The ERIGONE Model Checker

Software Documentation

Version 3.2.5

Mordechai (Moti) Ben-Ari
Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100 Israel
http://stwww.weizmann.ac.il/g-cs/benari/

December 14, 2012

1 Directories

ERIGONE is a single program written in ADA 2005. The main procedure is `Erigone` in the source file `erigone.adb` contained directly in the directory `src`. The source files are organized in subdirectories:

- `compile`: The PROMELA compiler and preprocessor.
- `data`: Data structures: states, symbols, transitions, stacks, hash table;
- `execute`: Execution of the algorithms for simulation and verification;
- `general`: General code for IO, command-line arguments, and configuration;
- `ltl`: Translation of LTL formulas to Büchi automata.

Library-level variables are not initialized when they are declared, but only in explicit initialization subprograms. This enables the program to be run repeatedly—perhaps from a development environment—without reloading.

2 Global and configuration data

The package `Version` declares the version information and contains the full GNU GPL copyright notice.

Global declarations are in package `Global`. All values and indices are of type `Byte` (an 8-bit byte) so this places a natural limitation on the number transitions, etc. The constant `None` is declared as the last value of `Byte` and should not be used otherwise. Each identifier is of the string subtype `Name`, and statements and strings are of the subtype `Line`; all these fixed strings are padded with blanks. The type `Byte_Array_Base` is an array of `Byte`, indexed by `Byte`, that is used for storing most data. The array is unconstrained to allow components that are smaller arrays; the subtype `Byte_Array` is constrained to the full range of the index `Byte`. The type `Four_Bytes` is used when converting an integer type to and from an array of bytes.

The configuration data are described in the user's guide. Package `Config` defines subtypes of `Byte` that are used as indices of arrays storing data and can be changed to compile and execute larger programs. Package `Config_State` defines subtypes of `Byte` that are used to constrain the size of a compressed state vector on the stack and in the hash table (see `Compress_Vectors`). They are declared in a package separate from `Config` so that changing them will have a minimal impact on the necessary recompilation. These constants are compiled into ERIGONE though a future modication could compute them from a PROMELA program as in SPIN.

Some configuration data is repeated in the `Compiler_Global`. This is done to maximize the independence of the compiler from the model checker.

3 Packages in the general subdirectory

- `Byte_IO` is an instantiation of `Ada.Text_IO.Modular_IO` for bytes.
- File extensions and a pointer to the file name are declared in `Files`.
- `Random_Numbers` constains an instantiation of `Ada.Numerics.Float_Random` and a function that converts these values to values of type `Byte`.
- Package `Options` contains the declarations of run-time options such as the modes for ERIGONE, the stack sizes and the flags controlling what data is displayed.
- The values of the options are set in `Arguments` which parses the command line.
- Package `Times` stores the start time and the times when the compilation and execution have finished; these are displayed by calling `Print_Times`.
- Package `Utilities` contains subprograms that perform string and IO processing.

4 Static data structures

4.1 Symbol table

Data declarations

Package `Symbol_Tables` stores the symbol table that is obtaining by parsing data in the automata file produced by the compiler (subprogram `Read`). A symbol consists of an identifier, its type, the offset (relative to its process) in the state vector of its value, and the byte code needed to initialize the variable. Other components are an indication if the symbol is a parameter of a `proctype` and its scope (0=global, 1=local). For arrays, there are components for type of the elements, the size of an element and the length of the array.

The package exports the global variable `Variables` which stores the number of variables in the PROMELA program. Within its body are tables for numbers, strings and `mtype` names.

The array `Frame_Starts` is indexed by the process identifier and stores the offsets within the state vector of the local data for each process.

`Channel_Table` stores data for each channel: the buffer size (zero for rendezvous channels), the offset and length of the channel within the state vector, the number of elements of each message and their types and sizes. `Message_Length` is the sum of the sizes of the elements of a message and is declared to avoid recomputing it.

For a rendezvous channel the state vector stores a single message of this is so that when backtracking is done through a receive statement, the results of the computation of the matching send statement are available.

The global variable Channels exports the number of channels. As in SPIN the channels are numbered from 1, not 0. Note the difference between a channel and a channel variable. In the declaration:

```
chan ch = [2] of { byte, byte, byte };
```

ch is a channel variable that is just a byte holding the handle (index) of the channel defined by the initializer. Channel variables can be assigned to, so the association of variables with channels can change at runtime.

