems then array items xx[1], xx[2] and xx[3] will be formed, and one of them will contain the variable k defined above.

In addition, the following "constants" are also supplied:

- _id — All variables are equal to their primed version.
- _vars — All variables (unprimed).
- _def — All variables which are defined variables.

The last two constants should be used only in quantification. For example if we want to get the primed version of the bdd "state" we could do the following:

```
>>    Let next_state := (state & _id) forsome _vars;
```

This can be achived more easily by the command:

```
>>    Let next_state := next(state);
```

Any dynamic variable can be modified by the user, this includes _vars, _i, _t[], _d[] and _pres[]. However, this is not always a good idea.

If you want to change the transition system after reading the SMV file and then not change it anymore then this seems reasonable. You probably want to do this to get over some limitation of TLV, and how it translates programs to bdds. However if you want to change these variables during some calculation, then this is more dangerous, especially if after this calculation you perform other calculations which rely on the integrity of the transition system.

If you are not sure of yourself then you can always copy variables of the transition system to another set of variables and work on them instead of the original ones.

2.5 Expressions

Expressions are constructed from variables, constants and a collection of operators. The available operators are:

2.5.1 Logical operators

operator	TLV
$\wedge$	&, /\
$\vee$	| , \/
$\neg$	!
$\rightarrow$	`->`
$\leftrightarrow$	`<->`

The bdds which are a result of the expressions cannot have leaves with a value which differs from 0/1.

An additional logical operator is:

- expr1 `&&&` expr2

  This is equivalent to sat(expr1 `&` expr2) but requires much less memory and is much faster. This is useful for printing counter examples, where the entire conjunction is not needed — we only need a single assignment which satisfies the conjunction.

2.5.2 Numeric operators

Comparitive : `=  !=  <  >  <=  >=`
Computational : `+  -  *  /  mod`

The operators "*", "/" and "mod" obey the following law:

```
y * (x/y) + (x mod y) = x
```

2.5.3 Conditional Expressions: a ? b : c and `case`

This is similar to the C programming language. The syntax is

```
expr1 ? expr2 : expr3
```

For example, in the following

```
next(k) := a > 5 ? b : c ;
```

If the condition before the "?" is true, then the returned value is b, else it is c. It is shorter to write instead of "case" expressions which have a small number of options.

Conditional expressions associates to the right, so the following are equivalent:

```
next(k) := a ? b : c ? d : e;

next(k) := a ? b : (c ? d : e) ;
```

The case expression is more complex and is useful mainly in SMV programs:

```
case
  expr_a1 : expr_b1 ;
  expr_a2 : expr_b2 ;
  ...
  expr_an : expr_bn ;
esac
```

This is a case expression which returns the value of the first expression on the right hand side, such that the corresponding condition on the left hand side is true. Thus if expr_a1 is true then the result is expr_b1. Otherwise if expr_a2 is true, then the result is expr_b2, etc. If none of the expressions on the left hand side is true, the result of the case expression is the numeric value 1. It is an error for any of the left hand side expressions to contain a leaf other than 0/1.

2.5.4 Checking for validity, contradictions and equality

There are two functions: is_true, and is_false, which are useful in If or While statements, since these statements test only for satisfiability. Functions is_true and is_false can be used to test for validity or contradictions, for example:

```
If ( is_false( _i & _trans) )
  Print "_i & _trans is a contradiction\n";
End
```

is equivalent to

```
If ( _i & _trans )
  Local x := 0; -- dummy statement
Else
  Print "_i & _trans is a contradiction\n";
End
```

An additional function, is_equal can test whether two expressions represent exactly the same set or function. Note that this is very different from the "=" operator. Consider the following example:

```
-- Define boolean variable.
>> w : boolean;

-- Define two functions, r1, r2 from w to {3,4,5}
>> Let r1 := w ? 4 : 5;
>> Let r2 := w ? 4 : 3

-- Prints assignments of w for which the function
-- values of r1, r2 are equal.
>> Print r1 = r2;
w = 1,
```

```
-- Prints assignments of w for which the function
-- values of r1, r2 are not equal.
>> Print r1 != r2;
w = 1,

-- is_equal checks whether r1, and r2 are exactly the same.
-- This is done very quickly since it is done by checking
-- whether they both point to the same bdd.
>> Print is_equal(r1, r2);
0
```

2.5.5 Value sets

These expressions are used to specify sets of values which a variable may be equal to.

The set constant has the following form:

```
{ val1, val2, ... , valn }
```

2.5.6 Set operators

- expr1 in expr2

  This is the inclusion operator which checks for membership in a set expr2.

- expr1 notin expr2

  The negation of the "in" operator.

- expr1 union expr2

  Returns the union of two value sets. If either argument is a number of a symbolic value instead of a set it is coerced to a singleton set.

2.5.7 Variable sets

Variable sets are used mainly for quantification over sets of variables, by the following functions:

- expr1 forsome expr2
  expr1 forall expr2

  Returns forsome and forall functions where expr2 contains the variables to be quantified. expr2 should be a variable set.

The predefined variable "_vars" is a bdd which represents the set of all program variables. The following functions manipulate sets of variables.

- quant(a)
  support(a)
  vset(a)

  quant, support and vset do the same thing (vset stands for Variable Set). They return the set of bdd variables of the variables which are given as parameters. For example:

```
-- Remove y from transition relation 1.
>> Let sety := support(y);
>> Let _t[1] := _t[1] forsome sety;
```

However, support and vset can accept a list of bdd's instead of just one. So one could do:

```
>> Let k := support(proc1.loc, proc2.loc, y);
```

To obtain a OBDD corresponding to the set of variable proc1.loc, proc.loc and y.

- id_of(v)

  The parameter v contains an unprimed set of variables V. This function returns an expression equivalent to : V = V' .

- set_union(a,b)
  set_intersect(a,b)
  set_minus(a,b)

  perform the usual set operations. Notice that there is no negation since for that we must know the set of variable which we think of as the entire domain. Therefore when negation is needed you should use a set_minus operation. For example:

```
>> Let not_sety := set_minus(_vars,sety);
```

- set_member(v)

  The parameter v is a variable set. The function returns a variable set which contains only one variable which belongs to v. The function chooses one variable from v. For example, the following command iterates over all the variables in a variable set:

```
To iterate_vars var_set;

    -- Iterate until var_set is empty.
    -- An empty set is represented as "1" .
    While (! is_true(var_set) )

        -- Get one member of var_set.
        Let one_var := set_member(var_set);

        Print one_var;

        -- Delete member from set.
        Let var_set := set_minus(var_set, one_var);
    End
End
```

2.5.8 Satisfing assignments

- sat(expr)

  Returns a bdd which includes only one path to a leaf with the value of 1. If you have a bdd which describes many assignments to variables then the "sat" of that bdd will return a smaller group of such assignments. However, this group has a compact representation (a single path in a bdd) and is easy to print.

- rsat(expr)

  Like sat, but returns a random group of assignment, in contrast to sat which always return the same group of assignments.

- fsat(expr)
  fsat(expr, set_expr)

  This is similar to sat, but instead of returning a group of satisfying assignments, it always returns a single assignment. This still looks like a single path, but may be bigger than the original OBDD since each system variable is assigned a value, whereas in sat, some of the system variables may not appear at all.

  With two parameters, the second parameter is a variable set. The returned assignment has a single value for each variable in the variable set. If only one parameter is given, the set of variables is all the system variables which have been defined so far.

- frsat(expr)

  Like fsat with one parameter, but returns a random assignment.

2.5.9 Constructor loop expressions

.

There are two types of constuctor loops. One in expressions, and the other to duplicate parts of a section (the latter is explained in section 3.3.7).

Examples of a constructor loop in an expression:

```
 -- sum_red is the number of red traffic lights in array "light".
 -- can be declared either in DEFINE or ASSIGN sections.
 sum_red  :=  +  for (k = 1; k <= M; k = k + 1) { light[k] = red };

INIT
   -- Initialize token in the first item of the array, and
   -- nowhere else.
   token[1] &
   &  for (i = 2; i <= M; i = i + 1) { ! token[i] }
```

The first operation can be either `&`, +, or `|`.

For further details see section 3.3.7.

2.5.10 next, prime and unprime

- prime(expr)
  unprime(expr)

  Returns the primed/unprimed version of the bdd which is the result of the expression.

- next(expr)

  Similar to prime(), but should only be used for priming a single variable. The greatest difference from prime is when it is applied to an array item that has an index which is a system variable. For example, prime(a[i]) refers to item prime(i) of array a, whereas next(a[i]) refers to item i of array a.

2.5.11 Systems and Transitions: sys_num, succ, pred

- sys_num(sys_name)

  Returns the number of the system whose name is sys_name. To obtain the system number of the "main" module use sys_num(""). Otherwise the name of the system can be provided without quotes. For example in the following program sys_num(B) would return a number greater than 0. You can use this number to refer to dynamic arrays like _i, to obtain the transition system corresponding to that system. If a system of that name does not exist, then the number 0 is returned.

```
MODULE main
VAR
  B : system sys1;
  C : system sys1;

MODULE sys1
  ...
```

- succ(trans,state)
  pred(trans,state)

  These functions are equivalent to the following expressions

  succ(trans,state) = unprime ($\exists V : trans(V, V') \wedge state(V)$)

  pred(trans,state) = $\exists V' : trans(V, V') \wedge state(V')$

  They allow faster execution for model checking routines. You could avoid using these and use their definitions instead, however, the conjunction of the trans and state bdds usually creates a large bdd. The existential quantification causes it to shrink. Therefore we obtain a large intermediate bdd. Instead, the succ and pred routines do the entire calculation in one swoop without generating the intermediate bdd at all.

  The arrays _t and _pres use V and V', whereas _d uses only variable in V. Thus a possible usage of succ would be:

```
>> Let next_states := succ(_trans[k] & _pres[k], _d[k] & curr_state);
```

2.5.12 Assigning and Obtaining System Variable Values

- assign(dvar,pvar,val)

  dvar is a dynamic variable which contains a state of the program. pvar is a program variable. val is a value which is in the range of pvar. The assign function returns a state which is identical to the one in dvar, except that the variable pvar is assigned the value val.

  For example, assume that y is a boolean program value which represents a semaphore. In the initial condition _i, the value of y is 1. Suppose we want to see what happens when y is 0 in the initial condition. We could do the following:

```
>>  Let _i := assign(_i,y,0);
```

- value(expr)
  value(dynamic-var, system-var)

  With one parameter, returns value of some non-zero leaf of the bdd. With two parameters, returns the value of a particular system variable from a bdd inside a dynamic variable. For example:

```
>> Print value(_i, y);
```

  This will print the value of the system variable y, in the dynamic variable _i which contains the initial condition.

2.5.13 Other operators: size, exist

- size(expr)

  Returns the size of the bdd which is the result of the expression.

- exist(term)

  Returns one if the term is an existing variable.

2.6 Basic Commands: Let, Print, Load and Quit

Commands require ";" at the end to be processed. You may enter a command which may span many lines but only when the ";" character is read the command will be executed.

The following commands are recognized in interactive mode:

- Let variable := expression;

  Evaluate bdd of expression and assign to a dynamic variable. The variable will contain the entire bdd. If the variable hasn't been defined before it is dynamically created. The variable name can be an atom or a reference to an entry in an array (atom[index]). Variables which are declared in the input file (system variables) cannot be assigned to.

- Print expr,expr,... ;

  Prints the value of expressions. If the result of the expression is a bdd then the values of the variables of the model in the current and next states are printed according to the bdd. Note that a bdd can represent many assignments but only one is printed. If the expression is a string it is printed. To print a new line use `"\n"`. Other escape characters such as `"\t"`, `"\r"`, `"\f"`, `"\v"`, `"\b"` work as in C.

- Load "*file-name*";

  Load file *file-name* into the system. The loaded file can be a rules file or a proof script file. The loaded file can also load other files.

- Exit n;
  Quit;

  Both these commands exit tlv. Exit evaluates the parameter as a number and sets the unix variable "status" to that value.

  The parameter of Exit could be either a number, or any bdd which is an integer leaf. For example:

  ```
  >> Exit 1;

  >> Exit _tn;
  ```

2.7 Procedures and Functions

Procedures and functions allow users to extend TLV with new verification rules. Compound statements such as iterative and conditional statements, only work inside procedures and functions.

Names for functions, procedures and dynamic variables have three different name spaces, so there is not problem for a function, procedure and variable to have the same name.

- To proc-name [parameter_list];
  Proc proc-name ([parameter_list]) ;

  Define a new procedure (or overriding it if it is already defined). for example

  ```
  To myproc param;
    Let k := ! param;
    ...
    ...
  End
  ```

  Note that the "To" and "Proc" forms of defining procedures are exactly equivalent. The "To" form does not use parenthesis to enclose the parameter list.

  The parameters in the parameter list are separated by commas.

- Func func-name ([parameter_list]) ;

  Define a new function (or overriding it if it is already defined). The function returns its value using the "Return" command.

  for example

  ```
  >> Func k(i,j);
       Return i+j;
     End

  >> Print k(4);
  ```

- Run proc-name [expr1,expr2,...];
  Call proc-name ([expr1,expr2,...]) ;
  proc-name [expr1,expr2,...];

  Run a previously defined procedure. The number of parameters must match the definition of the procedure. Note that a procedure can be called simply by writing its name, without adding "Call" or "Run" in order to call it. This gives the illusion that procedures are really integral TLV commands.

- Return [expr];

  Exit procedure or function. When exiting a function, an expression should be supplied.

2.7.1 Default Values

In both functions and procedures, default values can be specified in the parameter list. A subroutine which has some parameters which have default values, can be called with fewer arguments. The value of the formal parameters, for which there is no correspoing argument, is calculated by evaluating the default value.

The default value is specified by an assignment operator, ":=", followed by an expression. For example

```
-- Prove that p is an invariant of the program
-- The second parameter is an optional strengthening assertion
Proc inv(p, phi := -1);
  If (phi = -1)
    ...
  Else
    ...
  End
End
```

The procedure "inv" can be called with either 1 or 2 arguments.

If a default value is specified for a parameter, then all following parameters also must have a default value specified.

Note that when the default value is used (i.e., when the corresponding actual argument is not specified in a call to the routine), then it is evaluated again upon each call. For example:

```
-- Return successors of a set.
Func successors(set := _i);
...
End
```

When "successors" is called with no arguments, the value of the variable _i is assigned to the parameter set. If between two subsequent calls of "successors" the value of _i would change, then so would the value of the parameter "set".

2.7.2 Local Variables

The command:
Local variable;
Local variable := expression;
is similar to the "Let" command, however, the variable will disappear after exiting the script (i.e, the procedure or function). If before calling the script such a variable already existed then after returning from the call the variable will have its original value. The scope of local variables is dynamic. For example:

```
To sort_t_by_support top;

  -- The following two lines could also be
  -- replaced by:  Local k := top;
  Local k;
  Let k := top;

  -- Create local array of support bdds.
  While (k)
    Local t_support[k] := support(_t[k]);
    Let k := k - 1;
  End

  -- Sort the t array according to the size
  -- of bdds in the t_support array.
  Run sort_t_inner 1,top;

End
```

This routine create a temporary array of support bdds which corresponds to the _t array of bdds. After exiting the routine, the array "t_support" will not exist.

However, the routine "sort_t_inner" which is called from sort_t_by_support *will* be able to access t_support. This is what meant by "dynamic scope". In this routine we used the dynamic scope in order to create the temporary array t_support, and "pass" it as a global (yet termporary) array to another routine.

Dynamic scope could be dangerous though. If the routine sort_t_inner assigns to the variable k then this would change the value of k in the calling routine even though it is declared as local. To

prevent this, if "sort_t_inner" uses k it should declare it local as well. This would create a new local variable k. The value of previous local variable k will be restored on exit of the sort_t_inner.

2.7.3 Parse Tree Parametrs

Paramaters can also be passed as parse trees, by prefixing the parameter name with a “”’. See section 2.9 for more details.

2.7.4 By-name Parameters

Parameters to functions and procedures can be passed by name. This is done by prefixing the formal parameter with "&". Passing a value by name can be used to output a value via the parameter (the only other way to output values is through global variables or the return value of a function). Passing an argument by name can also be used to pass arrays as parameters.

