MDG Tools (V1.0) User's Manual

Zijian Zhou and Nancy Boulerice

support: mdgverif@iro.umontreal.ca
http://www.iro.umontreal.ca/labs/lasso/research/mdgverif/mdgverif_eng.html\

Members in the group

Koty Dominique Anon:	anon@iro.umontreal.ca
Nancy Boulerice:	bouleric@iro.umontreal.ca
Eduard Cerny:	cerny@iro.umontreal.ca
Francisco Corella:	fcorella@hprpcd.rose.hp.com
Michel Langevin:	langevin@gmd.de
Xiaoyu Song:	song@iro.umontreal.ca
Sofiene Tahar:	tahar@iro.umontreal.ca
Ying Xu:	xu@iro.umontreal.ca
Zijian Zhou:	zhouz@iro.umontreal.ca

1. Introduction

RT (Register-Transfer) level hardware designs can be suitably represented by *Multiway Decision Graphs* (MDGs), a class of decision graphs that subsumes the class of ROBDDs (Reduced Ordered Binary Decision Graphs) while accommodating abstract sorts and uninterpreted function symbols [1]. In this document, we describe how to use the MDG tools - a prototype implemented in Prolog for the verification of RTL designs. The MDG tools are intended for Abstract State Machine (ASM) [1][2] verification rather than Finite State Machine (FSM) verification. They can be used for FSMs as well, but they are less efficient than ROBDDs for this purpose, due in part to the space requirements of our current Prolog implementation.

The MDG tools include an MDG package, a reachability analysis procedure and the applications for combinational and sequential hardware verification at RT level. The MDG package implements manipulation algorithms for MDGs. The reachability analysis algorithm checks that an invariant holds in all the reachable states of an ASM using the abstract implicit enumeration technique [1].

Applications. Four applications for combinational and sequential hardware verification are provided:

- ASM state space exploration: reachability analysis for an ASM with an invariant which is always true.
- ASM safety property checking: reachability analysis for an ASM with a certain invariant.
- ASM equivalence checking: reachability analysis for the product machine of two ASMs with an invariant that states the equivalence of the corresponding outputs.
- Combinational verification: equivalence checking of circuits having the same input, output and state variables (if any) by comparing their transition and output relations.

Application scope. An ASM can be a specification or an implementation. The implementation considered here is usually an RTL structural description given as a network of components connected by signals. The specification can be another structural description known to be correct, or a higher level description (e.g., an instruction set architecture for microprocessors). When an implementation is compared with a specification, it must be couched with the specification in terms of the same set of uninterpreted function symbols. If necessary, conditional rewrite rules can be used to partially interpret the otherwise uninterpreted function symbols.

Environment. The MDG tools run under Quintus Prolog V3.2. It is possible to run under a lower version with a few modification.

Contents of this document. This document is about how to use the above applications. It contains:

- A Prolog-style HDL (Hardware Description Language);
- User commands for the applications;
- Practical issues in using the applications.

The examples will be given in 11 points `courier font.` In a Prolog-style declaration, '[...]' means a list and a string starting with an upper case character is a Prolog variable.

Further references. This manual is not self-contained. Readers should refer to the references, especially [1] for terminology and techniques about Multiway Decision Graphs and reachability analysis of abstract state machines. To use the applications, though it is not necessary, some preliminary knowledge of Prolog is preferred. For the readers who want to build their own applications using the MDG package, they are recommended to read *MDG tools (V1.0) developers manual*.

2. MDG-HDL: A Prolog-Style HDL

The MDG tools accept as hardware description a Prolog-style HDL, MDG--HDL, which allows the use of abstract variables for representing data signals. The MDG--HDL description is then compiled into the ASM model in internal MDG data structures. MDG--HDL supports structural descriptions, behavioral ASM descriptions, or a mixture of structural and behavioral descriptions. A structural description is usually a netlist of components (predefined in MDG--HDL) connected by signals. A behavioral description is given by a *tabular* representation of the transition/output relation or truth table.

Besides circuit descriptions, a variety of information, such as sort and function type definitions, symbol ordering and invariant specification, etc., has to be provided in order to use the applications outlined in Section 1. All of these are organized into six kinds of input files: the algebraic specification file, the symbol order file, the circuit description file, the invariant specification file, the state encoding file and the manual partitioning file for transition/output relations.