Subprograms

Since declarations of local variables, as well as strings, numbers, etc., can occur when reading the transitions, the subprograms Set_... for parsing and storing them are exported. Corresponding Get_... subprograms are used for retrieving data from the tables in the package body. During the compilation of a process, the size needed for local variables is written to the automata file; when this is read, Set_Local is used to update the Frame_Starts table.

Get_Variable_Initials returns a vector of the initial values after evaluating the initializing expressions. Get_Data_Size returns to total size of the data that needs to be stored in the (compressed) state vector and is used to check that there is sufficient room.

4.2 Automata

Package Automata stores a data structure for the transitions, as well as data structures for *locations*, which are the possible transitions from the set of location counters in a state. Since the package is quite large, the subprograms for displaying transitions and locations are placed in a child package Automata.Display.

Data declarations

The types declared in the package specification are:

- Byte_Code is a record containing the opcode and operands.
- Byte_Code_Array is an array of byte codes.
- Transitions is a record containing the data for transitions: source state, target state, the source code statement and its line number, flags (atomic, end label and accept), and an array of byte codes and the number of byte codes in the array. Since transitions are frequently copied, for efficiency the two large components (the statement name and the array of byte codes) are allocated and access values used instead.

- `Location_Type` is a record with two components: a `Process` index and an index of a `Transition` in the process.
- `Location_Record` stores sets of locations; its components are:
  - The locations are stored in `Location_Array` of `Location_Array_Type` whose components are of `Location_Type`;
  - A `Count` of the locations in the array;
  - `Never_Index` is the index within `Location_Array` of the first transition of a never claim (counting from zero), or `None` is a never claim is not used.

  For example, suppose that process `P` has two transitions for its location, process `Q` has one transition, and the never claim has two transitions. If both transitions for `P` and both transitions for the never claim have the same source state, then `Get_All_Locations` (see below) will return a `Location_Record` with:

```
Count           5
Never_Index     3
Location_Array
Process    Transition
   0           0
   0           1
   1           0
   2           0
   2           1
```

The transitions are sorted with the source state as the primary key and the target state as the secondary key, except that a transition corresponding to `else` is always placed last. For verification with fairness, a null transition is added to each process; see Section 8.3 for an explanation.

The transitions are stored in an array `Program` with one component of type `Process_Type` for each process. The components of this record type are:

- `Identifier` is the name of the process;
- `Initial_State` is the initial state of the process;
- `PID` is the process id (*pid*);
- `Count` is the number of transitions in the process;
- `Is_Active` is used when assigning PIDs and to compute `_nr_pr`;
- `Copies` is the number of copies of the process and is one unless `Active[N]` is used; the `run` instruction will also increment this component;

- `Variables` is the size of the local variables;
- `Transition_List` is an array of transitions for this process.

To facilitate locating accepting states (states appearing as source states of transitions marked with the accept flag), a separate array `Accept_Table` whose components are pairs of (`Process`, `State`) is maintained.

There are four public counters and indices:

- `Processes` is the number of entries in `Program`;
- `Never` is the index of the never claim within the processes;
- `Accept_Count` is the number of entries in `Accept_Table`.

For example, if there are two processes and a never claim, then `Processes` will be 3 and `Never` will be 2 (counting from zero).

`PID_Counter`, declared in the package body, is used for assigning process identifiers.

Subprograms

The most important subprogram declared in the package is:

```
  function Get_All_Locations(S: State_Vectors.State_Vector) return Location_Record;
```

The function returns all transitions whose source state is the state of a location counter of a process in the state vector `S`. For example, in Dekker's algorithm the statement:

```
    if
    :: (turn == 1)
    :: (turn == 2) -> ...
    fi
```

gives rise to two transitions:

```
number=4,source=11,target=13,atomic=0,end=0,accept=0,line=13,statement=,
  byte code=byte_load 2 0,iconst 1 0,icmpeq 0 0,
number=5,source=11,target=7,atomic=0,end=0,accept=0,line=14,statement=,
  byte code=byte_load 2 0,iconst 2 0,icmpeq 0 0,
```

in each of the two processes. In a state where both processes are at state 11, there will be four *locations*, one for each of the two transitions in each of the two process. Of course, only one location in each process will be *executable*. During simulation, one must be *chosen*, while during verification, both must be pushed on the location stack so that the search can check both possibilities.