For example:

```
-- Passing a and b by name. These are expected to be arrays
-- of length n.
Func ex(&a,&b,n,s);
  Local result := FALSE;
  Let k := n;
  While(k)
    -- access by name.
    Let result := result | succ( a[k] & b[k],s);
    Let k := k - 1 ;
  End

  Return result;
End

-- Pass arrays _t and _pres.
Let k := ex(_t,_pres,_tn,_i);
```

The "ex" function could be used in model checking without "hardwiring" the names of the arrays in which the transition system resides. We can manually create new transtion systems and model check them, without having to use the preset arrays _t _pres and _d.

Each time the name "a" is referred to in the body of function "ex", it is replaced by the string which is the argument of the function, and then reevaluated. So referring to a[k] really accesses the array item _t[k] in the above call of ex.

Another simple example:

```
-- Example of an output parameter.
To set &out;
  Let out := 1;
End
```

```
-- This sets the value of a to 1.
Run set a;
```

2.8 Compound Statements

The following compound statements are recognized only inside scripts.

2.8.1 While and If

- While (expr) statements End
  If (expr) statements End
  If (expr) statements1 Else statements2 End

  These are the obvious control structures. The expression is considered FALSE if is equal to the constant bdds FALSE or 0 (These are in fact the same). Any non zero expression is considered TRUE.

  Inside a loop, the **Break** command exits the loop, and the **Continue** command goes to to start of loop.

2.8.2 Boolean Conditions

A common pitfall arises when one wants to ask whether an expression is a contradication or empty set (i.e. it is the OBDD leaf 0). A naive attempt to ask this can be done as:

```
-- Wrong
If ( k = 0 )
  ...
End
```

Note that the value of the expression `k = 0` will be true if the bdd representing k has **some** path leading to a zero leaf. But what we want is an expression which will be true if **all** paths lead to the zero leaf.

This can be done by using the “value” function. “value” always returns some leaf of its bdd argument. It tries to return a non-zero leaf, so it will only return 0 if its parameter is 0. So one way to ask whether k is a contradicition is:

```
If ( value(k) = 0 )
  ...
End
```

Alternatively you could add an else statement and move the desired code there. For example:

```
If ( k )
  -- Executed if k is satisfiable.
  -- Currently has some code which does absolutely nothing.
  Let k := k;
Else
  -- The code you want to perform in case k is a contradiction,
  -- (not satisfiable)
  ...
End
```

A similar problem happens if you want to check for the *validity* of an expression. Again, the following will not work:

```
-- Wrong
If ( k = 1 )
  ...
End
```

```
-- Wrong
If ( k )
  ...
End
```

Both of these only check that k is satisfiable, not that k is valid. But, checking for validity is the same as checking that the negation is a contradiction, and we already know how to do that. So to check for the validity of k try the following:

```
If ( !k )
  -- Executed if !k is satisfiable
  -- Currently has some code which does absolutely nothing.
  Let k := k;
Else
  -- The code you want to perform in case k is valid,
  ...
End
```

In TLV 4 there are two functions: is_true, is_false, which can be used instead. For example:

```
If ( is_true(k) )
  -- Executed if k is valid
  ...
End
```

2.8.3 For and Fix: additional loop statements

There are two additional looping statements.

- `For (`*var* `in [reverse]` *exp1* `...` *exp2*`)` *S* `End` — Without the `reverse` keyword, repeatedly execute statements *S* while setting local dynamic variable *var* from values *exp1* till *exp2*. If *exp1* is larger than *exp2* then the statements are not executed. If the `reverse` keyword is included, then the value of *var* starts from *exp2* and decreases towards *exp1*.

- `Fix (`*exp*`)` *S* `End` – Repeatedly execute statements *S* until the value of *exp* has reached a fixedpoint. The Fix statement is exactly equivalent to the following:

```
-- Assigned to a non existing value,
-- so that the following test will be true for the first time
-- it is evaluated.
Local new_temp := non_value;

While (new_temp != exp)
  Let new_temp := exp;
  S
End
```

2.9 Parse Tree Commands

A recent addition to TLV is the ability to handle parse trees in TLV-BASIC. A dynamic variable can hold a parse tree instead of a OBDD. The following extentions have been added to deal with parse trees:

- Let 'k := ptexpr;

  The single quote indicates that the expression on the right be evaluated as a parse tree, and not be converted to a OBDD.

- Print 'k ;

  This prints the parse tree in k. If k does not contain a parse tree then the string "k" will be printed.

  If there are several parameters then each parameter should have a separate indication for evaluation as a parse tree. For example:

  ```
  Print 'expr1, expr2, 'expr3.
  ```

  expr1 and expr3 are evaluated as parse trees, whereas expr2 is evaluated as normal.

- To 'k; Func 'k;

  Parameters of procedures and functions can be declared as parse trees. This means that the corresponding actual arguments will be evaluated as parse trees, and not be translated to OBDDs.

- A parse tree case statement has the following syntax:

```
Case (ptexpr)
     str_a1, str_a2, ... : stmts_a
     str_b1, str_b2, ... : stmts_b
     ...
     default             : stmts_z
End
```

This statement first evaluates the parse tree expression ptexpr. Then the string representation of the parse tree is compared to all the strings on the left hand side. If an equavlent string is found then the corresponding statements are executed. After they are executed, the execution continues from after the "End" of the case statement. This is unlike the C programming language where execution can "fall through" to the next statements (in C, without special care, after stmts_a is executed, the execution would continue with stmts_b).

If no string matches the expression, and if the optional default specification exists, then the default statements are executed.

For example, the following routine returns 1 if the prinicple operator is boolean, 2 if it is a numeric operator, 3 if it is negation, and 0 if it is anything else:

```
To prince_op('pt)
  Case (root(pt))
   "&", "|", "->", "<->" : return 1;
   "+", "-", "/", "*"    : return 2;
   "!"                   : return 3;
   default               : Print "An unexpected operator was entered\n";
                           return 0;
  End
```

A parse tree expression is either a normal expression or an expression which is enclosed by one of the parse tree built-in functions. When a parse tree expression is evaluated, the parse tree functions are evaluated until no more parse tree functions are found on the surface. Note that there might be parse tree functions deep inside the expression, but they will not be evaluated.

The following are the parse tree functions defined in TLV-BASIC:

- strlen(ptexpr)

  Returns a OBDD which contains the length of the parse tree expression when printed.

- ptexpr1 == ptexpr2

  Evaulates left side and right side as parse trees, and returns 1 if their representation as strings is identical.

- root(ptexpr)

  Returns a string of the principle operator of the parse tree ptexpr. In most cases the result is a straightforward string representation. The only exceptions are

  – Any identifier returns "ident".

  - Any number returns “number”.

  For example:

```
>> Print 'root(5);
number
>> Print 'root(a & b);
&
>> Print 'root(a /\ b);
&
>> Print 'root(_i);
ident
```

- op(num, ptexpr)

  Print the num-th operand of the parse tree expression ptexpr. For example:

```
>> Print 'op(1, a & b);
a
>> Print 'op(2, a & b);
b
```

- ops(ptexpr)

  Print the number of operands of the root node of the parse tree expression ptexpr. For example:

```
>> Let 'k := a & b;
>> Print ops(k);
2
>> Print ops(!s);
1
```

- addneg(ptexpr)

  Return the parse tree prefixed with a negation operator.

- and(ptexpr1,ptexpr2)
  or(ptexpr1,ptexpr2)
  iff(ptexpr1,ptexpr2)
  implies(ptexpr1,ptexpr2)
  equal(ptexpr1,ptexpr2)
  notequal(ptexpr1,ptexpr2)
  lt(ptexpr1,ptexpr2)
  gt(ptexpr1,ptexpr2)
  le(ptexpr1,ptexpr2)
  ge(ptexpr1,ptexpr2)
  eventually(ptexpr)
  always(ptexpr)

ltl_next(ptexpr)
ltl_until(ptexpr1,ptexpr2)
waitfor(ptexpr1,ptexpr2)
since(ptexpr1,ptexpr2)
backto(ptexpr1,ptexpr2)
hasalways(ptexpr)
once(ptexpr)
previous(ptexpr)
wprevious(ptexpr)

Return the parse tree obtained by using ptexpr or (ptexpr1, ptexpr2) as operands, with operator corresponding to the name of the function. For example:

```
>> Print 'and(!a, !b);
!a & !b

>> Print wprevious(a);
(~)a
```

- positive(ptexpr)

  Convert to positive form by pushing negations inside.

2.10 Status Commands

- procs;
  funcs;

  Prints a list of all currently defined TLV-BASIC procedures or functions.

- numstate dvar;

  Print the nymber of states which the dynamic variable "dvar" encodes. For example, to get the total number of possible states the variables of the program can attain (including unreachable states) do the following:

```
>> Let k := TRUE;
>> numstate k;
32
```

2.10.1 Performance

- Stats;

  Prints statistics such as number of OBDD nodes allocated so far.

- Settime;
  Chktime;

  These routines supply timing capabilities. To reset the time use Settime. To display the time which has past from the last Settime use Chktime.

2.10.2 Memory Use and Garbage Collection

- garbage;

  Forces garbage collection. This is needed more for debugging TLV.

- sz;

  Print the names of all dynamic variables with the sizes of the corresponding OBDDs.

- savelist;

  Print the sizes of all the OBDDs on the save list, and the total number of OBDDs on the save list. The save list includes all OBDDs which are saved, i.e. garbage collection will not collect them. This command is used mostly for internal debugging.

  Note that there are more OBDDs than the ones which appear in the printout of the sz command. These OBDDs include those which are used internally by TLV, or have been used for intermediate computations but have (perhaps mistakingly) not released.

The sz and savelist commands are useful if you have a lot of unnecessary OBDD nodes you want to throw away. You can use sz to see which dynamic variables have a lot of OBDD nodes. and then set their value to some small OBDD, for example, by setting them to 1. If no other dynamic variable refers to the same OBDD it will be thrown out during the next garbage collection.

2.11 Reordering commands

- reorder;

  Forces reordering now!

- reorder val;

  val should be either 0 or 1. 0 disables automatic reordering whereas 1 enables it. If reordering is enabled this does not mean that reordering will occur anytime soon, instead TLV continuously tracks the size of OBDDs and activates reordering when this becomes too large.

- reorder_bits val;
  reorder_size val;

  These have the same function as the corresponding options in the command line.

- sorder "*filename*";
  lorder "*filename*";

  Saves (or loads) the current order to a file.

2.12 Input Commands: read_num, read_string

- Let k := read_num();

  This is actually a function. It reads a line from standard input, and expects a number. It returns the number which the user has inserted. For example:

```
>> Let k := read_num();
34                                  << User input
>> Print k;
34
```

- Let 'k := read_string();

  Like read_num, it reads a line from standard input, but the user can enter any string. The function can only be used in a context where a parse tree is evaluated. For example:

```
>> Let 'k := read_string();
test1                               << User input

>> Print 'k;
test1

>> log k;

...          << All output is sent to file ``test1''

>> log;      << Restore output to stdout

>> Load k;   << Loads ``test1''
```

read_string can also be used to read **simple** expressions:

```
>> Let 'k := read_string();
_i                          << User enters

>> Print 'k;
_i

>> Print k;
        ( prints the same output as "Print _i")
```

Note that this only works for numbers and names of variables, but **not** for complex expressions. For example:

```
>> Let 'k := read_string();
a & b                           << User enters

>> Print k;

line 1: a & b undefined
```

This happens since TLV doesn't realize that "a & b" is an expression. Instead it thinks that k contains the name of a variable, and the name of the variable is "a & b".

Note that spaces matter, so if you enter a string which ends or starts with spaces, you will probably get an error when loading a file or evaluating a variable with that name. For example, if the user entered as a string "␣i␣" (where ␣ represents a blank space) then in the example above, the "Print" command will not find the correct variable, and will print an error message.

2.13 Output Commands

2.13.1 Logging Output

append "*filename*" ;
append;
log "*filename*" ;
log ;

With a single parameter, the standard output is redirected to the named file. The difference between log and append is that append appends the output to the file if the file already exists, whereas log erases the previous contents of the file.

Without any parameter, the standard output is restored to the screen.

2.13.2 Textual Output of OBDDs

- form expr;

  Prints a finite state expression which is equivalent to the expression. This is a more complete way to print out the contents of a OBDD. If you just print an expression directly, then only one satisfying assignment will be printed. "form" prints the entire expression. This is printed out as a series of TLV-BASIC assigments.

  If you copy the output of this command, and paste it back into TLV the original OBDD will be recomputed inside a dynamic variable called "formula". This can also be used to save a OBDD to a file, after a long computation for example, and then load it back later. For example:

- vform dvar;

  This is similar to the "form" command, except that the argument must be the name of a dynamic variable. If the output of the command is saved to a file and reloaded, the original OBDD will be recomuted inside a dynamic variable with the same name of as "dvar" (in contrast to "form" where the result will always be in the variable "formula").

```
   -- Function which runs a long time.
>> Let r := Reachbable();

   -- We want to save ‘‘r’’ to a file so we will not have to
   -- recompute it when we want to use it the next time we run tlv.

>> log "r.pf";  -- Direct output to file r.pf .
>> form r;      -- Print bdd to file .
```

```
>> log;         -- Stop directing output to file r.pf .
```

- bdd2cnf expr;

  Displays bdd of expression as propositional formula in conjunctive normal form (CNF). This formula uses different variable names than the system variables, since system variables with non boolean type usually have to be encoded into several boolean variables.

- bdd2prop expr;

  Displays bdd of expression as propositional formula. This is similar to bdd2cnf, but the printed formula is shorter, and not in CNF.

- bdd2cnfneg `a_1, a_2, ..., a_n` ;

  This produces a cnf output which is equivalent to the negation of the formula:

  F : `a_1 & a_2 & .... & a_n`

  If it is called with one parameter, i.e.

  ```
  bdd2cnfneg a;
  ```

  Then a is assumed to be an array, where a[0] contains the length of the array. So this is equivalent to:

  ```
  bddcnfneg a[1], a[2],... a[a[0]];
  ```

- bdd2reset;

  Resets sharing of variables which correspond to specific bdd nodes which were printed using bdd2cnf or bdd2prop or bdd2cnfneg. For example, suppose that we want to print the conjunction of two different OBDDs, however, these OBDDs share some nodes. We do not want to reprint the expressions corresponding to this node. For example:

  ```
  >> bdd2cnf a.a3 = 1;
  (!xxx__0 | a.a3_0 | FALSE) & (!xxx__0 | !a.a3_0 | TRUE) &
  (xxx__0 | a.a3_0 | !FALSE) & (xxx__0 | !a.a3_0 | !TRUE)

  >> bdd2cnf a.a3 = 1 & a.a2 = 1;
  (!xxx__1 | a.a2_0 | FALSE) & (!xxx__1 | !a.a2_0 | xxx__0) &
  (xxx__1 | a.a2_0 | !FALSE) & (xxx__1 | !a.a2_0 | !xxx__0)
  ```

  The second command did not print conjuncts which were already printed in the first command. To print all the conjuncts use bdd2reset before the command.

  ```
  >> bdd2reset;
  >> bdd2cnf a.a3 = 1 & a.a2 = 1;
  (!xxx__2 | a.a2_0 | FALSE) & (!xxx__2 | !a.a2_0 | xxx__3) &
  (xxx__2 | a.a2_0 | !FALSE) & (xxx__2 | !a.a2_0 | !xxx__3) &
  (!xxx__3 | a.a2_0 | FALSE) & (!xxx__3 | !a.a2_0 | xxx__3) &
  (xxx__3 | a.a2_0 | !FALSE) & (xxx__3 | !a.a2_0 | !xxx__3)
  ```

2.13.3 Dotty Output of OBDDs

The following comands dump bdds to a file in the format of "dotty" — a graphical tool for displaying graphs. Dotty is limited in the size of graphs it can handle. It is not recommended to display bdds over 100 nodes.

Dump expr;	`-- Dump bdd to file "out.dot".`
Dump expr,"fname";	`-- Dump bdd to file "fname.dot".`
Dump expr1,"fname",expr2;	`-- Evaluate expr2 to an integer I.`

The first command is useful during debugging via the prompt. The second and third commands are more likely to be used inside scripts. For example:

```
To model_check p;

    Local i;

    Let closure := p;
    Let new_in_closure := p;
    While (new_in_closure)

      -- Dumps current closure into files
      --     closure1.dot, closure2.dot ...
      Dump closure,"closure",i;

      -- Calculate closure
      Let old_closure := closure;
      Let closure := ...
      Let new_in_closure := closure & !old_closure;

      Let i := i + 1 ;
    End
End
```

In the displayed graph, dotted edges are 0 edges and solid edges are 1 edges. If a program variable has several corresponding bdd variables then the node is labeled with the variable name, suffixed with the bit number which this bdd encodes.