- The *algebraic specification file* defines sorts, function types and generic constants (see [1] for the logic) used in hardware descriptions. And if necessary, it also includes the rewrite rules which partially interpret the otherwise uninterpreted function symbols.
- The *symbol order file* provides the *custom* (user-defined) *symbol order* for all the variables and cross-operators which would appear in MDGs.
- The circuit description file declares signals and their sort assignments, component network, outputs and the mapping between state variables and next state variables. There is a special component construct *table* which is the tabular representation for behavioral descriptions. For sequential circuits, we also give the set of initial states and the transition/output relation partitioning strategy.
- The *invariant specification file* defines the invariant to be checked during the reachability analysis.
- The *state encoding file* provides the mapping of state encoding between two circuits to be compared. If such a mapping is not available or is not necessary for the verification, this file can be omitted.
- The *manual relation partitioning file* allows users to declare partitioned transition/output relations manually for the product machine. It could be omitted if no manual partitioning is necessary.

Below we explain each kind of the files in detail. The declarations are given as facts in Prolog. But it is not necessarily for the users to know Prolog. Most of the examples are taken from the MinMax example, which can be found in homeMDG/examples/minmax.

2.1 Algebraic Specification File

This file declares sorts, function types and generic constants used in the hardware description. If rewrite rules are needed to interpret the function symbols, they are also specified in this file.

The file should always include the following standard head required by the Prolog system:

```
:- multifile abs_sort/1.
:- multifile conc_sort/2.
:- multifile function/3.
:- multifile gen_const/2.
:- multifile rr/3.
:- multifile xtrr/3.
```

The algebraic specification is given by the following declarations.

1. **abs_sort**(*Sort*) declares *Sort* as an abstract sort.

   Example: `abs_sort(wordn).`

   It declares *wordn* is an abstract sort.

2. **conc_sort**(*Sort, IndivCons*) declares *Sort* as a concrete sort having list *IndivCons* as its enumeration.

   Example: `conc_sort(bool,[1,0]).`

   bool is a concrete sort whose enumeration is given in the list [1, 0], where 1 and 0 are treated as individual constants.

3. **function**(*Func_symbol, Args_sorts, Target_sort*) declares a function type. *Func_symbol* is the function symbol (also called as top symbol of the function). It can be an abstract function symbol or a cross-operator. *Args_sorts* is a list of the sorts of the arguments and *Target_sort* is the sort of the function range. Note that *Target_sort* and *Args_sorts* cannot all be concrete, since concrete function symbols are not used in MDGs.

   Example 1: `function(add,[wordn,wordn],wordn).`

   add (uninterpreted, but could mean addition) is a function which takes two arguments whose sorts are *wordn*, and returns a result of sort *wordn*. In this case, *wordn* must be abstract sort and *add* is an abstract function symbol.

   Example 2: `function(leq,[wordn,wordn],bool).`

   leq is a function (also uninterpreted, and intends to mean less then or equal) with two arguments of abstract sort *wordn*, and returns a result of Boolean sort *bool*. *leq* is the called as cross-function symbol (or cross-operator, or cross-op).

4. **gen_const**(*Gen_const, Sort*) declares a generic constant *Gen_const* as of sort *Sort.*

   Example: `gen_const(min,wordn).`

   min is a generic constant of sort *wordn*.

5. **rr/3** and xt**rr/3** define rewrite rules.
   rr(*Cs, LHS, RHS*) is a conditional rewrite rule: *Cs* => *LHS* -> *RHS*.
   LHS (left hand side) and *RHS* (right hand side) are terms. *Cs* is a list of conditions. A condition can be a pair (Xt, C) composed of a cross-term and an individual constant. Or it is simply a term which is the goal of a predicate.

   Example 1: `rr([(eq(X,Y),0), (iszero(X),1)], iszero(Y), 0).`
   It means if (eq(*X,Y*)=0) and (iszero(X)=1) then iszero(*Y)* equals 0.
   Here, '[(*eq(X, Y)*, 0), (*iszero(X)*, 1)]' is the condition list.

   xtrr(*Cs, LHS, RHS*) is a conditional rewrite rule for cross-terms: *Cs* => *LHS* -> *RHS*.
   LHS must be a cross-term and *RHS* must be an individual constant. *Cs* may contain arithmetic expressions, but must not contain cross-term and individual constant pairs.