The function `Get_All_Locations` is in this package because it is purely structural and does not depend on the runtime state; compare this with `Get_Executable_Transitions` which does depend on the state and thus can be found in the package Execute. Similarly, Remove_Transition performs the purely structural task of removing a transition from a value of Location_Record given an index.

Subprogram Translate_LTL calls LTL.LTL_To_Automaton to translate the LTL formula into a BA. The transitions of the BA are then added as an additional process in Program.

Reading the automata file

The "object code" produced by the compiler is in a form similar to that of the output of ERIGONE. The file is parsed by procedure Read. Procedure Extract_Byte_Code parses a string containing a sequence of byte codes and is exported for parsing expressions for symbol initialization and BA transitions.

When active[N] proctype P() is used, the automata file contains the number N for the process P. Initially, one proctype is created from the data read from the file. When this is complete, Replicate is called N-1 times to create a new process, copy the transitions and allocate a new frame in the state vector for the local variables. The names of the copies are P_2, P_3, ...

After all the processes are read, *pid*'s are assigned. If there is an init process, it is given *pid* 0. Then, all the processes are assigned *pid*'s in ascending order. When run is executed, its *pid* is assigned from the global variable PID_Counter.

When a verification backtracks over a run instruction, the allocation of the process should be undone but this is too hard to implement. Instead, the instruction is ignored. This should not cause a problem if run is only used in the pattern where all the operators appear within an atomic construct in the init process, because then the only backtracking is within the never claim and the process remains allocated.

5 Runtime data structures

5.1 State vectors

Package State_Vectors declares the type State_Vector with these components:

- Process is an array whose compoments are the current location counters of the processes;
- Variable is an array whose compoments are the current values of the variables, as well as the contents of buffered channels;
- Inner is a flag that is true for states in a nested search for an acceptance cycle (this is called toggle on pp. 179–181 of *SMC*).
- Fair is a byte that stores the copy of the state when checking fairness.
- Atomic is the process executing an atomic set of transitions or None.

The byte arrays of the type `State_Vector` are of the constrained subtype `Byte_Array` with 256 components.

5.2 Compressed vectors

For storing state vectors on the stack and in the hash table, *compressed* state vectors are used, where the corresponding arrays `Process` and `Variables` are declared with index subtypes `Process_Index` and `Variable_Index` from package `Config_State`. Functions to `Compress` and `Expand` state vectors are provided.

Type `Compressed_Vector` is declared as *private* in package `Compress_Vectors`. This ensures that very few packages in ERIGONE are semantically dependent on `Config_State`.

5.3 Stacks

There are two stacks: `State_Stack` for the state vectors and `Location_Stack` for the transitions to be tried in the depth-first search. (The pseudocode in Chapter 8 of *SMC* uses recursion instead of a location stack.) The stacks themselves are arrays that are allocated at run-time so that their sizes can be given as command-line arguments. Both stack packages record their maximum size for displaying runtime statistics.

Since the state stack is declared in the package body of `State_Stack`, units that are dependent on the specification of the package need not be recompiled if the compressed state vectors are modified.

The function `On_Stack` is used when verifying acceptance to see if a state exists on the stack (Section 8.2).

The location stack stores values of type `Location_Item` with components:

- `L` of type `Location_Type` (the `Process` and `Transition` indices of the location);
- `Never` is the current location within the never claim (see the discussion of the synchronous product in Section 8);
- `Visited` is a flag that is set to true when a location has been used in the depth-first search. Locations that have been visited define the trail which is written to a file by `Put_Trail` if a verification fails;
- The flag `Last` is set to true when the last location from a state has been tried.

5.4 Hash table

Package `Hash_Tables` uses an array of access types to linked lists of buckets that store (compressed) state vectors. The hash function used is FNV 1a, because it is easy to implement and efficient. See the Wikipedia page for *Fowler-Noll-Vo hash function*, or the webpage `http://isthe.com/chongo/tech/comp/fnv/`.

The hash table is an access to an array whose index is a modular type, constrained to a power-of-2 subrange of 2**32. Since an access value is used, the hash table can be allocated at run time, according to the argument -lhN.

The procedure Put_State returns a Boolean flag Inserted to indicate if the state was inserted into the set or not (because it was already there).

The package maintains counts of the number of states stored and matched, as well as a count of the numbe of collisions encountered when hashing.

Since the package needs to access the full type declaration of Compressed_Vector, it is a child package of Compress_Vectors. An additional package, also called Hash_Tables is declared at the library level and simply renames the subprograms of the child package. This is done so that the verification algorithms are not dependent on the representation of the compressed state vectors.