Run it from another window than TLV, but from the same directory from where you are running TLV. It is easier to load the graph files this way.

To use dotty, add the following to your path /home/verify/dot/graph/bin . It runs only on solaris.

Dotty opens a window. Pressing on the left or right mouse button opens two (different) menus. The left mouse button loads graph files such as out.dot (the default output file) or key.dot. The other menu (right mouse button) allows you to open or close windows, zoom in or out, etc. The most used command is the "load" command. There is a shortcut for load - press the "l" key.

If you have a black/white screen you may have troubles using dotty. Try to add the following to your .Xresources file.

```
DOTTY*Background:white
DOTTY*Foreground:black
```

Then do

```
$ xrdb -merge .Xresources
```

and rerun dotty.

If this still doesn't work, try to switch the "white" and "black" in the .Xresources file, and repeat the process.

2.14 Records

The notion of record is very weak in TLV, and only applies to dynamic variables. Currently, we say that a dynamic variable is in the same record as another variable, if they have the same prefix. The following are commands which treat dynamic variables in a way which is similar to a record.

- `rcopy term1, term2` — copies all the dynamic variables in the record of term2, to the corresponding variables in term1.
- `rprint term` — print the names of all dynamic variables in the record of the term.

For example:

```
>> Let ts.initial := y = 0;
>> Let ts.trans := next(y) = (y + 1) mod 4;

>> rprint ts;
ts.trans, ts.initial

>> rcopy ts2, ts;
Copying to variable ts2.trans
Copying to variable ts2.initial

>>  rprint ts2;
ts2.initial, ts2.trans
```

2.15 Pitfalls

- Note that the user must terminate each command with a semicolon. A common mistake when beginning to use TLV is to press the RETURN key, and wait for TLV to respond. However, TLV will not process the command until a semicolon is entered.
- TLV is case sensitive. For example, the following command will not work.

```
>> let i := x;
```

The command “Let” must start with a capital letter.

- Be careful when using the arithmetic operator “-”. The character “-” can appear in a variable name. If you want to use “-” as an arithmetic operator, then surround it with spaces. For example, the commands

```
>> Let a := 2;
>> Print a - 1;
1
```

work as expected. However, if you try the following:

```
>> Let a := 2;
>> Print a-1;
Line 5: a-1 undefined
```

TLV prints an error message. TLV looked for a variable called “a-1” instead of treating “-” as an arithmetic operator. Since a variable named “a-1” does not exist, an error message is printed.

- Passing strings as parameters to procedures causes the program to crash.
- Since the grammars of the SMV and the new commands are woven together the user can evaluate SMV expressions in the intercative environment which he is not supposed to. In particular the evaluation of CTL operators causes the program to crash.
- Dynamic variables and program variables occupy the same name space, therefore, conflicts can arise. For example, if a procedure tries to create a dynamic variable by assigning a value to it, but a program variable of the same name already exists. Be careful not to override global variables that you need or variables used by other procedures. To avoid problems, there is a convention for naming variables by prefixing them with a ”_”, as demonstrated by the code fragment just above.

2.16 Parsing Temporal Logic

TLV does not know how to interpret temporal logic, but it knows how to parse formulas in temporal logic. Such expressions can then be given as parameters to rules which are written in TLV-BASIC. Parse tree functions can then be used to extract parts of the formula and interpret it.

Parsing formulas in temporal logic is done by qualifying the type of temporal logic and surrounding the formula in parenthesis. For example:

```
>> Let 'k := ltl([] ! proc[1].error );

>> Let 'k := ctl(EG p & AF q);
```

To do verification with temporal logics you should look at the TLV modules in the second part of the manual, especially the chapter about model checking.

The following explains the syntax of the temporal logics in detail. Note, however, that for some logics, it is not necessary to use these special parsing capabilities. For some applications, a reasonable syntax for CTL can be simulated using only function calls, since it is easy to implement the semantics of CTL in such a way.

2.16.1 Propositional linear Temporal Logic (PTL)

A PTL formula is constructed out of propositional formulas and temporal operators. There are two classes of temporal operators, *future* and *past*. The future and past operators used in TLV are presented in table 2.1 and table 2.2, respectively.

Operator	Name	TLV representation
$\Box p$	Henceforth p	`[]`
$\Diamond p$	Eventually p	`<>`
$p \mathcal{U} q$	p Until q	`p Until q`
$p \mathcal{W} q$	p Waiting-for (Unless) q	`p Awaits q`
$\bigcirc p$	Next p	`()`

Table 2.1: Future Temporal Operators

Operator	Name	TLV representation
$\boxminus p$	So-far p	`[_]`
$\diamondminus p$	Once p	`<_>`
$p \mathcal{S} q$	p Since q	`p Since q`
$p \mathcal{B} q$	p Back-to q	`p Backto q`
$\ominus p$	Previously p	`(_)`
$\circleddash p$	Before p	`(~)`

Table 2.2: Past Temporal Operators

For the past operators we use underline, "_", in TLV rather than "-" to avoid conflict with the `<->` boolean connective.

Temporal logic formulas can be expressed in TLV when enclosed inside an "ltl()" qualifier. Such formulas can either be stored in dynamic variables as parse trees, or they can be passed as parameters to TLV-BASIC routines. For example:

```
-- Model checking: check that error does not occour
>> Run mc( ltl([] ! proc[1].error ) );
```

or

```
>> Let 'k := ltl([] ! proc[1].error );
>> Run mc (k);
```

Precedence

Temporal operators have higher binding power than the boolean ones. For example,

- `p Until q \/ x Awaits y` is interpreted as `(p Until q) \/ (x Awaits y)`
- `[] p /\ q` is interpreted as `([]p) /\ q`

Temporal operators with one operand have higher binding power than those with two operands. For example,
`[] p Until q` is interpreted as `([]p) Until q`.

Temporal operators with two operands have right associativity. For example,
`p Awaits q Awaits r` is interpreted as `p Awaits (q Awaits r)` .

All other operators in TLV are left associative.

Semantics

A *model* for a temporal formula φ is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state provides an interpretation for the variables occuring in φ.

For a given state s_j and propositional formula p we write $s_j \Vdash p$ when p is true when evaluated over s_j. Given a model σ we present an inductive definition for the notion of a temporal formula holding at a position $j \geq 0$ in σ, written as $(\sigma, j) \models p$:

For a propositional assertion,

- $(\sigma, j) \models p \iff s_j \Vdash p$

For the boolean connectives,

- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$ or $(\sigma, j) \models q$

For the future operators,

- $(\sigma, j) \models \square\, p \iff$ for all $k \geq j, (\sigma, k) \models p$
- $(\sigma, j) \models \diamondsuit\, p \iff$ for some $k \geq j, (\sigma, k) \models p$
- $(\sigma, j) \models p \mathcal{U} q \iff$ for some $k \geq j, (\sigma, k) \models q$
  and $(\sigma, i) \models p$ for every i such that $j \leq i < k$
- $(\sigma, j) \models p \mathcal{W} q \iff (\sigma, j) \models p \mathcal{U} q$ or $(\sigma, j) \models \square\, p$
- $(\sigma, j) \models \bigcirc\, p \iff (\sigma, j+1) \models p$

For the past operators,

- $(\sigma, j) \models \boxminus\, p \iff$ for all $0 \leq k \leq j, (\sigma, k) \models p$
- $(\sigma, j) \models \diamondminus\, p \iff$ for some $0 \leq k \leq j, (\sigma, k) \models p$

- $(\sigma, j) \models p\,\mathcal{S}\,q \iff$ for some $0 \leq k \leq j, (\sigma, k) \models q$ and $(\sigma, i) \models p$ for every i such that $k < i \leq j$
- $(\sigma, j) \models p\,\mathcal{B}q \iff (\sigma, j) \models p\,\mathcal{S}\,q$ or $(\sigma, j) \models \boxminus\, p$
- $(\sigma, j) \models \ominus\, p \iff j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models \circleddash\, p \iff j = 0$ or $(\sigma, j-1) \models p$

2.16.2 Propositional branching Temporal Logic (CTL)

CTL formulas can be expressed in TLV when enclosed inside a "ctl()" qualifier. Similar to linear temporal logic, such formulas can either be stored in dynamic variables as parse trees, or they can be passed as parameters to TLV-BASIC routines. For example:

```
>> Let 'k := ctl(EG p & AF q);
```

The syntax of CTL formulas is the same as the syntax for CTL formulas in SMV. Any boolean expression is a CTL formula. For CTL formulas `a, b`, the formulas `!a`, `a & b`, `a | b`, `a -> b`, `a <-> b` are CTL formulas. For a path formula `p`, the formulas `E p` and `A p` are CTL formulas. The temporal operator `E` is the existential path quantifier, and the temporal operator `A` is the universal path quantifier.

Let `a ,b` be CTL formulas. The syntax of a path formula is one of the following:

- `X a` — next time
- `F a` — eventually
- `G a` — globally
- `a U b` — until

Operators of equal precedence associate to the left. Parentheses may be used to group expressions. The order of precedence of operators is (from high to low):

```
E, A, X, F, G, U
!
&
|
->, <->
```

The following combinations of temporal operators should not have a space between them: `EX, AX, EF, AF, EG, AG`.

Chapter 3

The SMV Input Language

3.1 Introduction

Hardware systems are usually modeled using the SMV input language, which has been slightly extended in TLV. For a complete description of SMV refer to [1] [2]. This section will present some clarifications to SMV syntax and then will focus on the modifications to SMV which implemented in TLV.

A system description is prepared in a file whose name has the suffix `smv`. For example, in Fig. 3.1, we present file `mux-sem.smv` which contains the SMV description of a mutual exclusion algorithm MUX-SEM, which implements mutual exclusion by semaphores.

Such an SMV specification is input into the TLV system which creates internally the transition system corresponding to the specification. The justice requirement requests that each of the two expressions hold infinitely often in every computation.

3.2 Clarifications

The following are some issues which people have found to be unclear in the current documentation of SMV.

3.2.1 Asynchronous Composition

At the moment it is a known bug that it is not possible to nest asynchronously-composed modules using the `process` notation. That is, any variable of module type declared as `process` $X(\cdots)$ will cause module $X(\cdots)$ to be composed asynchronously with the *entire* system. This can produce unexpected results as in the example in Figure 3.2: Here variables `m3` and `m4` of module `M1` will be composed asynchronously with module `main`, despite the fact that they are declared in the scope of `M1`.

To work around this problem use the `COMPOSED` construct.

```
MODULE main
VAR
  y : boolean;  -- the semaphore variable. It is assigned by both processes.
  proc[1] : process user(y);  -- The two processes have interleaved execution.
  proc[2] : process user(y);

ASSIGN
  init(y) := 1;

JUSTICE
  !proc[1].loc=3, !proc[2].loc=3

COMPASSION
  (proc[1].loc = 2 & y > 0,proc[1].loc = 3),
  (proc[2].loc = 2 & y > 0,proc[2].loc = 3)


MODULE user(y)
VAR
  loc : {0,1,2,3,4};
ASSIGN
  init(loc) := 0;

  next(loc) :=
    case
      loc in {0,3}    : loc+1;
      loc = 1         : {1,2};
      loc = 2 & y = 1 : 3;
      loc = 4         : 0;
      1 : loc;
    esac;

  next(y) :=                       -- changes to the semaphore variable.
    case
      loc = 2 & next(loc) = 3 : 0; -- turned off when moving from l_2 to l_3
      loc = 4 & next(loc) = 0 : 1; -- turned on when moving from l_4 to l_0
      1 : y;
    esac;
```

Figure 3.1: File `mux-sem.smv`: an SMV description of Algorithm MUX-SEM for $n = 2$ processes.

```
MODULE main
VAR
  m1 : M1;
  m2 : M1;

MODULE M1
VAR
  m3 : process M2;
  m4 : process M2;

MODULE M2
...
```

Figure 3.2: Example of nested asynchronous composition

3.2.2 Integer Range Types in Variable Declaration

There a shorter way to declare an integer variable. Instead of

```
loc : {0,1,2,3,4,5};
```

You can simply write

```
loc : 0..5;
```

3.2.3 case Expressions

It is not mandatory to end a case expression with "1 : expr;" This is stated in page 11 of the SMV manual, if none of the expressions on the left hand side is true, the result of the expression is the numeric value 1. This seems rather arbitrary, so it is recommened to omit the "1 : expr" only if it is guarenteed that one of the expressions on the left hand side must be true.

3.2.4 Scope of Module Parameters

The scope of a module is the entire section up to the next MODULE keyword. This includes the JUSTICE and COMPASSION sections. This means that expressions in all sections (including JUSTICE and COMPASSION) can refer only to local system variables of the module and to parameters of it.

Names of formal parameters cannot be used outside of that module. For example, the following will not work.

```
MODULE main
VAR
  y     : boolean;
  proc1 : process user(y);
```

```
JUSTICE
  proc1.semaphore  -- This will not work since module main
                   -- does not recognize the formal paramters
                   -- of module user.

MODULE user(semaphore)
VAR
```

3.2.5 Modules as Records

Note that modules can be used to define record types. By not providing a transition relation for the modules, and by using synchronous composition, (i.e., not declaring the module as a process) the module effectively becomes a record type.

3.2.6 Translating ASSIGN to TRANS, and vice versa

In some cases it may be easier to use one section rather than the other. You may want to translate an ASSIGN section to a TRANS section or vice versa.

This is actually quite easy. To move an ASSIGN section take the following steps. If you wish, move the init() assignments in the assignment to the INIT section. The rest can be moved to the TRANS section.

When moved to the TRANS and INIT section, each line from the ASSIGN section turns into a conjunct in a large expression. The most important thing to be aware of is that the assignment operator, `:=` , which is used in the ASSIGN section is not equivalent to `=`. It is, however, equivalent to `in`. This is especially true if the left hand side of the assignment is nondeterminstic.

3.3 Changes to the SMV input language

3.3.1 FAIRNESS

TLV ignores the SPEC and FAIRNESS sections of the smv input file since they are specifically tailored for the model-checking algorithms which were implemented in the smv system.

Two new sections have been added in order to specify the justice and compassion sets.

```
JUSTICE
  phi_1,phi_2,...

COMPASSION
  (p_1,q_1),(p_2,q_2),...
```

Where phi_i, p_i and q_i are all propositional expressions.

A computation is just if it contains infiniely many phi_i states for all expressions in the justice section.

A computation is compassionate if it contains only finitely many p_i states or infinitely many q_i states for all (p_i,q_i) pairs in the compassion section.

The effect of these sections is the creation of arrays of dynamic variables which contains the bdds of these expressions.

Note that both sections DO NOT end with semicolon.

3.3.2 Systems

It is now possible to define several systems in the same SMV file, and load them both in TLV. This is useful for refinement for example.

Previously, to load two systems, one had to describe one of them in TLV-BASIC. Now both systems (or even more than two if desired) can be described in SMV.

The syntax uses the word "system". This word can be used where ever the word "process" can be used.

In each system, the preservation of variables is only calculated relative to variables which are owned by that system, and no other systems.

3.3.3 kind of

Variables can be followed by a "kind of" qualification. This allows the user to specify sets of variables. The sets of variables are exported to TLV's interactive environment, so the user can use them. For example:

```
MODULE s
VAR
  k : 0..3 kind of xx;
```

The string can be any string the user desires, as long as it is a valid name for a dynamic variable. All the variables of a system which have the same kind will be included in the same set xx[i], where i is the system number.

3.3.4 COMPOSED

The COMPOSED section allows a different kind of description of synchronous and asynchronous composition. In a module where a COMPOSED section exists, TRANS, ASSIGN and OWNED sections are not allowed, and also processes and systems cannot be declared.

The VAR section of a module with a COMPOSED section has a different meaning than usual. Modules which are instantiaed within such a VAR section are only actually instatiated if they are mentioned in the COMPOSED section. The COMPOSED section defines how the modules are composed.

`||` specifies asynchronous composition, whereas `|||` specifies synchronous composition.