   Example 2: `xtrr([], eq(X,X), 1).`
   It represents the rewrite rule: X=Y => *eq(X,Y)* -> 1. The condition X=Y (syntactic equivalence) is implicitly expressed in the rule. The condition list is thus empty.

   Example 3: `xtrr([square(X,Y)], sq(X,Y), 1).`
   square(X,Y):- Y is X * X.
   It represents the rewrite rule: X^2=Y => *sq(X,Y)* -> 1. The condition X^2=Y is an arithmetic expression. It is thus evaluated by a Prolog predicate: square(X,Y):- Y is X * X.

For some of the commonly used algebraic declarations which are given in the file common.pl (Appendix 9.1), it is not necessary to define them again in the user's algebraic specification file.

2.2 Symbol Order File

Like ROBDDs, the MDGs require a total order over all the nodes in the graphs. This order is manually provided by the user as order_main(*Symbols*) where *Symbols* is a list of variables and cross-operators.

Example: `order_main([r,x,c_A,n_c_A,rm_A,n_rm_A,rM_A,n_rM_A,leq]).` (the symbol order for MinMax specification (mm2_s.pl in /mdgpac/example/minmax).)

We say that symbol *a* comes before symbol *b* if *a* is at the left side of *b* in the list. In an MDG, *a* will appear above *b*. We may also write it as *a* < *b*.

MDGs have some requirements on the node ordering of abstract variables and cross-operators (NO requirement for concrete variables).

- If a variable *a* will appear as a secondary variable in an edge label of node *b*, then *a* < *b*. (Note that secondary variables are always abstract variables).

Example: for $b=f(a)$ where f is an abstract function symbol, then $a < b$.

- If a variable a will appear as a secondary variable in a cross-term having cross-op f, then $a < f$.

  Example: for $f(a)=c$ where f is a cross function symbol, then $a < f$.

- Because of the renaming substitution, the state variables and next state variables must be in a correspondent order.

  Example : if the state variables are in order $a < b < c$, then the corresponding next state variables should be also in such an order as $n\_a < n\_b < n\_c$.

- If using rewriting algorithm 2, the conditional rewrite rule rr(*Cs, LHS, RHS*), it requires that the cross-operators in *Cs* must come before the terms in *LHS* in the graph structure. For example, for rule: rr([(*eq(X,Y)*, 0), (*iszero(X)*, 1)], *iszero(Y)*, 0), cross-operator *eq* should come before *iszero*.

Besides the above requirements, the symbol order can affect, sometimes critically, the size of an MDG. So it is an important task to find a good order. Some ordering heuristics, according to our experience, are given in section 4.2.

2.3 Circuit Description File

This file should always include the following standard head required by the Prolog system:

```
:- multifile signal/2.
:- multifile component/2.
:- multifile outputs/1.
:- multifile st_nxst/2.
:- multifile init_val/2.
:- multifile init_var/2.
:- multifile par_strategy/2.
:- multifile next_state_partition/1.
:- multifile output_partition/1.
```

A circuit description can be a structural description, a behavioral ASM description, or a mixture of structural and behavioral descriptions. A structural description is usually a netlist of components (predefined in MDG--HDL) connected by signals. A behavioral description is the transition/output relations of an ASM given by *table* construct. In this file, we need to declare the following:

1. signal(*Signal, Sort*) declares a *Signal* to be of sort *Sort*. Each signal must be declared once and only once.

   Example: `signal(x,wordn).`

   It declares that *x* is a signal of sort *wordn*.

2. component(*CompName, CompDef*) declares *CompName* to be a component defined by *CompDef*. *CompDef* instantiates the input/output ports of a predefined component module. A set of predefined component modules are given later in this section.

3. outputs(*Outputs*) declares *Outputs* as a list of outputs of the circuit.

For sequential circuit, the user should provide the mapping between state variables and next state variables.

4. st_nxst(*Stvar, NextStvar*) declares that *NextStvar* is the next state variable of a state variable *Stvar.*

For sequential verification, the initial states are given by init_val/2. State variables which are initially *don't cares* need not be specified. In some circumstances, the initial states should be generalized in order to avoid the non-termination problem [1]. If this is the case, the initial value then has to be declared as variable using init_var/2. It is up to the user to decide which initial state value should be generalized. More discussions about this issue are given in Section 4.1.