6 Execution of the model checking

Package Execute is the parent of all the packages that perform the simulation and verification. Its only public declaration is the procedure Run which calls the single public declaration of either either Execute.Simulate or Execute.Verify. The global variables declared in the private part are:

- Atomic is set to the index of a process that is executing an atomic statement (this is the variable exclusive in *SMC*, pp. 160–161);
- Current holds the current state of the simulation or verification;
- End_State of type End_Type is used to return the end state: valid, invalid or termination of the never claim;
- Handshake holds the current channel index when executing a rendezvous (see *SMC*, pp. 555–556);
- L_Rec of type Automata.Location_Record is used to store the set of locations (transitions) from the current state.

The package body contains step counters that are incremented as each transition is simulated or as each transition is taken during a verification.

Procedure Get_Executable_Transitions returns a set of locations that are executable in the current state. First, the set of all locations whose source is in the current state is obtained from Get_All_Locations in package Automata. Then, non-executable transitions are removed by considering:

- If End_Label is set: if the transition is for the never claim, the verification has found a counterexample; if the transition is for the halt statement at the end of a process, it can be removed.

- If the transition is for a `proctype` that has not yet been activated, it is removed.
- The values of the variables in the state that might make a transition non-executable, in which case it is removed.
- For a buffered channel, a send statement is not executable if the channel is full and a receive statement is not executable if the channel is empty. For receive statements which have an argument that is not a variable, the state vector is copied to a temporary varible and `Evaluate` is called. `Evaluate` interprets the byte code for loading constants and evaluating expressions (within `eval`), and the receive instruction is executed, although it just modifies the temporary state vector.
- For a rendezvous channel, if `Handshake` is zero, a rendezvous is *not* in progress. If that is the case, send is executable (provided that there is a receive statement for the same channel), and, when it is executed, it sets `Handshake` to its channel index. If `Handshake` is non-zero, *no* statement is executable, except for receive statements on that channel. When executed, the receive statement then resets `Handshake` to zero. This method of implementing rendezvous is described on pp. 158–160 and 555–556 of *SMC*.
- An `else` transition may be taken only if no other transitions for its process are executable; if not, the `else` transition is removed.
- If `Atomic` contains the index of a process and if there are executable transitions for that process, all other transitions must be removed.

Finally, the procedure also checks the end-state status: if there are no executable transitions, the end state is valid if and only if the all locations are labeled `end`.

Procedure `Execute_At` executes a selected transition in the current state by calling the interpreter, while procedure `Execute_Never` changes the location of the never claim.

6.1 Interpreting statements and expressions

Package `Execute.Statement` is the interpreter for the stack-based byte code. It contains the procedure `Evaluate` that receives the `Current` state vector and the byte code to be executed. The parameter `Process_ID` is used as an index in the frame table containing the offsets of the local variables; it is also used to implement the predefined read-only variable `_pid`. For a statement that stores a value, `Evaluate` updates the state vector. It always returns a `Result` that can be used to decide if an expression is executable.

For a channel, the send instruction pops the values of its arguments from the stack to an array `Message_Buffer`. From here it is copied to the first position in the channel buffer in the state vector for FIFO send; for sorted send, a search is made for the correct position to place the message.

Receiving a value from a channel is similar if the arguments are variables. If not, the value of the argument in the stack must equal the value of the corresponding field in a message in the channel buffer. This is implemented my creating an array Is_Variable of flags with an element for each argument; it is built when the arguments of the receive instruction are interpreted. An argument is considered to be a variable if it is pushed onto the stack by a load_address instruction; other instructions that push values indicate values. When the receive instruction is interpreted, it is now a simple matter to check the flags in order to decide whether to copy the message field to the variable (unless this is a poll instruction) or to check for equality. For FIFO receive, just the first message is checked, while for random receive, the same code is executed in a loop for all messages. The channel buffer is "closed up" to remove the matched message; if a matched message is not to be removed (ch?<args> or ch?[args]), this step is skipped.

6.2 Byte code instructions

The byte codes are declared in the package specification Compiler_Declarations. In addition to the enumeration type, it contains two constant arrays indexed by the byte codes. The array Executable indicates whether the instruction is Always executable (a store instruction, for example), or whether it indicates an Expression or Channel instruction that must be evaluated to determine executability. The array Data_Size is used to quickly get the size of a load or store instruction: 1 for bytes, 2 for short and 4 for integers.