The following is an example of using the COMPOSE section:

```
-- Each step should either increase or decrease (mod 5) bc, and either
-- subtract 1 (mod 5) from f or add one to d. f,d are oreserved when
-- they are not assigned to).
```

```
MODULE main

VAR
   bc  : 0..4;
   d   : 0..4;
   f   : 0..4;

   Z : add_one(bc) ;
   B : add_one(bc) ;
   C : subtract_one(bc) ;
   D : add_one(d) ;
   F : subtract_one(f) ;

COMPOSED
  (B || C) ||| ( D || F )

MODULE add_one(x)
ASSIGN
  init(x) := 0;
  next(x) := (x + 1) mod 5;

MODULE subtract_one(x)
ASSIGN
  init(x) := 0;
  next(x) := case
               x = 0 : 4;
                   1 : x - 1;
             esac;
```

Note that since Z does not appear in the COMPOSED section, its declaration is meaningless.

The COMPOSED section has a special form of constructor loops, similar to constructor loop expressions in section 2.5.9. . These loops can be used to compose an array of process either synchronously or asynchronously. For example:

```
MODULE main

DEFINE
  M := 3;
VAR
  async : array 1..M of Flip;
  sync  : array 1..M of Flip;
COMPOSED
  -- Asynchronous composition. Each step, only one process moves.
  ||  for (i=1 ; i <= M; i = i + 1) { async[i] }

  |||
```

```
  -- Synchronous composition.  Each step, all processes moves.
  ||| for (i=1 ; i <= M; i = i + 1) { sync[i] }

MODULE Flip
  ...Do something...
```

3.3.5 OWNED

Usually, the set of owned variables is the set of variables a processes assigns to, and therefore, it is the set of variables which other processes should preserve when they are executing.

However, this raises a problem of what variables are owned when a TRANS section is used. The TRANS section gives us no way to find out what the owned variables are.

This can be solved by the OWNED section. In the OWNED section you can explicitly mention the variables which should be considered owned by this process. These are added to any owned variables which are already known (for example, from an ASSIGN section).

The syntax of the OWNED section is as following:

```
OWNED
  x, y
```

This declares the variables x and y as owned by the current process.

Therefore, there is no problem from now on to use the TRANS section freely. Preservation of variables will be computed.

3.3.6 Using system variables as indexes of arrays

In the original SMV array items could only be referenced by using constants. Now you can use other system variables as array indexes. For example:

```
DEFINE M := 4;
VAR
  j     : 1..M;
  a     : array 1..M of boolean;
ASSIGN
  next(a[j]) := !a[j];
```

Note that such assignments are not checked for circular and multiple assignments. This allows the user to do:

```
next(a[i]) := b;
next(a[j]) := c;
```

Which might be useful, however, if "i" and "j" happen to be the same value, and c and b have different values then the conjunction with the transition relation will lead to a contradiction, and the computation will terminate.

Another issue is that the other array items are not preserved, and the array items are not considered to be owned by the process. You have to attend to this explicitly. This can be done using constructor loops.

3.3.7 Constructor Loops

A contructor loop has the form:

```
for (var = expr1; cond; var = expr2) { }
```

Where expr1, cond, and expr2 can be any expressions which use constants, variables which are defined by enclosing constructors (since constructors can be nested), or DEFINEd symbols whose values are constants or such variables.

For example, in order to preserve the rest of the array a, in the example above, (using system variables as indexes of array) one could add the following to the ASSIGN section:

```
ASSIGN

  -- Preserve, all items other than j. Don't limit the
  -- value of Item a[j] ( do this by assigning to it the
  -- entire range of a[j] ). Note that this assignement not only
  -- preserves the values when a[j] is assigned, but also causes
  -- the array items to be owned by the current process, and thus
  -- to be preserved when other processes are executed.

  next(a[j]) := !a[j];
  for (k=1;k <= M; k = k + 1) {
    next(a[k]) := k != j ? a[k] : {0,1};
  }
```

Another example:

```
VAR
    -- Diner template, passing left and right forks.
    fork : array 1..M of boolean ;
    for (i = 1; i <= M; i = i + 1) {
      diner[i]: process Dine(fork[i],fork[(i mod M) + 1]);
    }
```

Note that the index of the contructor loop is actually a dynamic variable, which is destroyed after the loop has finished.

Such constructors can be used in other sections as well. In the ASSIGN, DEFINE and VAR sections, the "}" token should *not* be followed by a semicolon. In the JUSTICE, COMPASSION and OWNED sections, the for {} construct has to be separated with commas from other terms which come before or after it. For example:

```
JUSTICE
  !r[1],
  for (i=1;i <= M; i = i + 1) {
    a[i]
  },
  !r[2]
```

The COMPOSED section has a special form of constructor loops. See section 3.3.4.

3.3.8 Conditional Statements

The DEFINE, VAR, JUSTICE, and COMPASSION sections can use conditional statements in the same way that constructor loops are used. Conditional statments are not available in the ASSIGN section.

For example:

```
DEFINE
  M := 3;
VAR
  for (k=1; k <= M; k = k + 1) {

    if ( k = M ) {
       p[k]: boolean;
    }
    else {
       p[k]: {0,1,2,3};
    }

  }
```

Of course, the "else" clause can be omitted. Note that the curly brackets are always needed for the "then" and "else" parts, in contrast to C where a single statement does not require a new "block" as indicated by the curly braces.

3.3.9 default in

Can be used only in the ASSIGN section. It provides a default value. For example, to pass a token in a ring, assuming the token is currently at position j:

```
  default {
    -- Preserve all array items.
    for (k = 1; k <= M; k = k + 1) {
       next(token[k]) := token[k];
    }
  }
  in {
    -- Move token.
      next(token[j]) := 0;
      next(token[(j mod M) +1]) := 1;
  }
```

The "in" block overrides the default block. Note that there is no special treatment for items `j` and `(j mod M) + 1` in the default block.

Chapter 4

The SPL Input Language

4.1 Overview

The SMV input language is suitable for describing hardware circuits. For a higher level description for reactive programs, we adopted the *Simple Programming Language* (SPL).

The full formal definition (according to [3]) of the syntax and semantics of SPL can be found in appendixes A and B. These appendixes were taken from Boris Liberman's masters thesis (who also wrote the SPL to SMV compiler [5]).

Following are some of the most important features of SPL:

- A Pascal-like language, augmented by parallelism and some special statements for coordination and communication among concurrent processes.
- Each program consists of one or more concurrent processes.
- Each statement of the program has a label. This label gives rise to *location* of control for each of the processes.
- Schematic statements, which provide an abstract representation of more detailed activities.
- Concurrent execution of the parallel processes is represented by *interleaving.* That is, computation proceeds by steps where, at each step, a single transition belonging to one of the currently active processes is taken.

4.2 Example — Peterson's Algorithm for Mutual Exculsion

Fig. 4.1 presents Peterson's algorithm for mutual exclusion written in SPL. In the problem of mutual exclusion, processes are competing for some resource which should be properly shared between them.

The reactive (non-terminating) nature of the program is represented by the `loop forever` statement which never terminates.

The `await` statements in l_3 and m_3 of Fig. 4.1 are examples of coordination statements. The intended meaning of an `await` statement is that execution of the process has to wait until the

```
local y1, y2: bool where y1, y2 = F;
s           : int  where s = 1;

P1::
[
  l_0: loop forever do
  [
    l_1: noncritical;
    l_2: (y1, s) := (T, 1);
    l_3: await (!y2) | (s != 1);
    l_4: critical;
    l_5: y1 := F
  ]
]

||

P2::
[
  m_0: loop forever do
  [
    m_1: noncritical;
    m_2: (y2, s) := (T, 2);
    m_3: await (!y1) | (s != 2);
    m_4: critical;
    m_5: y2 := F
  ]
]
```

Figure 4.1: Program MUX-PET: Peterson's algorithm for mutual exclusion.

argument of the statement becomes true. The argument of an `await` statement must be of boolean type.

The `noncritical` statement is a schematic statement which expresses all activity of the process which is independent of any other competing process. Another schematic statement is `critical` which represents the activity which has to be performed in protected mode with respect to competing processes. These two statements as they appear in the program are usually referred to as the *non-critical* and *critical sections* respectively.

Declarations

An SPL program starts with a *declaration* which consists of a sequence of *declaration statements* of the form

$$\text{mode variable}, \ldots, \text{variable} : \text{type } \mathbf{where}\ \varphi$$

The mode of each declaration statement can be **in**, **local**, or **out**. Specifying the mode of a variable is optional.

The optional assertion φ restricts the initial values of the variables. For *local* and *output* variables it is restricted to be of the form $y = e$, specifying an initial value. For an *in* variable, it may be any constraint on the range of values expected for these variables (this also applies when no mode is specified).

The program is not allowed to modify (i.e., assign new values to) variables that are declared as **in** variables. The distinction between modes **local** and **out** is mainly intended to help in understanding the program and has no particular formal significance.

The type of the variables can be one of the following:

bool — boolean variable.

[i..j] — An integer variable of range i till j.

Since TLV can only handle finite state programs, there is no unbounded integer type. An integer type is specified by a range of allowed values.

Statements

Statements in the body of the program may be labeled. The labels are used as names for the statements in our discussion of the program and later in program specifications and proofs. Following are some statements which are available in SPL:

- $\overline{y} := \overline{e}$ — Assign the list of expressions $\overline{e}$ to the list of variables $\overline{y}$ of the same length and corresponding types.

- **await** c — Waits until the boolean expression c becomes true, at which point the command terminates.

- **if** c **then** S_1
  if c **then** S_1 **else** S_2 — Conditional statements.

- **while** c **do** S
  loop forever do — Iteration statements.

- **request** r
  release r — Semaphore statements.

- $S_1 \Big\| S_2$ — Parallel execution. The execution is interleaved so at each step, one process is chosen and one step of it is executed.

4.3 The Implementation of SPL in TLV

Differences from Formal Definition of SPL

Only part of SPL has been implemented in TLV. Other parts implemented in the future, but there are also details which cannot be implemented in TLV, mainly because TLV can only handle finite state systems.

The current implementation supports the following:

- Constants — T, F (True and False)

- Operators — `& | ! -> + - * / div > < = <= >= !=`

- Declarations — local, in , out, bool

- Simple Statements — skip, await, request, release, assignment (no more than two items inside paranthesis)

- Schematic Statements — noncritical, critical , produce, consume

- Statements — when-do, if-then-else, loop forever, while-do, or (selection), ";" (concatenation), `||` (cooperation) "[" "]" (block)

The following are **not** currently supported:

- Arrays are not supported yet.

- channels — note that unbounded asynchronous channels are impossible to implement in TLV.

- Grouped statements — `<S>`

Additional Features

- Comments in SPL are similar to that of C++. When the token `//` appears, the entire line after the `//` is ignored. Also it is possible to enclose comments between `/*` and `*/`.

- There are special expressions which can be used in TLV for programs written in SPL.

  The expression `at_l_0` is a shortcut in TLV which translates into the expression `pi1 = l_0`.

- All labels must be of the form letter_numbers

Part II

TLV Modules

Chapter 5

Utilities

The file Util.tlv contains various utilities, including the array utilities, which are described in the next section. Future routines which can be useful for different modules should be in this file.

Util.tlv also includes routines to print all satisfying assignments of a OBDD. There are two routines in Util for this : print_sat and print_fsat, which use sat and fsat respectively. Both routines accept a single parameter. Their implementaton is very simple. Here, for example is the implementation of print_fsat:

```
To print_fsat e;
   Local s;
   While(e)
    Let s := fsat(e);
    Print s;
    Let e := e & !s;
  End
End
```

Chapter 6

Array Utilities

In TLV-BASIC it is permissible to define arrays which have noncontinuous ranges. TLV-BASIC allows the use of dynamic variables which have an index, but TLV-BASIC does not have a strong notion of what an array object is. For example, given the name of an array, it is difficult to know what items of this array are actually occupied.

The standard rule files have some useful routines for manipulating arrays. The arrays manipulated with these routines are assumed to start from index 1, and have their length stored in index 0.

The routines are:

- `push arr, val` – If the array does not exist it will be created.
- `pop(arr)` — Function which also returns the last array item.
- `top(arr)` — Returns the last array item.
- `new arr` — Initialize array to have 0 items
- `delarr arr` — Delete the array
- `length(arr)` — Return number of items in array.
- `copy arr1, arr2` — Copy arr2 to arr1.
- `append arr1, arr2` — Append arr2 to arr1.
- `printarr arr` — Print array.

For example:

```
>> push a, 5;
>> push a, 6;
>> push a, 7;
>> Print length(a);
3
>> copy b, a;
```

```
>> append b, a;
>> printarr b;
5
6
7
5
6
7
>> Print pop(b);
7
>> Print top(b);
6
```

Chapter 7

Transition System Module (TS)

A user or rule programmer can directly access the dynamic variables which represent the transition system. However, this representation is a bit inconvenient. Furthermore, if one wants to represent a transition system using a different implementation (for performance reasons) then one would probably define new specific rules which utilize that specific implementation, and the rest of the predefined rules will not work for that implementation.

The TS module encapsulates transition systems separating what one can do with transition systems, and how they are implemented. To add a new implementation, you just have to change the implementation of TS (which is not difficult), and the rest of the rules will continue to work as before.

The TS module can be used to create new transition systems easily. Transition systems are identified by a number. Many of the routines in the TS module (and other modules) accept a Transition System Number (tsn) which identifies the transition system the routine should act on. The Transition System Number is usually given as the last parameter and has a default value, which is the current system. Thus, once the current system is correctly set, you can call the routines without providing the Transition System Number.

The following sections describe the routines provided by the TS module. Read the rule files for many examples which use TS.

7.1 Types of Transition Systems

There are currently two implementations.

- DSPLIT transition systems, are represented as a disjunction of transitions, each of which is a conjunct which is composed of three parts. This is useful for representing transition systems which originated from systems which were specified in the smv file.

- DISJUNCT transition systems are represented simply as an array of disjuncted transitions. This is used, for example, for creating testers.

7.2 Initialization

File "Rules.tlv" initializes the TS module, by calling "init_ts", creating either DSPLIT or DISJUNCT transition systems for all systems loaded from the smv file. The default is to create DISJUNCT transition systems.

```
Proc init_TS( ts_type := DISJUNCT );
```

7.3 The Current Transition System

```
-- Set and get the current transition system.
Proc set_curr_ts(name or number)
Func get_curr_ts();

Proc push_curr_ts(name or number);
Proc pop_curr_ts();
```

The current transition system can be set either by name or by number. For example, suppose the smv file is as follows:

```
MODULE main
VAR
  B : system Concrete;
```

Then the following statements are equivalent:

```
set_curr_ts(B);

set_curr_ts("B");

set_curr_ts(sys_num(B));
```

To set the current transition system to the "main" system, use the empty string, "", as the parameter.

The push and pop routines change the current transition system, but allow restoring it. If you write or use a rule which does not have a transition system number as a parameter, you can push the current system number. The old one is saved in a stack. All invocations of rules which use the TS module will now use the new system. When you are done, calling pop_curr_ts will restore the old system to be the current one. You can push as many systems as you like, since it is really implemented as a stack. However, every push should be matched by a corresponding pop.