5. init_val(*Stvar, InitVal*) declares state variable *Stvar* has *InitVal* as its initial value.

   Example 1: `init_val(pc,zero).`

   Example 2: `init_val(c,1).`

6. init_var(*InitVal, Sort*) generalises the initial state value *InitVal* to be a variable of sort *Sort.*

   Example:
   ```
   init_val(pc,init_pc).
   init_var(init_pc,wordn).
   ```

   It declares that init_pc is the initial value of pc and it is a variable of sort *wordn*.

There are different kinds of strategies for representing transition relations [3]. Normally, we use partitioned transition relations. The user is provided with an interface to select the desired method to generate the partitioned transition relations.

7. par_strategy(*OrdMethod, ParMethod*) declares *OrdMethod* as the ordering strategy for individual relations and *ParMethod* as the partitioning strategy. If *OrdMethod*=auto, the program uses the order generated by the heuristic ordering algorithm [3]. If *OrdMethod*=default, the program uses the order specified by next_state_partition/1 (to be explained below). If *ParMethod*=single, we use one partition block containing all the individual transition relations. If *ParMethod*=auto, the partition blocks are formed by the automatic grouping algorithm. If *ParMethod*=default, the partition blocks are as specified by next_state_partition/1 [3].

8. next_state_partition(*Partitions*) is a 3-level nested list. The inner most list gives a list of next state variables. The transition relation MDG containing all the variables in the list forms an individual transition relation. The middle level list specifies a default partition block (i.e., such a partition is used when *ParMethod*=default in par_strategy/2.

   Example 1: `next_state_partition([[[n_a]],[[n_b]],[[n_c]]]).`
   It declares that each individual transition relation has exactly one next state variable. And each default partition block has only one individual transition relation.

   Example 2: `next_state_partition([[[n_a]],[[n_b, n_c]]]).`
   In this case, the second individual relation has two next state variables, n_b and n_c.

Example 3: `next_state_partition([[[n_a]],[[n_b], [n_c]]]).`
It declares that each individual transition relation has exactly one next state variable. But the second default partition block has two individual transition relations.

9. output_partition(*Partitions*) is similar to next_state_partition/1. The variables appearing in *Partitions* are the outputs declared in outputs/1, excluding state variables.

As a summary, items 1 - 3 are needed for the comparison of combinational circuits, items 1 - 4 for the comparison of transition/output relations of sequential circuits, and items 1 - 9 for sequential verification.

2.4 Invariant Specification File

This file specifies the invariant condition to be checked during reachability analysis. An invariant condition can be specified by a combinational circuit whose output signals are named by the variables that occur in the condition. By convention, an assignment of values to those variables satisfies the condition iff the outputs of the combinational circuit take those values for some assignment of values to the inputs. An MDG representing the invariant is obtained from the MDG representing the functionality of the combinational circuit by existentially quantifying the concrete inputs. The variables representing abstract inputs are left in the graph as implicitly quantified secondary variables [4].

For example, for the equivalence checking of two ASMs, we need to specify the equality of two corresponding signals as the invariant. This is expressed by the simple fork as shown in Figure 1 (a). The fork may yield different MDGs depending on the sort of the signals. If *u*, *x* and *y* are of the Boolean sort, then *u* is existentially quantified and we get the MDG as shown in Figure 1 (b) which simply represents *x*=*y*. If *x* and *y* are of an abstract sort, then we get an MDG as shown in Figure 1 (c) which represents the formula $(x = u) \wedge (y = u)$. Taking the secondary variable *u* to be existentially quantified, the invariant is $\exists u\, ((x = u) \wedge (y = u))$, which is logically equivalent to $x = y$.

This combinational circuit is described completely in this invariant specification file, including the following predicates: signal/2, component/2, outputs/1 and order_cond/1 which gives the node order for the variables and the cross-function symbols appeared in the circuit.