7 Simulation

Package Execute.Simulate performs a simulation. Get_Executable_Transitions from the parent package Execute is called and one executable transition is chosen according to the options: random, interactive or guided. Procedure Execute_At is called to execute the chosen transition. Exception Termination_Error is raised when the simulation terminates, either normally in a valid end state, or abnormally because of errors like an invalid end state or a false assertion. The exception is handled and the reason for the termination displayed.

8 Verification

Package Execute.Verify is the parent of two child packages that contain the code for verification in safety mode and acceptance mode (with or without fairness).

The visible part of the package contains the single subprogram Verify, while the private part contains subprograms that are used in both modes of verification.

Function Is_Accept_State returns true if some process is in a state that is accepting.

Get_And_Push_Transactions calls Get_Executable_Transactions from the parent package. If: (1) there are no executable transactions, or (2) the only executable transactions are from the never claim, or (3) the never claim is terminated, then the state stack is popped (after checking the end state in safety mode). Otherwise, Push_All_Transitions pushes all locations in the location record onto the location stack. The last location (which is pushed first) has `Last` set to true so that we know when to terminate the search in this state. All locations have `Visited` set to false.

For search diversity, the array of locations is read cyclically starting from a random index.

When there is a never claim, the *synchronous product* must be pushed: Each executable location of the program is pushed once for each executable transition of the claim.

The private part also contains the declaration of the variable `Unfolded` that is used when verifying with fairness; see Section 8.3 below.

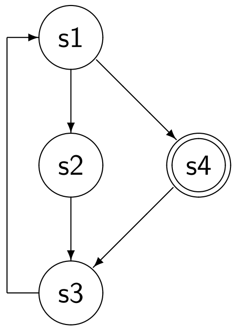
8.1 Safety

Package `Execute.Verify.Safety` performs the verification of safety properties. The main loop makes the top of the state stack the current state and the top of the location stack the current location. If this location has been `Visited`, it is popped; if it was the `Last` location of this set of locations, the state stack can be popped. Otherwise, the transition at this location is executed, as is a transition of the never claim (if any). An attempt is made to put the resulting state into the hash table. If successful (that is, if this is a new state), the depth-first search is implemented by pushing the state on the state stack and calling `Get_And_Push_Transitions`.

8.2 Acceptance

Package `Execute.Verify.Acceptance` performs the verification of liveness properties by looking for acceptance cycles. The overall structure is similar to that for verification of safety, except that when an accepting state is encountered an inner search is commenced.

Consider the the following state diagram, where s1 is the initial state:



Suppose that the search begins s1 → s2 → s3. Now s1 has already been visited, so the search backtracks and tries s4, an accepting state. The next transition is s4 → s3, but the state s3 has already been visited so the search terminates, although there is a computation with an acceptance cycle s1 → s4 → s3 → s1 → s4.

When an accepting state like s4 is encountered, it is pushed on the stack with `Inner` set to 1 and a new search is begun. Now, the states s3 and s1 *with* `Inner` *set to 1* have *not* been visited and the acceptance cycle will be found. The accepting state from which the inner search is started is saved in `Seed` and must occur again for an acceptance cycle to exist. Function `On_Stack` is called to see if a new state already exists on the stack; if so, a path to the seed must exist. See pp. 179–181 of *SMC* or Section 4.4 of Baer and Katoen for more detail.

8.3 Fairness

Verification with weak fairness is implemented by *unfolding* the state space as described on pp. 181–188 of SMC. Copies of states are used to ensure that an acceptance cycle executes a transition of each enabled process. The copies are indexed from 0 to $k + 1$, where k is the number of processes not counting the never claim; the value $k + 1$ is stored in the variable `Unfolded`. Each state vector has a counter `Fair` (initialized to 0) that is used to keep track of which copy the state belongs to, and only copy 0 has accept states.

When an *accepting state* is encountered in copy 0 of a state, the current state becomes copy 1 of the state; transfer from copy i to copy $i + 1$ occurs whenever a transition of process i is executed. The inner search is begun from the $k + 1$'st copy. Since all transitions from accepting states (in copy 0) lead out of the copy and since the only way of returning to copy 0 is by traversing all the other copies, an accepting cycle must contain transitions from all processes and thus is fair.