7.4 Obtain Number of Systems

```
-- Return number of transition systems which were loaded from the
-- smv file.
Func get_num_sys();

-- Return number of transition systems which currently exist.
```

```
-- This can be greater than get_num_sys, since the user may have
-- created new transition systems.
Func get_num_ts();
```

7.5 Creating Transition Systems

The empty transition system has FALSE as a transition relation and TRUE as an initial condition.

```
-- Get Transition System Number for a new transition system.
Func new_ts(type := DISJUNCT);


-- Copy transition system.
Proc copy_ts(to, from);

-- Return a new transition system which is a copy of tsn.
Func copy_ts(tsn := _s.cur);


-- Return new system which is synchronous/asyncronous
-- composition of the systems identified by the two parameters.
Func sync (ts1, ts2);
Func async(ts1, ts2);


-- Delete last transition system which was created.
Proc delete_last_ts();
```

7.6 Manipulating Components of Transition Systems

7.6.1 I, V

```
-- Set and get initial condition.
Proc set_I (_i, tsn := _s.cur);
Func I(tsn := _s.cur);


-- Set and get variable set of transition system.
Proc set_V(v, tsn := _s.cur);
Func V(tsn := _s.cur);

-- Add a variable to the transition system.
```

```
Proc add_V(v, tsn := _s.cur);
```

The parameter "v" in set_V and add_V is a variable set (see section 2.5.7). For example, to add a variable `x` to a transition system, do:

```
add_V  vset(x);
```

The following line adds 4 variables, 2 of which are defined inside an instantiated module called `m1`:

```
>> add_V vset(x , y,  m1.z, m1.loc);
```

7.6.2 T

```
-- Add a disjunct, or a conjunct, to the transition relation.
Proc add_disjunct_T(_t, tsn := _s.cur);
Proc add_conjunct_T(_t, tsn := _s.cur);

-- Add a disjunct in DSPLIT format.
-- Use only in DSPLIT transition systems.
Proc add_split_T(seq, comb, pres, tsn := _s.cur);

-- Re-initialize transition relation to FALSE.
Proc erase_T(tsn := _s.cur);

-- Return disjunctive component i of transition relation.
Func Ti(i, tsn := _s.cur);

-- Return the number of disjunctive components.
Func nT(tsn := _s.cur);

-- Return the total transition relation.
Func T(tsn := _s.cur);
```

7.6.3 Fairness

```
-- Add justice requirement
Proc add_J(_j, tsn := _s.cur);

-- Append justice requirements from one system to another.
-- The justice requirements of the ``from'' system do not change.
Proc append_J( to, from) ;

-- Remove last justice requirement which was added.
Proc pop_J( tsn := _s.cur);
```

```
-- Return justice requirement i.
Func Ji(i, tsn := _s.cur);

-- Return the number of justice requirements.
Func nJ(tsn := _s.cur);


-- Add compassion requirement
Proc add_C(_cp, _cq, tsn := _s.cur);

-- Append compassion requirements from one system to another.
-- The compassion requirements of the ''from'' system do not change.
Proc append_C(to, from);

-- Remove last compassion requirement which was added.
Proc pop_C( tsn := _s.cur);

-- Return compassion requirement i.
Func Cpi(i, tsn := _s.cur);
Func Cqi(i, tsn := _s.cur);

-- Return the number of compassion requirements.
Func nC( tsn := _s.cur );


-- Append both justice and condition from one system to another.
Proc append_fairness(to, from);
```

7.6.4 Owned

```
-- Return set of variables specified as kind ''own'' in
-- the smv file.
Proc owned(tsn := _s.cur);

Proc set_O (o, tsn := _s.cur);
Proc add_O (o, tsn := _s.cur);
```

In situations where there are several systems, and one wants to prove abstraction, it is sometimes needed to know which variables are owned by which system. TS does not use this internally, it is only used by other rules. You can safely ignore these routines.

7.7 Operations on Transition systems

```
-- Change the type of a transition system.
```

```
Proc set_type(type, tsn := _s.cur);

-- Returns new system which is equivalent to the system identied
-- by the "tsn" parameter, but which has more disjunctive components,
-- which may improve performance. The parameter ‘‘arr’’ is the name
-- of an array, according to which the disjunction is performed.
Func disjunct (&arr, tsn := _s.cur);

-- Modify the transition system such that all states which have no
-- successor become idling states, who have themselves as a successor.
Proc Add_idle(tsn);
```

7.8 Successors and Predecessors

```
-- Return 1-step predecessors/successors.
Func pred1(s, tsn := _s.cur);
Func succ1(s, tsn := _s.cur);

-- Get all successors or predecessors of a set.
Func successors(set, tsn := _s.cur);
Func predecessors(set, tsn := _s.cur);

-- Return the subset of "state" which has no successors.
Func no_successors(state, tsn := _s.cur);

-- Return the subset of "state" which has successors to "to".
Func has_successors_to(state, to, tsn := _s.cur);

-- Get set of reachable states.
Func reachable(tsn := _s.cur);

-- Compute shortest path from source to destination using tsn.
-- The result is appended to array arr.
Proc path(source, destination, & arr, tsn);
```

7.9 Refute

```
-- Conjunct ‘‘conj’’ with the transition system, and find
-- a single counterexample, if this conjunction is satisfiable.
Func refute_T(conj, tsn := _s.cur);
```

Refute can be useful to verify premises of deductive rules.

7.10 Restriction

There are situation where one wants to temporarily restrict the transition relation of a transition system, and restore it back later. The following routines allow the user to restrict and reverse the restriction of a transition relation.

Restriction can be called multiple times. The combined effect is equivalent to the conjunction of all restrictions. However, a single unrestriction reverts the transition system back to its original state, reversing the effect of all restrictions.

```
-- Restrict transition system, i.e.,
--   new_T := T & pre(V) & post(V')
-- However, you do not have to prime post,
-- the routine does that.
Proc restrict(pre, post, tsn := _s.cur);

-- Restrict with a relation, i.e.,
--   new_T := T & res(V,V')
Proc restrict_rel(res, tsn := _s.cur);

-- Reverse effect of all restrictions.
Proc unrestrict(tsn := _s.cur);
```

Chapter 8

Simulations

8.1 Creating

Simulations can be use to sample the behaviour of the program, to learn how it works, and to find bugs in the program. The following commands create simulations:

- `simulate n`
  `sim n`

  Create a new simulation, where n is the number of execution steps which should be carried out. However, it is possible that the program will terminate before the number of requested simulation steps.
- `start e` — create a new simulation which has only the first step (which is the initial state), constrained according to expression e. This command can be used to start a simulation with the input variables initialized with a particular assignment.
- `cont n` — extend an existing simulation by n steps.
- `step e` — try to extend an existing simulation by one step, where in the next step, the expression e holds. If there is no such step then the simulation is not extended.
- `trunc n` — truncate simulation at step n.

In the following invocation of "simulate" the user requests to create a simulation of 100 steps. The first step is chosen at random such that it will conform to the initial condition. If the program halts then no more steps are generated. In the example the program halted after 13 steps.

```
>> simulate 100;
Adding 99 new steps to simulation array
Simulation execution terminated at step 13
New simulation created
```

Each execution of the simulate command results in a different random simulation which starts with different initial values (as long as they conform with the initial condition). Running the simulate command erases the previous simulation.

8.2 Printing

The following commands display simulations:

- `show_all`
  `sa`

  Print the entire simulation. Each step is displayed as a state which the program is in. In a state, each variable of the program has a specific value. If the loaded program was written in SPL then additional variables, pi1,pi2,... will appear. These are program variable which corresponds to program locations of different processes.

- `curr` — print current (i.e. last) item of the simulation.

- `last` — print last step of the simulation. The difference from the "curr" command is that if the simulation halted, then "curr" will print the halted state, HALT, whereas "last" will print the state before halting.

- `show n` — show a part of a simulation which includes step n. Try to also show several steps before and after n.

Evaluating Expressions over a State

The most basic operation we can perform on boolean expressions is to find their truth value given an interpretation to the propositions. States of a simulation can be used as such an interpretation.

- `eval exp,step_no` — evaluates the expression "exp" according to the interpretation of step "step_no" of the simulation.

- `eval_next exp,step_no` — evaluates the expression "exp" according to the interpretation of step "step_no" as the unprimed variables, and step "step_no" + 1 as the primed variables.

For example, consider the following execution:

```
>> sim 3;
First step of simulation has been created
Adding 2 new steps to simulation
New simulation created

>> sa;
---- Step 1
y = 1,              proc[1].loc = 0,    proc[2].loc = 0,    proc[3].loc = 0,
---- Step 2
y = 1,              proc[1].loc = 0,    proc[2].loc = 0,    proc[3].loc = 0,
---- Step 3
y = 1,              proc[1].loc = 0,    proc[2].loc = 1,    proc[3].loc = 0,

>> eval y & proc[1].loc= 1, 1;
0
```

```
>> eval y & proc[1].loc= 0, 1;
1

>> eval proc[2].loc = 0 & next(proc[2].loc) = 0, 2;
1
>> eval_next proc[2].loc = 0 & next(proc[2].loc) = 0, 2;
0
```

The first two invocations of eval printed evaluations of the formulas on the first state of the simulation. The following invocations use the second state of the simulation.

The third invocation of eval printed 1 since it does not take state 3 into consideration. eval_next uses state 3 for values of the primed variables, and that is why it prints the correct result.

Additional commands

- `find e` — Find a simulation step (if exists) for which the expression "e" holds. This is similar to checking whether $\Diamond e$, but for a finite simulation.

8.3 Annotation with Temporal Formula

Annotation of simulations with temporal formula is useful mainly for education, since it demonstrates how different temporal formulas are interpreted along a sequence of states (see the tutorial for more information). Annotation can be performed for simulations which are generated by the commands presented in the previous section, or it is possible to create arbitrary sequences of states.

There are two commands to annotate simulations:

- `fsimtl ltl( temporal_formula )`
  Annotates simulation by a temporal formula. The formula cannot use past temporal operators. The computation being annotated can be infinite. `fsimtl` is an acronym for "Future-limited version of SIMulations annotated by Temporal Logic".

- `simtl ltl( temporal_formula )`
  Annotates simulation by a temporal formula, which can contain both future and past temporal operators. The computation being annotated must be finite. The command does its best to try to interpret future temporal operators, however, the semantics of future temporal operators for finite sequences[6] are different than usual.

Both commands display the current simulation annotated with temporal formulas. Each step of the simulation is printed with the truth value of the temporal formula, and the truth values of any subformula whose principal operators are temporal.

Note that in order to generate an infinite sequence of states you should use the `setloop` command. Without this, formulas of the form `[] x` will always be false. Before executing a `simtl` the loop should be removed, if you have set one. This can be done as follows:

```
>> setloop 0;
Loop back has been canceled
```

Note that formulas of the type $\square\, \varphi$ are always false on finite sequences. On the other hand $\diamondsuit\, \varphi$ is true for state j of the sequence iff for some state larger or equal to j the formula φ holds. Thus $\diamondsuit$ for finite sequences is similar to the `find` command for simulations.

The semantics for finite sequences tries to approximate from below the answer for an infinite sequence. If a formula holds for the finite semantics, on a finite sequence f, then it should also hold using the regular semantics on all infinite extensions of f. On the other hand, if a formula does not hold for the finite semantics on finite sequence f, there still may be infinite extensions for f such that the temporal formula *does* hold.

Creating arbitrary simulations

The following commands allow the user to create arbitrary sequences which are unrelated to the program which tlv has been loaded with. Their main use is for educational purposes. One can create arbitrary simulations and observe how they are annotated by temporal formulas, in order to learn the meaning of various temporal operators.

- `setstep n,e` — replace step `n` of the simulation by a new state on which the expression `e` holds.
- `appstep e` — append a new step to the end of the simulation such that the expression `e` holds on the new step.
- `setvar n,v,e` — in step `n`, change the value of variable `v` to be `e`.
- `setloop n` — create a loop in the simulation such that step `n` is a successor of the last step of the simulation. To cancel the loop use the command `setloop 0`.

The commands `setstep` and `appstep` modify or add steps to the simulation. The value of the variables can be whatever the user desires with no consideration of the program currently loaded by TLV. The `setvar` command lets the user modify the value of a single variable in a step. The `setloop` command is needed so we will be able to represent infinite sequences of states using a finite representation.

Chapter 9

Simple Model Checking

The rules described in this chapter use model checking techniqes, but do not accept temporal formulas as a parameter. Each rule checks a specific property. These routines are typically faster compared to model checking with temporal formulas, but they are of course more limited in the properties which they can check.

9.1 Deadlocks

Deadlocked states are those whose only successor is themself. The routine "check_deadlock" checks for the abscence of deadlocks in the program. The routine accepts an optional parameter, tsn, indicating the transition system number which is to be checked. Consider the program mux-sem.smv from section 3.1 for mutual exclusion with semaphores. The program has no deadlocks. We can check this by doing:

```
>> check_deadlock;

 Check for the absence of Deadlock.
Model checking Invariance Property

*** Property is VALID ***
```

Note that in program mux-sem, if all processes are trying to enter the critical section, `loc = 3`, then it is guaranteed that the semaphore is such that one process may proceed. However, if we change the initial condition this is no longer the case, and the program may deadlock.

```
>> set_I(TRUE);
>> check_deadlock;

 Check for the absence of Deadlock.
Model checking Invariance Property

*** Property is NOT VALID ***
```

```
Counter-Example Follows:

---- State no. 1 =
y = 0,              proc[1].loc = 2,    proc[2].loc = 2,    proc[3].loc = 1,

---- State no. 2 =
y = 0,              proc[1].loc = 2,    proc[2].loc = 2,    proc[3].loc = 2,
```

The counter example which is printed demonstrates a computaton from an initial state to a state which is deadlocked. The deadlocked state is state number 2, where all the processes are at location 2, trying to enter the critical state, but the semaphore does not allow them to.

9.2 Invariance

The invariance rules verifies that a property holds on all reachable states. For program mux-sem we can prove the following invariance property.

```
>> Invariance( !(proc[1].loc = 3 & proc[2].loc = 3) );
Model checking Invariance Property

*** Property is VALID ***
```

Calling Invariance(p) is the same as verifying that $\Box\, p$ (The routine accepts an optional parameter, tsn, indicating the transition system number which is to be checked). The property in the above example shows that the property of mutual exclusion holds for processes 1 and 2. The following is an example where invariance does not hold:

```
>> Invariance( !(proc[1].loc = 3 & proc[2].loc = 2) );
Model checking Invariance Property

*** Property is NOT VALID ***

Counter-Example Follows:

---- State no. 1 =
y = 1,              proc[1].loc = 0,    proc[2].loc = 0,    proc[3].loc = 0,

---- State no. 2 =
y = 1,              proc[1].loc = 0,    proc[2].loc = 1,    proc[3].loc = 0,

---- State no. 3 =
y = 1,              proc[1].loc = 0,    proc[2].loc = 2,    proc[3].loc = 0,

---- State no. 4 =
y = 1,              proc[1].loc = 1,    proc[2].loc = 2,    proc[3].loc = 0,
```

```
---- State no. 5 =
y = 1,              proc[1].loc = 2,    proc[2].loc = 2,    proc[3].loc = 0,

---- State no. 6 =
y = 0,              proc[1].loc = 3,    proc[2].loc = 2,    proc[3].loc = 0,
```

A counter example is printed which is a computation which starts from an initial condition, and ends with a state for which the property does not hold.

9.3 Response

The following are rules which verify response properties.

- `Temp_Entail p, q [,tsn]` — verify whether $\Box(p \rightarrow \Diamond q)$ is valid. The final optional parameter is the transition system number for which the property is checked.
- `Temp_Entail_tl p, q [,tsn]` — similar to Temp_Entail, but is implemented by calling more general model checking routines. Temp_Entail should be faster.
- `check_react p,q,r [,tsn]`
  `check_react2 p,q1,q2,r [,tsn]`
  `check_react3 p,q1,q2,q3,r [,tsn]`

  Verify whether the following is valid:

$$\Box((p \ \wedge \ \bigwedge_{\forall i} \Box \Diamond q_i) \rightarrow \Diamond r)$$

Chapter 10

Model Checking

Given a finite state program P and specification φ (usually given as a formula in temporal logic) we want to decide whether φ is valid over finite state program P, i.e. whether all the computations of P satisfy φ. The problem of deciding whether φ is valid over P is known as *Model Checking*.

10.1 Automata as Specification

The TS module can be used to manually build an automata which will serve as a negation of the specification. Such an automata can be composed with the program, and we can check that the composition is feasible.

To compose the tester with the program and perform model we use the command `mc_tester` . If the composition is feasible then a counter example is presented. For example:

```
-- Create new transition system (the tester)
>> Let t := new_ts();

-- Set the transition of the tester to TRUE.
>> add_disjunct_T TRUE, t;

-- Note that initial condition is TRUE by default.

-- Find feasible computation of synchronous composition of tester t
-- with the current transition system.
>> mc_tester t;
Model checking...

*** Property is NOT VALID ***

Counter-Example Follows:
...
```

Since this tester did not restrict the system in any way, we just get some computation of the system. The message "Property is NOT VALID" refers to the negation of the tester.

10.2 Propositional Linear Temporal Logic (LTL)

For LTL there are rules for checking validity and for general model checking. The syntax of LTL is specified in section 2.16.1.