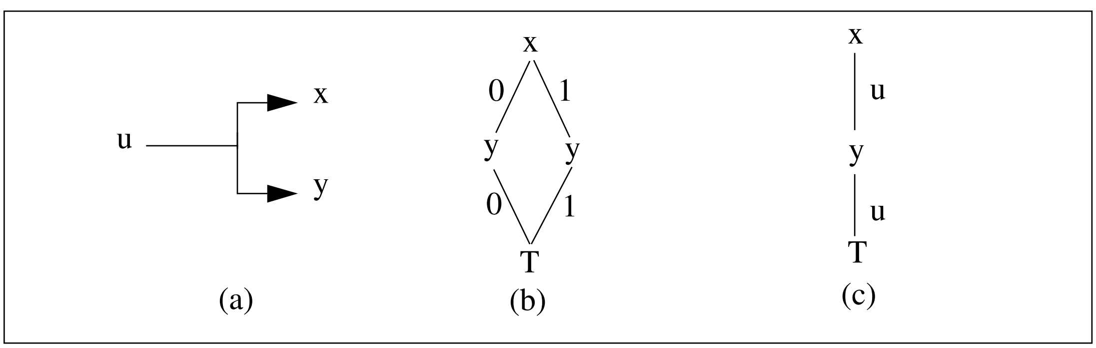


FIGURE 1. Invariant specification

2.5 State Encoding File

This file should always include the following standard head required by the Prolog system:

```
:- multifile st_encode/1.
```

st_encode/1 gives the state encoding between state variables. For example, if a state variable *s* is of sort *word2* whose enumeration is [0, 1, 2, 3] and it is encoded using 2 Boolean variables *r1* and *r2*, then we declare one possible state encoding as:

st_encode([[r1, r2, s], [0, 0, 0], [0, 1, 1], [1, 0, 2], [1, 1, 3]]).

2.6 Manual Relation Partitioning File

This file should always include the following standard head required by the Prolog system:

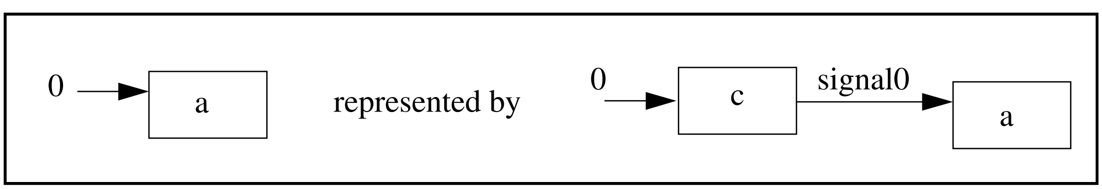
```
:- multifile output_partition/2.
:- multifile next_state_partition/2.
:- multifile par_strategy/2.
```

The par_strategy/2, output_partition/2 and next_state_partition/2 are declared as in Section 2.3. The next state variables from both circuits to be compared can be mixed up. In this sense, we provide a manual relation partitioning over the product state machine.

2.7 Predefined Component Modules

1. Constant signal: **constant_signal**(value(*Const)*, signal(*Signal: Sort*)).
   For a constant in a circuit, we use an extra signal which takes the constant as its value to represent it (this is to simplify the compilation of the circuit). The component module 'constant_signal' is for this purpose. Signal takes Const as its constant value. If Sort is a concrete sort, then Const must be an individual constant in the enumeration of Sort. Or if Sort is an abstract sort, then Const must be a generic constant of sort Sort.

   Example:



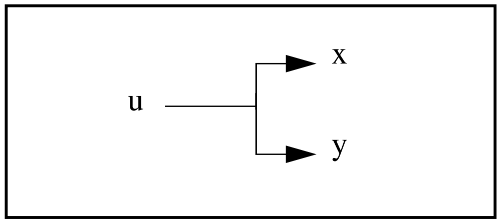
   For a Boolean constant 0, we use an extra signal *signal0* of Boolean sort to represent it and use the following component to generate *signal0*:

```
component(select,constant_signal(value(0),signal(signal0))).
```

2. Fork: fork(input(*Input: Sort*), output(*Output: Sort*)).

   Input and *Output* must be of the same sort *Sort*. Fork is used to represent the equality of two or more signals.

   Example:

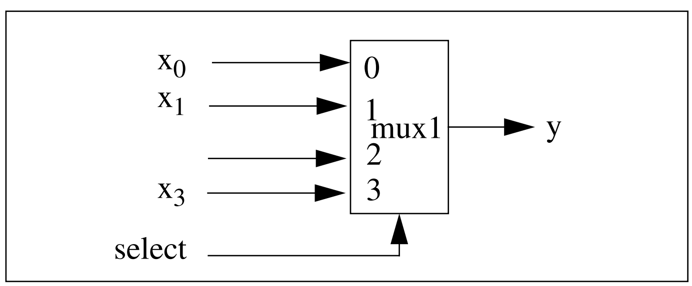



```
component(fork_x,fork(input(u),output(x))).
component(fork_y,fork(input(u),output(y))).
```

3. Multiplexer: **mux**(sel(*Sel:* $Sort_s$), inputs(*Port_Signal_pairs:* [($Sort_s$, *Sort*), ..., ($Sort_s$, *Sort*)]), output(*Output:Sort*)).