When checking fairness, null transitions for blocked processes must be added when in the unfolded states (p. 183 of *SMC*). A dummy transition is created (by `Automata.Read`) at the end of the transitions for each process (except the never claim); this transition is ignored when executing `Get_All_Locations` in package `Automata`. `Get_And_Push_Transitions` of package `Execute.Verify` calls procedure `Add_Null_Transition`, which checks if there are executable transitions for the copy associated with the current value of the fairness counter (not including the 0'th and $k + 1$'st copies). If not, a null transition is constructed by calling `Set_Null_Transition` in package `Automata` which sets the source and target states to be the current state and returns the transition index. The null transition is then added to the list of executable transitions before the transitions of the never claim. Null transitions are not written to the trail.

When processes are allocated using `run`, the formal `proctypes` have process IDs, while the values of `Unfolded` must be checked for the process IDs of the allocated processes. To implement this, it is required that the formal `proctypes` appear first in the program and the variable `Unfolded_Bias` is used as the start of the active or allocated processes.

9 Translation of LTL formulas to Büchi automata

The algorithm implemented is from:

> Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. 1996. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, 3-18.

See also Section 4.4 of Kröger and Merz, or Section 5.2 of Baer and Katoen.

To make the algorithms easier to follow, the unary temporal operators `<>` and `[]` are directly implemented, rather than being translated into `U` and `V`.

Package `LTL` is the parent for a set of *private* packages that implement the algorithm. Formulas and nodes are declared in the private part of package `LTL`, along with sets of formulas and nodes that are implemented using `Ada.Containers.Ordered_Sets`.

A formula represented as a string is translated into a tree data structure. For unary operators `!`, `[]` and `<>`, the *left* subtree is null; this is so that in both forms of future formulas (`<>q` and `pUq`), the subformula to be fulfilled is in the right subtree. Sets of formulas are implemented by ordered sets whose `Element_Type` is `Formula_Pointer`, a pointer to the tree representation of a formula.

The visible part of package `LTL` declares a *function* `LTL_To_Automaton` which takes a formula represented as a string as a parameter and returns a set of transitions representing the BA. The function first calls `Push_Negation` to push negation inwards. Function `Construct_Tableau` is called and returns the set of nodes that are states. Finally, `Convert` is called to convert the states into transitions.

Package `LTL.Formulas` declares subprograms for LTL formulas. There are procedures to display formulas and sets of formulas. `To_Formula` translates a string into a formula and `Get_Literal` returns the string representation of a literal. Function `Push_Negation` returns the formula after pushing negations inward and function `Contradiction` checks a set of formulas for contradictions and returns a Boolean value. Procedure `Decompose` decomposes a formula according to the rules of LTL: one or two subformulas that must be true in the current state, and—for the temporal operators—suformulas that must be true in the next state.

Package `LTL.Nodes` constructs the tableau as described in the article. Each formula in a node is decomposed into its subformulas; when no more decomposition is possible, either the node is closed because of a contradiction or it is open or a new node must be created because there are formulas that must be true in the `Next` state. If a new node has the same formulas as one that already exists, it is not created; instead, the current node is added to the list of `Incoming` nodes of the existing node.

Package `LTL.Automaton` contains the function `Convert` to convert the states into transitions. The `OldF` field of each state contains the formulas that must be true in that state. The transitions are defined as follows: for every state s_i in the `Incoming` set of state s, there is a transition $s_i \rightarrow s$. Any literal in `OldF` of s is a condition of the transition.

The resulting BA is optimized as follows: if there is a set of states $s_1, s_2, \ldots$ such that they all have transitions to the same state t and labeled by the same condition, then the set can be collapsed. This is performed in procedure `Optimize`, which calls `Remove_Duplicates` to collapse duplicate transitions that result from collapsing the states.

Function `Get_Future_Formulas` returns the set of all formulas F of the form `<>q` or `pUq`. An accepting state is one such that F is not in the state or `q` is in the state.

If there is more than one future formula, the BA that results is called a *generalized Büchi* automaton with multiple sets of accepting sets, one for each future formula. The generalized BA must be *degeneralized* to a BA with one set of accepting states. The algorithm, described in Baer and Katoen, pp. 195–196, is implemented in procedure `Degeneralize`. Copies of the states are made, one for each future formula. The only transitions that move between copies are those from an accepting state for the i'th formula; they have a target in the next copy (modulo the number of copies). Only states in the first copy retaining their status as accepting. It follows that if there is an accepting cycle, it must have traversed all the copies and hence accepting states for all future formulas.

10 Compilation

The compiler is a distant descendant of the one used in the PASCALS interpreter written by Niklaus Wirth that was the basis of concurrency simulators that I developed. As such, it is retains much of the structure inherited from standard PASCAL.