10.2.1 Validity and Satisfiability

A temporal formula φ is *valid* if it holds over *all* models σ. A temporal formula φ is *satisfiable* if it holds over *some* model σ. TLV can be used to check the validity and satisfiability of LTL formulas. For this we do not require a running program since we are not checking whether the temporal formula holds for a computation of a program. All we need is to declare some program variables to use as propositions. For the following examples we use the program in Fig. 10.1.

```
MODULE main
VAR
  p: boolean;
  q: boolean;
  r: boolean;
```

Figure 10.1: Program empty.smv

The command `valid` checks for validity of temporal formulas for both future and temporal operators. The syntax is:

- `valid ltl( temporal_formula )`

Here are some examples for checking validity:

```
>> valid ltl([]p -> <>p);
Model checking...

*** Property is VALID ***


>> valid ltl([]p);
Model checking...

*** Property is NOT VALID ***

Counter-Example Follows:
```

```
---- State no. 1 =
p = 0,
!( []p )

Loop back to state 1
```

If the temporal formula is valid, as in the first invocation of `valid`, then a message is printed. If it is not, as in the second invocation, then a counter example is printed. The counter example is a sequence of states which demonstrates why the formula is not valid. The counter example is annotated with the evaluation of all the subformulas which have a principle operator which is temporal. Note that the states of the model only consists of the values of the propositional variables (in this case, the variable `p`). The evaluation of the temporal formulas is only printed to help the user understand why complex temporal formulas are not valid.

Previously we checked that the formula $\Box\, p \rightarrow \Diamond\, p$ is valid. Is the formula $\Diamond\, p \rightarrow \Box\, p$ valid as well? We can check this as follows:

```
>> valid ltl(<>p -> []p);
Model checking...

*** Property is NOT VALID ***

Counter-Example Follows:

---- State no. 1 =
p = 0,
<>p,  !( []p )

---- State no. 2 =
p = 1,
<>p,  !( []p )

Loop back to state 1
```

Since in state 2, `p = 1` then $\Diamond\, p$ always holds. Since in state 1, `p = 0` then $\Box\, p$ never holds. Therefore the formula $\Diamond\, p \rightarrow \Box\, p$ is false for all states in this model. In particular it is false in state 1, and that is why $\Diamond\, p \rightarrow \Box\, p$ is not valid.

Note that validity requires that the formula holds on the *first* state of all models. It may or may not hold on the other states of the model — this does not affect its validity.

TLV does not have a special routine for checking satisfiability, however, a formula is satisfiable iff its negation is not valid. Therefore it is easy to check for satisfiability: just check for the validity of the negation of the formula, and reverse the result. See the tutorial for additional examples for checking validity and satisfiability.

10.2.2 Model Checking

The following command perform model checking of LTL formulas:

- `mc ltl(temporal_formula)[, tsn]` — general model checking.
- `mcseq ltl(temporal_formula)` — model check sequential systems. The problem with such systems is that they terminate, and thus their executions are finite. The algorithms for model checking require infinite executions. The `mcseq` command uses a modified version of the transitions system, which adds idling transitions for all states which have no successors. This command is typically used in an educational setting, and when SPL programs are loaded.

For example, verifying mutual exclusion can simply be done as follows:

```
>> mc ltl( [] !(proc1.state = Critical & proc2.state2 = Critical));
Model checking...

*** Property is VALID ***
```

As in the command `valid` for checking validity, if the property is not valid over the program then a counter example is printed.

10.3 CTL

Model checking CTL formulas is exactly like model checking LTL formula, (`mcseq` does not accept CTL). Instead of qualifying the formula as LTL they should be qualified as CTL. See section 2.16.2 for the syntax of CTL formulas. For example:

```
>> mc ctl( AG !(proc1.state = Critical & proc2.state2 = Critical));
```

Currently, model checking CTL formulas does not produce counter examples.

10.4 CTL*

First we define the model of *fair computation structures,* which is the semantical model for CTL* formulas. Next, we present the syntax and semantics of CTL* with past operators.

10.4.1 Fair Computation Structures

Let V be a finite set of variables. A *fair computation structure* over V is a tuple $\mathcal{K}_V : \langle S, S_0, R, J, C\rangle$, consisting of the following components.

- S : A (possibly infinite) set of all states over V.
- $S_0 \subseteq S$: A subset of *initial states*.
- $R \subseteq S \times S$: A transition relation, relating a state $s \in S$ to its $\mathcal{K}$-successor $s' \in S$.
- $J = (K_1, \ldots, K_k)$: A set of justice sets, where $K_i \subseteq S$ for every $i \in [1..k]$.
- $C = (\langle r_1, t_1\rangle, \ldots, \langle r_m, t_m\rangle)$: A set of compassion sets, where $r_i, t_i \subseteq S$ for every $i \in [1..m]$.

The set of justice and compassion sets is denoted the *fairness set*. Let $\pi : s_0, s_1, \ldots$ be an infinite sequence of states, and S be a set of states. We say that j is an S-position in π if $\pi_j \in S$, where π_j is the state at position j of π. Let $\mathcal{K}$ be a fair computation structure. The sequence $\pi = s_0, s_1, \ldots$ is said to be a *path in* $\mathcal{K}$[1] if it satisfies the following requirements:

- *Initiality:* $s_0 \in S_0$.
- *Consecution:* $(s_i, s_{i+1}) \in R$, for every $i \geq 0$.
- *Justice:* For every $K \in J$, π contains infinitely many K-positions,
- *Compassion:* For every $\langle r, t \rangle \in C$, if π contains infinitely many r-positions, it must contain infinitely many t-positions.

10.4.2 The Logic CTL*

There are two types of formulas in CTL*: *State formulas* which are interpreted over states and *path formulas* which are interpreted over paths. Let $\mathcal{P}$ be a finite set of propositions. The syntax of a CTL* formula is defined inductively as follows.
State formulas:

- Every proposition $p \in \mathcal{P}$ is a state formula.
- If p is a *path formula*, then $E_f p$ and $A_f p$ are state formulas.
  We refer to E_f and A_f as *path quantifiers*.
- If p and q are state formulas then so are $\neg p$ and $p \vee q$.

Path formulas:

- Every state formula is a path formula.
- If p and q are path formulas then so are $\neg p$, $p \vee q$, $\bigcirc p$, $p \mathcal{U} q$, $\ominus p$ and $p \mathcal{S} q$.

CTL* is the set of state formulas generated by the above rules.

The semantics of a CTL* formula p is defined with respect to a fair computation structure $\mathcal{K}$ over the vocabulary of p. The semantics is defined inductively as follows.
State formulas are interpreted over states in $\mathcal{K}$. We define the notion of a CTL* formula p holding at a state s in $\mathcal{K}$, denoted $(\mathcal{K}, s) \models p$, as follows:

- For an assertion p,
  $(\mathcal{K}, s) \models p \iff s \models p$
- $(\mathcal{K}, s) \models \neg p \iff (\mathcal{K}, s) \not\models p$
- $(\mathcal{K}, s) \models p \vee q \iff (\mathcal{K}, s) \models p$ or $(\mathcal{K}, s) \models q$
- $(\mathcal{K}, s) \models E_f p \iff (\mathcal{K}, \pi, j) \models p$ for some path $\pi \in \mathcal{K}$ and position j satisfying $\pi_j = s$.

Path formulas are interpreted over a path in $\mathcal{K}$. We define the notion of a CTL* formula p holding at position $j \geq 0$ of a path π in $\mathcal{K}$, denoted $(\mathcal{K}, \pi, j) \models p$, as follows:

[1] A path in a fair computation structure is defined as a fair path. A CTL* path formula is interpreted over a path in the structure, which is assumed to be a fair path.

- For an assertion p,
  $(\mathcal{K}, \pi, j) \models p \iff (\mathcal{K}, s) \models p$, for $s = \pi_j$
- $(\mathcal{K}, \pi, j) \models \neg p \iff (\mathcal{K}, \pi, j) \not\models p$
- $(\mathcal{K}, \pi, j) \models p \vee q \iff (\mathcal{K}, \pi, j) \models p$ or $(\mathcal{K}, \pi, j) \models q$
- $(\mathcal{K}, \pi, j) \models \bigcirc p \iff (\mathcal{K}, \pi, j+1) \models p$
- $(\mathcal{K}, \pi, j) \models p \mathcal{U} q \iff (\mathcal{K}, \pi, k) \models q$ for some $k \geq j$ and $(\mathcal{K}, \pi, i) \models p$ for every i, $j \leq i < k$
- $(\mathcal{K}, \pi, j) \models \ominus p \iff (\mathcal{K}, \pi, j-1) \models p$
- $(\mathcal{K}, \pi, j) \models p \mathcal{S} q \iff (\mathcal{K}, \pi, k) \models q$ for some $k \leq j$ and $(\mathcal{K}, \pi, i) \models p$ for every i, $j \geq i > k$

In the case that $(\mathcal{K}, s_0) \models p$, we say that the state formul a p *holds* on $\mathcal{K}$, and denote it by $\mathcal{K} \models p$.

The notion of temporal validity (satisfiability) of a CTL* formula is similar to that of an LTL formula. The notion of $\mathcal{D}$-satisfiability (validity) is also similar, as follows. Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS. Then $\mathcal{D}$ generates a fair computation structure $\mathcal{K}_V : \langle S, S_0, R, J, C \rangle$ as follows:

- The states of $\mathcal{K}$ are all the V-states of $\mathcal{D}$.
- $s \in S_0$ iff $s \models \Theta$.
- $(s_i, s_{i+1}) \in R$ iff $\rho(V_i, V_{i+1}) = \mathrm{T}$.
- $s \in K_i$ iff $s \models J_i$.
- $s \in r_i$ (t_i) iff $s \models p_i$ $(s \models q_i)$.

A CTL* formula p is $\mathcal{D}$-*valid* if it holds over all initial states of $\mathcal{D}$.

10.4.3 Model Checking

The command for model checking CTL* formulas is:

- `mctls ltl(temporal_formula)`

The syntax for path formulas is identical to that of LTL. However, for state formulas, two operators have been added, EL and AL, which stand for E_f and A_f respectively. They are used like functions, i.e. the operand of these must always appear in parenthsis.

There are two odd points in this syntax. First that the parameter is surrounded by "ltl" rather than something like "ctl*". The "ltl" keyword is used so that TLV can parse the LTL syntax. However, the syntax of CTL* is almost identical, so the current implementation uses the "ltl" keyword to enclose formulas in either logic. The second odd point is the naming of the state formula EL and AL. This is because all the other operators (like E, EF) are already taken by CTL.

For example, the following command tests if there exists a path such that in some state along the path proc1.state = entering. Printing counter examples is not yet implemented for CTL* model checking.

```
mctls ltl( EL( <> (proc1.state = entering) ) );
```

Chapter 11

Deductive Rules

11.1 Invariance

Invariants are properties of the form $\Box\, p$. There are three rules for proving invariance: binv, inv and invx. The syntax for calling them is:

```
binv p, [, tsn]
invx p, varphi [, tsn]
inv  p, varphi [, tsn]
```

Where tsn is an optional parameter specifying the transition system number which is to be verified.

Following are the premises which need to be shown valid for each of the rules. Θ, ρ are the initial condition and the transition relation of the transition system, respectively. p is the property whose invariance we want to prove, and φ is a strengthening assertion.

Rule BINV

$$\begin{array}{ll} \text{I1}: & \Theta \rightarrow p \\ \text{I2}: & \rho \wedge p \rightarrow p' \\ \hline & \Box\, p \end{array}$$

Rule INVX

$$\begin{array}{ll} \text{I1}: & \Theta \rightarrow p \\ \text{I2}: & \rho \wedge p \wedge \varphi \rightarrow p' \\ \hline & \Box\, p \end{array}$$

Rule INV

$$\begin{array}{ll} \text{I1}: & \Theta \rightarrow \varphi \\ \text{I2}: & \rho \wedge \varphi \rightarrow \varphi' \\ \text{I3}: & \varphi \rightarrow p \\ \hline & \Box\, p \end{array}$$

Rule BINV is the basic invariance rule. Premise I1 is the base of the induction, and I2 is the inductive step. However, a property may be an invariant of the program, but not be strong enough to carry a proof of its invariance. Rule INVX uses an additional invariant φ to carry the proof. It

is assumed that φ has already been proved to be an invariant of the program by some other means (for example, by previous invocation of one of the deductive invariance rules, or by model checking techniques).

Rule INV also uses the auxiliary invariant φ, but its variance is proved rather than assumed. Rule INV is similar to BINV, but it uses φ instead of p to perform the induction, and it proves the invariance p by requiring that φ implies p.

The following example uses rule inv for verifying mutual exclusion:

```
>> inv p, p_strong;
Checking Premise I1
Premise I1 is valid. Checking Premise I2.
Premise I2 is not valid. Counter-example =
y = 1,0             proc[1].loc = 2,3   proc[2].loc = 0,0   proc[3].loc = 3,3
```

In this case, the rule did not succeed, and a counter example was printed. The counter example is an assignment which satisfies the negation of the premise.

11.2 Response

set of rules for proving *response* properties, i.e., properties that are specifiable by LTL formulas of the form $p \Rightarrow \Diamond q$.

Chapter 12

Floyd's Method

Floyd's method [7][8] for verifying partial correctness of computational programs requires the user to find a set of program locations, and attach an assertion to each of them.

This set of locations is called the *cut points* set. The initial and final program locations should be in the cut points. The cut points should cut all loops of the program.

The assertion related to each cut point characterizes the states at this cut point whenever it is reached.

Given a cut point set and corresponding assertions, TLV can verify that the assertions hold for all cut points which are reachable from the initial state.

To invoke the Floyd method we use the chk_inv command. As parameters we provide an array of cuts and an array of expressions which correspond to the cuts. The chk_inv command verifies that the cuts indeed cut all loops and checks whether all expressions indeed hold of their corresponding cuts. The syntax is

```
chk_inv cut_array, expression_array, n;
```

Where `cut_array` is the array of cuts, `expression_array` the array of expressions and `n` the length of the arrays. The length of both arrays should be at least n, but could be more (the cuts in indexes larger than `n` will not be checked). If the length of one of the arrays is smaller than `n` then the chk_inv will fail.

The TLV tutorial contains many examples of using Floyd's method.

Chapter 13

Abstraction

Bibliography

[1] K. L. McMillan. *The SMV System Draft.* Carnegie-Mellon Univerisity, 1992.

[2] K.L McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

[3] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), August 1986.

[5] B. Liberman. Temporal prototype verification system. Implementation. SPL Compiler. Master's thesis, Weizmann Institute, 1996.

[6] Armin Biere, Alessandro Cimatti, Edmond Clarke, and Zhu Yunshan. Symbolic model checking without bdds. In *To be published*, 1999.

[7] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[8] Z. Manna and R. Waldinger. Is 'sometime' sometimes better than 'always'?: Intermittent assertions in proving program correctness. *Comm. ACM*, 21:159–172, 1978.

Index

|, 19
==, 33
&, 19
&&&, 19
->, 19
<->, 19

and, 34
append, 37
assign, 25

bdd, 6
bdd2cnf, 38
bdd2cnfneg, 38
bdd2prop, 38
bdd2reset, 39
Break, 30

Call, 13, 27
case, 20
Chktime, 35
command
    append, 37
    Chktime, 35
    Exit, 26
    form, 38
    funcs, 35
    garbage, 35
    Let, 12, 25
    Let ', 32
    Load, 26
    log, 37
    lorder, 36
    numstate, 35
    Print, 26
    Print ', 32
    procs, 35
    Quit, 26
    rcopy, 41
    read_num, 36
    read_string, 36
    reorder, 36
    reorder_bits, 36
    rprint, 41
    Settime, 35
    sorder, 36
    Stats, 35
    sz, 35
    vform, 38
COMPASSION, 49
COMPOSED, 50
computation, 49
Conditionl Statement, 53
constructor loop, 52
constructor loop expressions, 23
constructor loops, 51
Continue, 30
CTL, 44
ctl*, 81
    model checking, 80

default in, 54
delvar, 13
dotty, 39
Dump, 39
dynamic scope, 28
dynamic variables, 12
    predefined, 17
dynamicvariables, 16

exist, 25
Exit, 26

FAIRNESS, 49

Fix, 32
Floyd, 85
For, 32
forall, 21
form, 38
forsome, 21
frsat, 23
fsat, 23
Func, 27
funcs, 35
function
    and, 34
    assign, 25
    exist, 25
    iff, 34
    implies, 34
    op, 34
    ops, 34
    or, 34
    positive, 34
    pred, 24
    prime, 24
    root, 33
    size, 25
    strlen, 33
    succ, 24
    unprime, 24
    value, 25

garbage, 35

id_of, 22
If, 14, 30
iff, 34
implies, 34
in, 21
Invariance, 15
is_equal, 20
is_false, 20
is_true, 20

JUSTICE, 49

kind, 18, 49

Let, 12, 25
Let ', 32
Load, 26
Local, 28
log, 37
lorder, 36
LTL, 42
    model checking, 79

model checking, 77
    ctl*, 80
    LTL, 79

next, 24
notin, 21
numstate, 35

op, 34
ops, 34
or, 34
OWNED, 51

parameter
    parse tree, 32
paramter
    by name, 29
parse tree, 32
    Case, 32
partial correctness, 85
positive, 34
pred, 24
prime, 24
Print, 26
Print ', 32
Proc, 13, 26
process, 14
procs, 35

Quit, 26

read_num, 36
read_string, 36
reorder, 36
reorder_bits, 36
Return, 27
root, 33
rsat, 23
Run, 27

sat, 23
savelist, 35
set_intersect, 22
set_member, 22
set_minus, 22
set_union, 22
Settime, 35
simulation, 70
size, 25
smv, 5, 46
    COMPOSED, 50
    kind, 49
    OWNED, 51
    system, 49
sorder, 36
SPEC, 49
spl, 55
Stats, 35
strlen, 33
succ, 24
support, 21
sys_num, 24
system, 49
system variables, 12
sz, 35

Temporal Logic, 42
    satisfiability, 78
    validity, 78
To, 26

union, 21
unprime, 24

value, 25
variable ordering, 6, 8, 36
variable set, 16
variables, 42
    dynamic, 12
    system, 12
vform, 38
vset, 21

While, 14, 30

Part III

Appendixes

Appendix A

SPL: Syntax

We introduce a simple concurrent programming language in which example programs will be written.