   Port_Signal_pairs is a list of select number and input signal pairs. The select number should be within the enumeration of the sort $Sort_s$. *Input* and *Output* must be of the same sort *Sort*.

   Example: The sort of signal *select* has an enumeration [0, 1, 2, 3].




```
component(mux1,mux(sel(select),inputs([(0,x0),(1,x1),(3,x3)]),out-
put(y))).
```

4. Not gate: **not**(input(*Input: bool*), output(*Output: bool*)).

   Input and *Output* must be of Boolean sort.

5. And gate: **and**(input(*Input1: bool, Input2: bool*), output(*Output: bool*)).

   and3(input(*Input1, Input2, Input3*), output(*Output*)).

   and4(input(*Input1, Input2, Input3, Input4*), output(*Output*)).

   and5(input(*Input1, Input2, Input3, Input4, Input5*), output(*Output*)).

   and6(input(*Input1, Input2, Input3, Input4, Input5, Input6*), output(*Output*)).

6. Nand gate: **nand**(input(*Input1, Input2: bool*), output(*Output: bool*)).

   nand3(input(*Input1, Input2, Input3: bool*), output(*Output*)).

   nand4(input(*Input1, Input2, Input3, Input4: bool*), output(*Output*)).

   nand5(input(*Input1, Input2, Input3, Input4, Input5: bool*), output(*Output*)).

   nand6(input(*Input1, Input2, Input3, Input4, Input5, Input6: bool*), output(*Output*)).

7. Or gate: **or**(input(*Input1, Input2: bool*), output(*Output: bool*)).

   or3(input(*Input1, Input2, Input3: bool*), output(*Output*)).

   or4(input(*Input1, Input2, Input3, Input4: bool*), output(*Output*)).

   or5(input(*Input1, Input2, Input3, Input4, Input5: bool*), output(*Output*)).

   or6(input(*Input1, Input2, Input3, Input4, Input5, Input6: bool*), output(*Output*)).

8. Nor gate: **nor**(input(*Input1, Input2: bool*), output(*Output: bool*)).

   nor3(input(*Input1, Input2, Input3: bool*), output(*Output*)).

   nor4(input(*Input1, Input2, Input3, Input4: bool*), output(*Output*)).

   nor5(input(*Input1, Input2, Input3, Input4, Input5: bool*), output(*Output*)).

   nor6(input(*Input1, Input2, Input3, Input4, Input5, Input6: bool*), output(*Output*)).

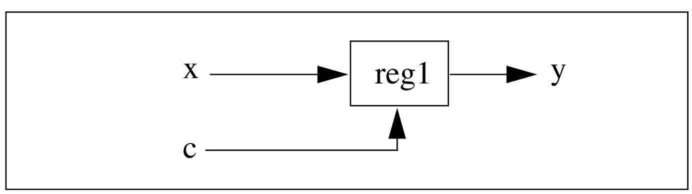
9. Xor gate: **xor**(input(*Input1, Input2: bool*), output(*Output: bool*)).

10. Register with control input:

    reg(control(*Control: bool*), input(*Input: Sort*), output(*Output: Sort*)).

    Output is the state variable. Its next state variable should be given using st_nxst/2.

    Example:




```
component(reg1,reg(control(c),input(x),output(y))).
```

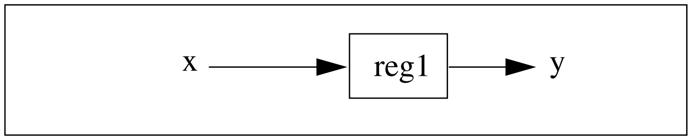
    y is the state variable. Its next state variable, say *n_y*, should be declare as st_nxst(*y, n_y*).

11. Register without control input:

    reg(input(*Input: Sort*), output(*Output: Sort*)).

    The state and the next state variable mapping should be given as st_nxst(*Output, Input*).