The compiler is a one-pass recursive descent compiler, where transitions (with the byte code for their statements and expressions) are constructed on-the-fly. However, control structures like guarded commands, `break` and `goto` require that the target states of the transitions be "fixed up" when later statements are compiled, so the transitions are stored in a table and only written out when a the compilation of a `proctype` has been completed.

10.1 Global declarations

The package specification `Compiler_Global` declares four data structures (all arrays with their associated counters):

- The symbol table `Tab`. Its fields are the name of a symbol, whether it is a variable or some other kind of symbol, a variable's type, the offset in the state vector, whether it is global or local, and byte code for its initialization. Arrays include the element type and number of elements, and `proctypes` include the indices of the symbols for the formal parameters. The counter is `T`.
- The transition table `T_Tab` is the same as the table in the model checker. The counter is `Transition_Counter`.
- A table of integer constants `I_Tab`. The counter is `Number_Counter`.
- A table of string constants `String_Tab`. The counter is `String_Counter`.
- A table of the identifiers `ID_Tab` which is used by the preprocessor.
- A table of tokens `Token_Tab` which is used by the preprocessor. Its components are records of type `Token_Rec` and contain the symbol and—for integers, strings and identifiers—an index into the appropriate table. In addition, the line number and a pointer to the source code is saved.

There are other counters for local tables described below: `Channel_Counter`, `Fix_Counter`. Other global variables keep track of the current `State_Number` and `Process_Number`, static `Level` (0 for global or the symbol table index of a process for a local variable), and the state vector `Offset`.

Finally, there are a number of flags that are used to influence code generation. These were used to limit modifications to the code for the PASCALS compiler. For example, when `a[n]` is seen at the beginning of a statement, we do not know if it is the beginning of an assignment statement `a[n]=b` or an expression used as a statement. Therefore, when compiling a `Factor` of an expression, the flag `LHS_Array` is used to indicate that `a[n]` has already been compiled. The flags are documented in comments.

10.2 Lexer

The lexical analyzer procedure `Get_Symbol` is called by the preprocessor to read characters and translate them to tokens, which are stored in `Token_Tab`. This table is modified as needed by the preprocessor and then the compiler reads from the token table by calling `In_Symbol`. The global variable `Sy` holds the symbol of a token. For identifiers, `Id` contains the name; for integer constants, `Inum` contains the value and for string constants, `Str` contains the index into the string table.

In some cases, two-symbol lookahead is required, so there are declarations of additional variables `Next_Sy`, etc. When `In_Symbol` and `Get_Symbol` are called, they copy the "next" variables into the current ones and reads new values into the next variables.

There are variables that store the input line buffer and line and character counters.

10.3 Preprocessor

The preprocessor performs the following tasks (in the order given):

- Extraction of embedded LTL specification.
- Textual replacement for `#define`.
- Textual replacement for `inline`.
- Translation of `select` statements to `do` statements.
- Translation of `for` statements to `do` statements.

The preprocessor calls the lexical analyzer to read the input file into `Token_Tab` and then performs all the processing by adding and deleting sequences of tokens. For `#define` and `inline`, the array `Define_Array` stores the identifer that is replaced, the formal parameters for `inline` and the sequences of tokens that replace the identifier.

Embedded LTL specifications are processed when the tokens are read. A command-line argument will identify which named specification to use (or to use the default one). This source code of this specification is copied to `LTL_Buffer` in package `LTL`, where it is later parsed during the translation of the formula to a Büchi automaton.

10.4 Compiler

The compilation is performed in package `Compile`, either by procedure `Compile_File` for a program or by procedure `Compile_Expression` for an expression in an LTL formula. After initialization, global variables (including channels and mtypes) are compiled, followed by the proctypes. The subprograms follow the syntax of the these constructs, but note the following:

- `mtype` values are assigned in reverse order of declaration.
- Several variables can be declared in one declaration: `byte a, b, c`. After the type is compiled, a loop enters the variables into the symbol table.
- For an array of channels, the channel initializer is written to the automata file once for each element of the array.
- A PROMELA program may consist of a non-terminating loop, but if there is a final statement or a jump past the end of the statements, an additional transition with `end` indicated is produced.