A.1 Simple Statements

These statements represent the most basic steps in the computation.

- *Skip*: A trivial do-nothing statement is

  skip.

- *Assignment*: For a list of variables $y_1, \ldots, y_k$ and a list of expressions of corresponding types $e_1, \ldots, e_k$,

  $(y_1, \ldots, y_k) \ := \ (e_1, \ldots, e_k)$

  is an *assignment* statement. For the case that $k = 1$, we write simply $y := e$.

- *Await*: For a boolean expression c,

  await c

  is an *await* statement. We refer to condition c as the *guard* of the statement. Execution of "**await** c" changes no variables. Its sole purpose is to wait until c becomes true, at which point it terminates.

The following two statements, called *communication statements*, support communication by message-passing. These statements refer to a *channel* which is a state variable of a special kind. Similar to other variables, each channel is associated with a type, identifying the type of values that can be sent via the channel.

- *Send*: For a channel α and expression e of compatible type,

  $\alpha \ \Leftarrow \ e$

  is a *send* statement. The expression e is evaluated and its value sent via channel α.

- *Receive*: For a channel α and variable u of compatible type,

  $\alpha \;\Rightarrow\; u$

  is a *receive* statement. This statement causes a message to be read from channel α and placed in variable u.

The following two statements support semaphore operations.

- *Request*: For an integer variable r,

  request r

  is a *request* statement. This statement can be executed only when r has a positive value. When executed, it decrements r by 1.

- *Release*: For an integer variable r,

  release r

  is a *release* statement. Execution of this statement increments r by 1.

A.2 Schematic Statements

The following statements provide schematic representation of segments of code that appear in programs for solving the mutual exclusion or producer-consumer problems. Typically, we are not interested in the internal details of this code but only in its overall behavior concerning termination.

- *Noncritical*:

  noncritical

  is a *noncritical* statement. This statement represents the noncritical activity in programs for mutual exclusion. This statement is not required to terminate.

- *Critical*:

  critical

  is a *critical* statement. This statement represents the critical activity in programs for mutual exclusion, where coordination between the processes is required. This statement is required to terminate.

- *Produce*: For an integer variable x,

  produce x

  is a *produce* statement. This terminating statement represents the production activity in producer-consumer programs. Its effect is to assign a nonzero value to variable x, representing the produced value.

- *Consume*: For an integer variable y,

  consume y

  is a *consume* statement. This terminating statement represents the consumption activity in producer-consumer programs. Its effect is to set variable y to zero, representing the consumption of y's previous value.

It is assumed that none of these statements modify any of the program variables, except for the production variable x and the consumption variable y, explicitly mentioned in the corresponding statements.

A.3 Compound Statements

Compound statements consist of a controlling frame applied to one or more simpler statements, to which we refer as the *children* of the compound statement.

- *Conditional*: For statements S_1 and S_2 and a boolean expression c,

  if c **then** S_1 **else** S_2

  is a *conditional* statement. Its intended meaning is that the boolean expression c is evaluated and tested. If the condition evaluates to T (true), statement S_1 is selected for subsequent execution; otherwise, if the condition evaluates to F (false), S_2 is selected. Thus, the first step in an execution of the conditional statement is the evaluation of c and the selection of S_1 or S_2 for further execution. Subsequent steps continue to execute the selected substatement.

- *One-branch-conditional* statement is a special case of the conditional statement:

  if c **then** S_1.

  Execution of this statement in the case that c evaluates to F terminates in one step.

- *Concatenation*: For statements $S_1, \ldots, S_k$,

  $S_1; \; \cdots; \; S_k$

  is a *concatenation* statement. Its intended meaning is sequential execution of statements $S_1, \ldots, S_k$ one after the other.

  With concatenation, we can define the

- *When* statement

  when c **do** S

  as an abbreviation for the concatenation **await** c; S.

- *Selection*: For statements $S_1, \ldots, S_k$,

  S_1 **or** $\cdots$ **or** S_k

  is a *selection* statement. Its intended meaning is a non-deterministic selection of a statement S_i and its execution. The first step in the execution of the selection statement selects a statement S_i, $i = 1, \ldots, k$, that is currently enabled and performs the first step in the execution of S_i. Subsequent steps proceed to execute the rest of the selected substatement, ignoring the other S_i's. If more than one of $S_1, \ldots, S_k$ is enabled, the selection is non-deterministic.

- *While*: For a boolean expression c and a statement S,

  while c **do** S

  is a *while* statement. Its execution begins by evaluating c. If c evaluates to F, execution of the statement terminates. Otherwise, subsequent steps proceed to execute S. When S terminates, c is tested again.

- *Cooperation*: For statements $S_1, \ldots, S_k$,

  $[\ell_1\colon S_1;\ \widehat{\ell}_1\colon\ ] \parallel \cdots \parallel [\ell_k\colon S_k;\ \widehat{\ell}_k\colon\ ]$

  is a *cooperation* statement. Its intended meaning is the parallel execution of $S_1, \ldots S_k$. The first step in the execution of a cooperation statement is referred to as the *entry* step. It can be conceived as setting the stage for the parallel execution of $S_1, \ldots S_k$. Subsequent steps proceed to perform steps from $S_1, \ldots S_k$. When all $S_1, \ldots S_k$ have terminated, there is an additional *exit* step that closes the parallel execution. Each parallel component is assigned an *entry label* ℓ_i and an *exit label* $\widehat{\ell}_i$, which is the location of control after execution of S_i terminates. Label $\widehat{\ell}_i$ can be viewed as labeling an empty statement following S_i. We refer to component $[\ell_i\colon S_i;\ \widehat{\ell}_i\colon\ ]$ as a *process* of the cooperation statement.

  It is important to note that in the combination

  $\big[[\ell_1\colon S_1;\ \widehat{\ell}_1\colon\ ] \parallel [\ell_2\colon S_2;\ \widehat{\ell}_2\colon\ ]\big];\ S_3,$

  execution of S_3 cannot start until both S_1 and S_2 are terminated.

- *Block*: For a statement S,

  [local declaration; S],

  is a *block* statement. Statement S is called the *body* of the block.

  A *local declaration* is a list of *declaration statements* of the form

  local variable, ..., variable: type **where** φ_i.

  The local declaration identifies the variables that are local to the block, specifies their type, and optionally specifies their initial values.

- *Grouped Statement*

  In some cases it is necessary to execute a compound statement S in one atomic step with no interference from parallel statements. To denote that S is to be executed in such atomic manner, we write $\langle S \rangle$ and refer to this statement as a *grouped* statement. It is reasonable to restrict the grouping operation to compound statements whose execution involves a bounded number of steps. Therefore, we first define the notion of a *composite statement* as follows:

  – *Skip,* assignment, *await, send,* and *receive* statements are (basic) composite statements.

  – If $S, S_1, \ldots, S_k$ are composite statements, then so are

    when c **do** S, **if** c **then** S_1 **else** S_2, $(S_1$ **or** $\cdots$ **or** $S_k)$, $(S_1;\ \ldots;\ S_k)$.

  For simplicity, we only allow composite statements that do not contain more than a single communication (*send* or *receive*) statement.

  If S is a composite statement, then $\langle S \rangle$ is a *grouped statement*.

A.4 Programs

A program P consists of a declaration followed by a cooperation statement, in which processes may be named.

$$P :: \Big[\text{declaration; } \big[P_1 :: \ [\ell_1 \colon S_1; \ \widehat{\ell}_1 \colon \ ] \parallel \ \cdots \ \parallel P_k :: \ [\ell_k \colon S_k; \ \widehat{\ell}_k \colon \ ]\big]\Big]$$

The names of the program and of the processes are optional, and may be omitted. We refer to the cooperation statement as the *body* of the program.

A declaration consists of a sequence of *declaration statements* of the form

mode variable, ..., variable: type **where** φ_i.

Each declaration statement identifies the mode and type of a list of variables and, optionally, specifies constraints on their initial values.

The *mode* of each declaration statement may be one of the following

in — Specifies variables that are inputs to the program.

local — Specifies variables that are local to the program. These variables are used in the execution of the program, but are not recognized outside the program.

out — Specifies variables that are outputs of the program.

A statement S may refer to a variable only if the variable is declared at the head of the program or at the head of a block containing S. The program is not allowed to modify (i.e., assign new values to) variables that are declared as **in** variables. The distinction between modes **local** and **out** is mainly intended to help in the understanding of the program and has no particular formal significance.

The optional assertion φ_i imposes constraints on the initial values of the variables declared in this statement. For *local* and *output* variables it is restricted to be of the form $y = e$, specifying an initial value. For *in* variables, it may be any constraint on the range of values expected for these variables[1]. We refer to the variables declared in the program as the *program variables*.

Channels are declared by a *channel declaration* that may assume one of the two forms:

mode $\alpha_1, \alpha_2, \ldots, \alpha_n$: **channel of** type.
mode $\alpha_1, \alpha_2, \ldots, \alpha_n$: **channel** [1..] **of** type.

These declarations identify $\alpha_1, \ldots, \alpha_n$ as channels through which messages of the specified type can be sent and received. Channels defined by a declaration of the first form are called *synchronous channels*. Such channels have no buffering capacity and a communication can occur only when both sender and receiver execute their statements at the same step.

Channels defined by a declaration of the second form are called *asynchronous channels*. These channels have unbounded buffering capacity. This means that a sender can send an unbounded number of messages before the receiver reads the first one. Variables of type *channel of channel* are not allowed.

[1] The current version of the SPL compiler does not allow initialization of *in* variables.

Appendix B

SPL: Semantics

Each program defines a fair transition system. We will show how to interpret each of the components of a fair transition system for a given program.

B.1 System Variables

The system variables V consist of the program variables $Y = \{y_1, \ldots, y_n\}$, declared in the program, and a set of integer valued control variables $\overline{\pi} : \pi_1, \ldots, \pi_m$, one for each process of the program.

The program variables Y range over their respectively declared data domains. For each channel α of a declared type t, we include a system variable α whose type is a list of type t. This variable holds the list of pending messages, i.e., messages sent on α but not yet received from it.

For each process P_i, $i = 1, \ldots, m$, control variable π_i ranges over a finite set of integers which are the indices of the locations within P_i, plus the special value -1. A nonnegative value of $\pi_i = j \geq 0$ in a state indicates that control of process P_i is currently at ℓ_j (or m_j if the locations of P_i have the form $m_0, m_1, \ldots$). A negative value of π_i indicates that process P_i is currently *passive* (suspended). This is, for example the situation of all nested processes in the initial state.

For a label ℓ_j belonging to process P_i, we define the control predicate at_ℓ_j by

at_ℓ_j: $\pi_i = j$.

B.2 Initial Condition

Consider a program

$$P :: \left[\text{declaration; } \left[P_1 :: \ [\ell_1 \colon S_1;\ \widehat{\ell}_1\colon\ ] \parallel \cdots \parallel P_k :: \ [\ell_k \colon S_k;\ \widehat{\ell}_k\colon\ ]\right]\right].$$

Let φ denote the data-precondition of the program, obtained by taking the conjunction of all the assertions φ_i that appear in the *where* clauses of the declaration. The initial condition Θ for program P is defined as

Θ: $\pi_1 = \cdots = \pi_k = 1 \ \wedge \ \pi_{k+1} = \cdots = \pi_m = -1 \ \wedge \ \varphi$.

This implies that the first state in an execution of the program begins with the control set to the entry locations of the top-level processes, after all the initialization of the local and output variables has been performed. Note that all nested processes are suspended in the initial state.

For a program P, none of whose declaration statements has a *where* part, i.e., $\varphi = \mathrm{T}$, we simply define

Θ: $\pi_1 = \cdots = \pi_k = 1 \ \wedge \ \pi_{k+1} = \cdots = \pi_m = -1.$

B.3 Transitions

To ensure that every state has some transition enabled on it, we uniformly include the idling transition τ_I in the transition system corresponding to each program. The transition relation for τ_I is

ρ_I: $V' = V$.

We proceed to define the transition relations for the transitions associated with each of the statements in the language. To express the movement of control effected by transitions, we use the abbreviation

$$move(\ell_j, \ell_r): \quad \pi_i = j \ \wedge \ \pi'_i = r \ \wedge \ \bigwedge_{t \neq i} \pi'_t = \pi_t,$$

For labels ℓ_j and ℓ_r belonging to process P_i. This formula describes the movement of control from ℓ_j to ℓ_r. It follows from the definitions that, if $j \neq r$, then $move(\ell_j, \ell_r)$ implies that $\overline{\pi}$ satisfies $at_\ell_j \wedge \neg at_\ell_r$ while $\overline{\pi}'$ satisfies $at_\ell_r \wedge \neg at_\ell_j$.

Typically, every transition modifies only few of the state variables. We therefore introduce, for a subset U of the state variables, $U \subseteq V$, the notation

$$pres(U): \quad \bigwedge_{u \in U} (u' = u).$$

The formula $pres(U)$ states that all variables in U preserve their value at the current step.

In the following, we represent a statement S with label ℓ_j and post-label ℓ_r in the form $[\ell_j\!: S;\ \ell_r\!:\ ]$.

Simple Statements

- *Skip.* With the statement

  ℓ_j : **skip**; ℓ_r: ,

  we associate a transition τ_{ℓ_j}, whose transition relation ρ_{ℓ_j} is given by

  ρ_{ℓ_j}: $move(\ell_j, \ell_r) \ \wedge \ pres\big(V - \{\overline{\pi}\}\big)$.

  This transition modifies no program variables and simply moves from ℓ to ℓ_r.

- *Assignment.* With the statement

  ℓ_j : $\overline{u} := \overline{e}$; ℓ_r : ,

  we associate a transition τ_{ℓ_j}, whose transition relation ρ_{ℓ_j} is given by

  ρ_{ℓ_j}: $move(\ell_j, \ell_r) \ \wedge \ \overline{u}' = \overline{e} \ \wedge \ pres\big(V - \{\overline{\pi}, \overline{u}\}\big)$.

- *Await.* With the statement

  ℓ_j : **await** c; ℓ_r: ,

we associate a transition τ_{ℓ_j}, whose transition relation ρ_{ℓ_j} is given by

ρ_{ℓ_j}: $\quad move(\ell_j, \ell_r) \ \wedge \ c \ \wedge \ pres\big(V - \{\overline{\pi}\}\big)$.

The transition τ_{ℓ_j} is enabled only when control is at ℓ and condition c holds. When taken, it moves from ℓ_j to ℓ_r.

The following two definitions deal with communication statements over asynchronous channels.