Example:



`component(reg1,reg(input(x),output(y))).`

Here, the state and the next state variable mapping is st_nxst(*y, x*).

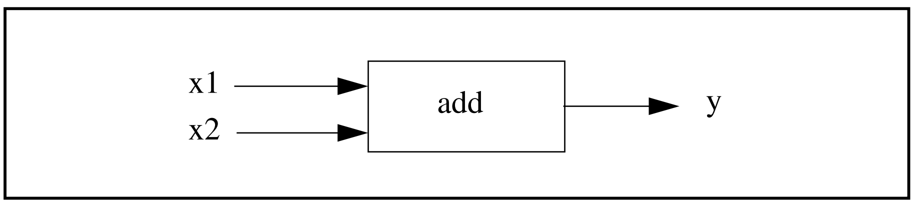
12. J-K Flip-Flop: **jkff**(input(*Input1, Input2: bool*), output(*Output: bool*)).

    Output is the state variable. Its next state variable should be given using st_nxst/2.

13. Uninterpreted functional block:

    transform(inputs(*Inputs:*[$Sort_1$, $Sort_2$, ..., $Sort_n$]), function(*Func_symbol*), output(*Output:Sort*)).

    Example:



`component(add1,transform(inputs([x1,x2]),function(add),output(y))).`

This is the declaration for an addition block which is viewed as a black box. The addition is represented using the uninterpreted function symbol *add*. The sort of *x1, x2, y* must be accordant with the type definition of *add* in the algebraic specification file.

14. Table: **table**(*Rows*).
    A table is similar to a truth table, but it allows first-order terms in the rows. *Rows* is a list of rows. The first row is a list containing variables and cross-terms. The last element of the list must be a (concrete or abstract) variable. All the other variables in the list must be concrete variables. Starting from the second row, each row is a list of values that the corresponding variables or cross-terms can take. The last element in the value list could be a first-order term, which means an assignment to the output variable. The other elements in the list must be either *don't cares* (represented by '*') or individual constants in the enumeration of their corresponding variable sort. The last element in *Rows* may be a term which serves as the default value.

    Example 1: `table([[x1,x2,y],[0,*,0],[1,0,0],[1,1,1]]).`
    This is a table declaration for an And gate (inputs: *x1, x2*; output: *y*).

    Example 2: `table([[c,leq(x,y),next_y],[1,1,x]|y]).`
    This is a table declaration for the following function: if (c=1) and (leq(x, y)=1)) then next_y <- x else next_y <- y. The term *y* after symbol '|' is the default value.

15. Bus: **drivers**(*Driver_Input_pairs:* [(*bool, Sort*)*, ...,* (*bool, Sort*)], output(*Output: Sort*)).
Driver_Signal_pairs is a list of bus driving signal and data signal pairs. The bus driving signal must be of *bool* sort and the data signals are of sort *Sort*. When the bus driving signal is 1, the output will take the data signal as its value. The program will automatically check for the bus accessing conflict (that is, at any time, only one bus driving signal can be 1).

3. MDG Tools

3.1 User Commands for Applications

The user should first set up correct program file search paths by modifying the "make.pl" file in the subdirectory homeMDG/programs/applications. The program search path is defined in program_path/1.

To run the program, first start a Prolog compiler from the directory which contains "make.pl" file (e.g., homeMDG/programs/applications). Then, at the Prolog prompt, type '**`[make].`**' and press the Return key. The Prolog compiler will load the "make.pl" file. Use the following commands to load files according to the applications:

- **`seq.`** For sequential verification.
- **`comb.`** For combinational verification.

Then use the following commands to start the application:

- **`seq_traverse.`** (or simply **`st.`**) for abstract state space exploration on one ASM.
- **`seq_property.`** (or simply **`sp.`**) for safety property checking of one ASM.
- **`seq_equivalence.`** (or simply **`se.`**) for equivalence checking of two ASMs.
- **`comb_verify.`** (or simply **`c.`**) for combinational verification.

When you have entered one of these commands, the program will prompt you printing options:

```
MDG printing options:
  No printing (Return) / To screen (Space) /
  To file (File Name)  / Interactively(?):[No printing]
```

These options control the printing of MDGs at certain pre-set printing points.