10.5 State

Package `State` compiles statements. Procedure `Statement` contains an `if`-statement for all the tokens that can start a statement. This subprogram is straightforward except that an expression can also start like a statement. One possibility is that it can start with a factor begin symbol like `(` or `!`. The other is that it starts like an assignment statement or a channel statement: `a[n]>b;` or `a[n]!b;` or `a[n]=b;`. In this case, additional lookahead is done to identify the construct being compiled.

For `goto` statements, there are two possibilities: If the label has already been encountered, in which case its `Adr` field contains its state and the target of the transition is set to this value. If the label has not been encountered, the label is entered into the symbol table with a dummy value `-1` for the component `Adr`; next, the transition index and the symbol table index of the label are entered into the table `Goto_Fixes`. When the label is later encoutered, the `Adr` field is given a real value and at the end of the compilation of the `proctype` `Fix_Gotos` is called.

For channel, `printf` and `run` statements, the arguments must be pushed in reverse order so that the first argument is on the top of the stack. Procedure `Reverse_Operands` copies the byte code for the arguments into a temporary variable and then copies them back in reverse order. An array `Offsets` stores the offset in the byte code array of each of the arguments.

`if` and `do` statements are difficult to compile because the transition source and target states are not easy to compute. The source of the first transition of each alternative must start in the same source state and this is stored in the variable `Source`. For an `if` statement, the target of the last transition of each alternative must be the state after the `fi`, while for a `do` statement it must be the source state of the `do`. Since the transitions are produced by recursive calls to `Statement`, their indices are stored in arrays `Firsts` and `Lasts` and fixed up when the `fi` and `od` are encountered. `goto` statements are either currently correct or will be fixed up later. Finally, an `fi` may appear just before a `od`, in which case, the jumps must be fixed up to return to the source of the `do`.

The array `Breaks` (with index `Break_Count`) is used to fix up the targets of `break` statements. Each entry holds not only the transition to be fixed up, but also the nesting level

Do_Level of the do statement. When the od is encountered, the targets of the break transitions *for the current nesting level* are fixed up and then the nesting level decremented.

10.6 Expr

This package contains the subprogram Expression with deeply nested subprograms for compiling expression. Flags Is_Copy, LHS_Array and Emit_Load_Address influence the code generation as described in the comments.

11 The COMPILER program

There is a main subprogram Compiler in the src directory that calls
Compiler_Declarations.Compile_File.

12 The LIST program

There is a main subprogram List which reads the aut file output by the compiler and reformats it in a text file with the extension lst. The aut file is read twice because the table of numbers appears after the processes since a number might appear as a literal in a statement.

13 The TRACE program

Constants and types are declared in package `Global`, together with `Process_Width`, `Line_Width`, `Statement_Width`, `Variable_Width` and `Title_Repeat`, which are declared as variables since they can be changed by command-line arguments.

Package `States` contains the subprograms for formatting the output. Its variables are declared as private so that the tables of excluded variables and statements can be set by the child package `Arguments`.

The main subprogram `Trace` contains a procedure `State_Loop` which reads the trace file of simulation and prints the state in the lines for the initial and next states and the chosen transitions.

14 The VMC program

The main subprogram is `VMC` and the packages `Global` and `Utilities` are self-explanatory. The creation of the graphics files is done in three stages, each implemented in a package.

Package `Model_Data` reads the variable and process names, records if this is a simulation or a fairness verification and reads the transitions of each process.

Package `Run_Data` reads the trace of the run of the simulation or verification: the construction of states in the state space, and, for a verification, pushing and popping states and transitions on the stack. Whenever a new state is created or matched, a transition from the current state is stored in `Edge_Table`. These transitions are called *edges* to avoid confusion with the transitions of the processes.

Each state is stored in a component of type `State_Record` in the array `State_Table`. The record stores the state's name and flags for: on the stack, a state that was matched one or more times, if this is the current state being expanded, if the state is in error (for example, an assertion is false or an invalid end state), and for a verification of acceptance or fairness if this state was generated during an inner search. The states name consists of the process location counters and the variable names and values as printed in the trace; for example: `wantp=1,wantq=1,critical=0,p=6,q=2,`. For a simulation, the record stores the number of executable transitions and the transitions themselves.

Package `Generate_Space` reads the data stored in the previous two states and generates a new DOT file for each significant step. The prologue of the DOT file is read from a file if it exists; otherwise, the file is written with defaults. The nodes of the DOT file are the components of `State_Table` and the edges are the components of `Edge_Table`. The subprogram `Format_Label` constructs the `label` field of the node containing both the label and the various graphical effects.