- *Asynchronous Send.* With the *send* statement

  $\ell_j : \ \alpha \Leftarrow e; \quad \ell_r$: ,

  where α is an asynchronous channel, we associate a transition τ_{ℓ_j}, whose transition relation is given by

  ρ_{ℓ_j}: $\quad move(\ell_j, \ell_r) \ \wedge \ \alpha' = \alpha \bullet e \ \wedge \ pres\big(V - \{\overline{\pi}, \alpha\}\big)$.

  The data part of the assertion ρ_{ℓ_j} describes the new value of α as being obtained by appending the value of e to the end of α.

- *Asynchronous Receive.* With the *receive* statement

  $\ell_j : \ \alpha \Rightarrow u; \quad \ell_r$: ,

  where α is an asynchronous channel, we associate a transition τ_{ℓ_j}, whose transition relation is given by

  ρ_{ℓ_j}: $\quad move(\ell_j, \ell_r) \ \wedge \ |\alpha| > 0 \ \wedge \ u' = hd(\alpha) \ \wedge \ \alpha' = tl(\alpha) \ \wedge \ pres\big(V - \{\overline{\pi}, u, \alpha\}\big)$.

  The data part of ρ_{ℓ_j} states that transition τ_ℓ is enabled only if channel α is currently nonempty and, when executed, its effect is to deposit the first element (head) of α in u and to remove this element (retaining the tail) from α.

The final definitions deal with semaphore statements.

- *Request.* With the statement

  ℓ_j : **request** r; $\quad \ell_r$: ,

  we associate a transition τ_{ℓ_j}, whose transition relation is given by

  ρ_{ℓ_j}: $\quad move(\ell_j, \ell_r) \ \wedge \ r > 0 \ \wedge \ r' = r - 1 \ \wedge \ pres\big(V - \{\overline{\pi}, r\}\big)$.

  Thus, this statement is enabled when control is at ℓ and r is positive. When executed it decrements r by 1.

- *Release.* With the statement

  ℓ_j : **release** r; $\quad \ell_r$: ,

  we associate a transition τ_{ℓ_j}, whose transition relation is given by

  ρ_{ℓ_j}: $\quad move(\ell_j, \ell_r) \ \wedge \ r' = r + 1 \ \wedge \ pres\big(V - \{\overline{\pi}, r\}\big)$.

  This statement increments r by 1.

Schematic Statements

- *Noncritical.* With the statement

  $\ell_j :$ **noncritical**; ℓ_r: ,

  we associate a transition τ_{ℓ_j}, whose transition relation is given by

  $\rho_{\ell_j}\!: \quad move(\ell_j, \ell_r) \ \wedge\ pres\big(V - \{\overline{\pi}\}\big)$.

  Thus, the only observable action of this statement is to terminate. The situation that execution of the noncritical section does not terminate is modeled by a computation that does not take transition τ_{ℓ_j}. This is allowed by excluding τ_{ℓ_j} from the justice set.

- *Critical.* With the statement

  $\ell_j :$ **critical**; ℓ_r: ,

  we associate a transition τ_{ℓ_j}, whose transition relation is given by

  $\rho_{\ell_j}\!: \quad move(\ell_j, \ell_r) \ \wedge\ pres\big(V - \{\overline{\pi}\}\big)$.

  The observable action of the critical statement is also to terminate.

- *Produce.* With the statement

  $\ell_j :$ **produce** x; ℓ_r: ,

  we associate a transition τ_{ℓ_j}, whose transition relation is given by

  $\rho_{\ell_j}\!: \quad move(\ell_j, \ell_r) \ \wedge\ x' \neq 0 \ \wedge\ pres\big(V - \{\overline{\pi}, x\}\big)$.

  The observable action of the produce statement is to assign a nonzero value to variable x.

- *Consume.* With the statement

  $\ell_j :$ **consume** y; ℓ_r: ,

  we associate a transition τ_{ℓ_j}, whose transition relation is given by

  $\rho_{\ell_j}\!: \quad move(\ell_j, \ell_r) \ \wedge\ y' = 0 \ \wedge\ pres\big(V - \{\overline{\pi}, y\}\big)$.

  The observable action of the consume statement is to set variable y to zero.

Compound Statements

- *Conditional.* With the statement

  $\ell_j :$ $\big[$**if** c **then** $\ell_1\!: S_1$ **else** $\ell_2\!: S_2\big]$; ℓ_r: ,

  we associate a transition τ_{ℓ_j}, whose transition relation is given by $\rho_{\ell_j}\!: \rho^{\mathrm{T}}_{\ell_j} \vee \rho^{\mathrm{F}}_{\ell}$, where

  $\rho^{\mathrm{T}}_{\ell_j}\!: \quad move(\ell_j, \ell_1) \ \wedge\ c \ \wedge\ pres\big(V - \{\overline{\pi}\}\big)$

  $\rho^{\mathrm{F}}_{\ell_j}\!: \quad move(\ell_j, \ell_2) \ \wedge\ \neg c \ \wedge\ pres\big(V - \{\overline{\pi}\}\big)$.

  Relation $\rho^{\mathrm{T}}_{\ell_j}$ corresponds to the case that c evaluates to true and execution proceeds to ℓ_1, while ρ^{F}_{ℓ} corresponds to the case that c evaluates to false and execution proceeds to ℓ_2.

  For the one-branch conditional

  $\ell_j :$ $\big($**if** c **then** $\ell_1\!: S_1\big)$; ℓ_r: ,

the two disjuncts of the transition relation are

$\rho^{\mathrm{T}}_{\ell_j}$: $\quad move(\ell_j, \ell_1) \ \wedge \ c \ \wedge \ pres\big(V - \{\overline{\pi}\}\big)$

$\rho^{\mathrm{F}}_{\ell_j}$: $\quad move(\ell_j, \ell_r) \ \wedge \ \neg c \ \wedge \ pres\big(V - \{\overline{\pi}\}\big)$.

- *While.* With the statement

  $\ell_j: \ \big(\mathbf{while}\ c\ \mathbf{do}\ (\ell_1\colon S)\big); \quad \ell_r\colon,$

  we associate a transition τ_{ℓ_j} (with two modes), whose transition relation is given by $\rho_{\ell_j}\colon \rho^{\mathrm{T}}_{\ell_j} \vee \rho^{\mathrm{F}}_{\ell_j}$, where

  $\rho^{\mathrm{T}}_{\ell_j}$: $\quad move(\ell_j, \ell_1) \ \wedge \ c \ \wedge \ pres\big(V - \{\overline{\pi}\}\big)$

  $\rho^{\mathrm{F}}_{\ell_j}$: $\quad move(\ell_j, \ell_r) \ \wedge \ \neg c \ \wedge \ pres\big(V - \{\overline{\pi}\}\big)$.

  Note that in the case of a true c control moves, according to $\rho^{\mathrm{T}}_{\ell_j}$, from ℓ_j to ℓ_1, while in the case of a false c it moves, according to $\rho^{\mathrm{F}}_{\ell_j}$, from ℓ_j to ℓ_r.

Consider the cooperation statement:

$$\ell_j: \ \big((\ell^1_1\colon S_1;\ \ell^1_{t_1}\colon\ ) \parallel \cdots \parallel (\ell^q_1\colon S_q;\ \ell^q_{t_q}\colon\ )\big); \quad \ell_r\colon$$

belonging to process P_i, where statements $S_1, \ldots, S_q$ are identified as processes $P_{i_1}, \ldots, P_{i_q}$, respectively. To express the movement of control in and out of this statement, we introduce the abbreviations:

$$enter\big(\ell_j, \{\ell^1_1, \ldots, \ell^q_1\}\big)\colon \quad \pi_i = j \ \wedge \ \pi_{i_1} = \cdots = \pi_{i_q} = -1 \ \wedge \ \pi'_i = -1 \ \wedge \ \pi'_{i_1} = \cdots = \pi'_{i_q} = 1 \ \wedge \bigwedge_{p \notin \{i, i_1, \ldots, i_q\}} \pi'_p = \pi_p$$

$$exit\big(\{\ell^1_{t_1}, \ldots, \ell^q_{t_q}\}, \ell_r\big)\colon \quad \pi_i = -1 \ \wedge \ \pi_{i_1} = t_1 \ \wedge \ \cdots \ \wedge \ \pi_{i_q} = t_q \ \wedge \ \pi'_i = r \ \wedge \ \pi'_{i_1} = \cdots = \pi'_{i_q} = -1 \ \wedge \bigwedge_{p \notin \{i, i_1, \ldots, i_q\}} \pi'_p = \pi_p$$

- *Cooperation.* Excluding the body of the program, we associate with each *cooperation* statement,

  $\ell_j: \ \big[[\ell^1_1\colon S_1;\ \ell^1_{t_1}\colon\ ] \parallel \cdots \parallel [\ell^q_1\colon S_q;\ \ell^q_{t_q}\colon\ ]\big]; \quad \ell_r\colon$

  an *entry transition* τ^{E}_{ℓ} and an *exit transition* $\tau^{\mathrm{X}}_{\ell_j}$. The corresponding transition relations are given, respectively, by

  $\rho^{\mathrm{E}}_{\ell_j}$: $\quad enter\big(\ell_j, \{\ell^1_1, \ldots, \ell^q_1\}\big) \ \wedge \ pres\big(V - \{\overline{\pi}\}\big)$

  $\rho^{\mathrm{X}}_{\ell_j}$: $\quad exit\big(\{\ell^1_{t_1}, \ldots, \ell^q_{t_q}\}, \ell_r\big) \ \wedge \ pres\big(V - \{\overline{\pi}\}\big)$.

  The entry transition begins execution of the cooperation statement by suspending process P_i (π_i set to -1) and activating each of $P_{i_1}, \ldots, P_{i_q}$ at its initial location. The exit transition can be taken only if all the parallel statements have terminated, which is detectable by observing that $\pi_{i_p} = t_p$ for each $p = 1, \ldots, q$.

Note that the absence of entry or exit transitions associated with the body of the program is consistent with the fact that, according to the initial condition, execution starts with π_i set to the entry location of the top-level process P_i for each $i = 1, \ldots, k$.

The discerning reader may have observed that no transitions are directly associated with the concatenation, selection, or block statements. This is because each transition occurring in the execution of one of these statements can be attributed to one of their substatements.

When there is no danger of confusion, we often refer to the transition τ_{ℓ_j} simply as ℓ_j.

- *Grouped Statement* To define the transition semantics of a grouped statement $\langle S \rangle$, we first define the notion of the *data* (*transformation*) relation $\delta(S)$ associated with a composite statement S. Similar to a transition relation, the data relation $\delta(S)$ describes the relation between a state s and the state s' resulting from executing S on s. Assertion $\delta(S)$ uses the data variables $Y = V - \{\overline{\pi}\}$ in order to refer to their values in s, and a primed copy Y' of these variables to refer to their values in s'.

  There are, however, two differences between the data relation $\delta(S)$ and the transition relation ρ_{ℓ_j} for a composite statement $\ell_j : S$. The first difference is that $\delta(S)$ completely ignores the control variable π and only concentrates on the data part of the state. The second difference is that, for a compound (nonbasic) S, ρ_{ℓ_j} is only concerned with the first step in the execution of S while $\delta(S)$ captures the state transformation effected by execution of the complete S.

  The data relation $\delta(S)$ is defined inductively as follows[1]:

  - *Skip*

    $\delta(\textbf{skip})\text{:} \quad pres(Y) - \{\overline{\pi}\}$
  - *Assignment*

    $\delta(\overline{u} := \overline{e})\text{:} \quad \overline{u}' = \overline{e} \;\wedge\; pres\big(Y - \{\overline{u}\}\big)$
  - *Await*

    $\delta(\textbf{await } c)\text{:} \quad c \;\wedge\; pres(Y)$
  - *Asynchronous Send*

    $\delta(\alpha \Leftarrow e)\text{:} \quad \alpha' = \alpha \bullet e \;\wedge\; pres\big(Y - \{\alpha\}\big)$
  - *Asynchronous Receive*

    $\delta(\alpha \Rightarrow u)\text{:} \quad |\alpha| > 0 \;\wedge\; \alpha = u' \bullet \alpha' \;\wedge\; pres\big(Y - \{u, \alpha\}\big).$
  - *When*

    $\delta(\textbf{when } c \textbf{ do } S)\text{:} \quad c \;\wedge\; \delta(S)$
  - *Conditional*

    $\delta(\textbf{if } c \textbf{ then } S_1 \textbf{ else } S_2)\text{:} \quad \big(c \wedge \delta(S_1)\big) \;\vee\; \big(\neg c \wedge \delta(S_2)\big)$

    $\delta(\textbf{if } c \textbf{ then } S)\text{:} \quad \big(c \wedge \delta(S)\big) \;\vee\; \big(\neg c \wedge pres(Y)\big)$
  - *Selection*

    $\delta(S_1 \textbf{ or } \cdots \textbf{ or } S_k)\text{:} \quad \delta(S_1) \;\vee\; \cdots \;\vee\; \delta(S_k)$

[1] We have excluded definitions of δ for synchronous message passing statements, since they are not currently implemented in the compiler.

– *Concatenation*

$$\delta(S_1;\ S_2):\quad \exists \widetilde{Y}\colon \Big(\delta(S_1)(Y, \widetilde{Y}) \ \wedge\ \delta(S_2)(\widetilde{Y}, Y')\Big).$$

The definition of the data relation for $(S_1; \ldots; S_k)$ for $k > 2$ is easily derivable from that of $k = 2$. Let S be a composite statement that does not involve synchronous communication. With the grouped statement

$$\ell_j :\ \langle S \rangle;\ \ell_r :$$

we associate a transition τ_{ℓ_j}, whose transition relation is given by

$$\rho_{\ell_j}:\quad move(\ell_j,\ \ell_r\,)\ \wedge\ \delta(S).$$

B.4 The Justice Set

The set of just transitions $\mathcal{J}$ includes all transitions, excluding the idling transition τ_I and the transitions associated with *noncritical* statement.

This implies that control may stay in front of a *noncritical* statement forever, representing the situation that execution of the noncritical section does not terminate. In contrast, the transition associated with a *critical* statement is in the justice set, implying that control cannot stay forever in front of a *critical* statement and, therefore, execution of critical sections always terminates.

B.5 The Compassion Set

The set of compassionate transitions $\mathcal{C}$ includes all transitions associated with the communication and semaphore statements: *send*, *receive*, and *request*.

Note that, since the requirement of compassion is stronger than the requirement of justice, it is theoretically possible to remove these transitions from the justice set. Observe that we do not require compassion for the *release* statement.

Appendix C

Error Messages of the SPL Compiler

In this section we list and explain all the error messages that can be produced when compiling programs written in SPL. Usually error message looks like this:

```
splc: line: line-number error: explanation-of-an-error
line-which-contains-an-error
     ^ - pointer to the place where error seems to take place
```

Following is the list of all error messages:

- `variable is declared more than once` – Program has some variable which has been declared in several places.
- `IN variables can not be initialized` – This happens when some declaration statement attempts to assign an initial value to an `in` variable.
- `variable initialized more than once` – Some variable presents in initialization twice or more times.
- `declaration expression is of a wrong type` – There is a type mismatch between the declared type of a variable and the expression specifying an initial value for this variable.
- `this variable can not be declared there` – Some variable is contained in both declaration and initialization parts of some declaration.
- `redeclaration of a variable` – Variable is redeclared.
- `undeclared variable` – A reference to an undeclared variable. If a variable name has one of the keywords as prefix, this message can sometimes be generated with the wrong character position. However, the erroneous line is always displayed correctly.
- `type mismatch (must be ...)` – This message is generated when as initialization expression contains incompatible types. Or, probably there is an attempt to initialize a variable of some type by an expression of incompatible type.
- `variable has a wrong range` – This message is generated when a ranged channel variable has an incorrect range.

- `redefinition of a process` – Process defined at this part of a program was defined earlier.
- `redeclaration of a label` – Label declared at this part of a program was declared earlier.
- `process has no final location` – A process with no final location is detected.
- `type mismatch` – Incompatible types are combined in an expression or an assignment.
- `variable gets value more than once` – Some variable is contained in left-hand side of multiple assignment more than once.
- `wrong size of left/right side of assignment` – Some multiple assignment has left and right hand sides of different size.
- `wrong type of expression` – `await` has an argument of a wrong type.

Finally, there are some error messages which appear under the heading `internal error`. In principle, they should never occur. However, if such an error does occur, please contact me, so that I will be able to fix a bug which is likely to happen there. Please enclose the input program and a log of what you have done in your message.