- A default option 'No printing' is provided between the square brackets, if this is your choice, press the return key.
- If you want the statistics of every MDG printed to your screen, hit the space bar and then return.
- If you want the statistics of every MDG printed to a file, type the filename and then return. The file will be saved in the directory specified by the data file search path given below.
- If you choose interactive printing, type **?** and then the return. The program will ask for your printing choice every time at those printing points.

After the printing option is chosen, the program will ask for the path from which the input data files can be found. For example:

```
| ?- Data file search path: [examples/gcd/] examples/minmax
```

A default path, ~/mdgpac/example/gcd/, is provided between the square brackets. If it is the right one, just press the Return key, else enter the path name, for example, ../examples/minmax.

Then, the program will ask, in turn, for the names of the input files. For example:

```
| ?- Algebraic specification file: [gcd_alg.pl] mm_alg.pl
```

A default file name, gcd_alg.pl, is provided between square brackets. If it is the right one, press the Return key, else enter the file name, for example, mm_alg.pl.

Different applications require different input files:

- ASM state space exploration: the algebraic specification file, the symbol order file and the circuit description file.
- ASM property checking: the algebraic specification file, the symbol order file, the circuit description file and the invariant specification file.
- ASM equivalence checking: the algebraic specification file, the symbol order file, the circuit description files for the specification and the implementation to be compared with, the invariant specification file, the state encoding file and the manual relation partitioning file. The two circuit description files must share and ONLY share the common input signals. And they should not share any components, i.e., a component name can only appear in one of the files. If the state encoding and the manual relation partitioning files are to be omitted, then type "space" and return.
- Combinational verification: the algebraic specification file, the symbol order file and two circuit description files for the specification and the implementation to be compared with. They must share and ONLY share the input and output signals.

When the program finishes, it will give out the result and provide detailed resource statistics. In the case that the sequential verification fails, a counterexample is generated. The execution flow of combinational verification and sequential verification are shown in Appendix B.

3.2 User Commands for Options

There are three run-time options that we can choose for sequential verification.

- Abstraction of cross-terms option:
  Switch on: `abs.` Switch off: `no_abs.`

  The system eliminates irrelevant cross-terms in a state MDG during reachability analysis. This increases performance when the graph size can be reduced greatly.
- Conditional term rewriting option:
  Switch on: `rew.` Switch off: `no_rew.`

  The system rewrites the outputs and newly reached states during reachability analysis.

- Generalization option:
  Switch on: `gen(N).`   Switch off: `no_gen.`

  N is a positive integer. With the switch on, the system eliminates all the cross-terms in the newly reached states (i.e., generalize the state) after *N* transitions during the reachability analysis.

3.3 Counterexamples

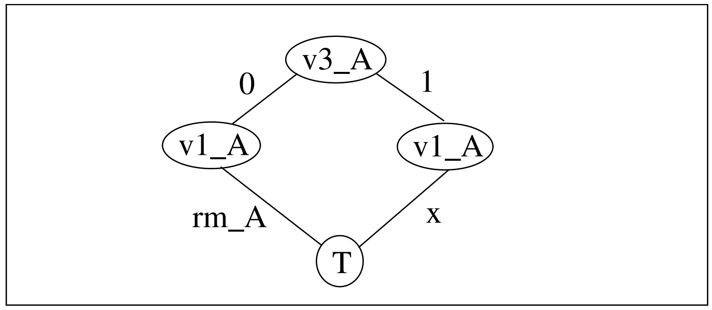
When the invariant is violated at some stage of the reachability analysis, a counterexample facility generates a sequence of input-state pairs leading from the initial state to the faulty behavior. This is very helpful for identifying design errors. Below is a counterexample generated when we compare the minmax specification with a fault implementation. In this counterexample, there is no assumptions. In clock cycle 2, there is no input value for signal *r*. This means that *r* is don't-care. In the symbolic output, the corresponding signals rM_A and rM_B, which should be equal, have different values. This suggests that there is an error associated with signal rM_B in the implementation.

```
=== The Counterexample ===
-------- Assumptions --------
-------- Initial state --------
c_A = 1
c_B = 1
rm_A = max
rM_A = min
rm_B = max
rM_B = min
-------- Clock cycle 1 --------
-- The symbolic input --
r = 0
x = x#1
-- The symbolic state --
c_A = 0
c_B = 0
rm_A = x#1
rM_A = x#1
rm_B = x#1
rM_B = min
-------- Clock cycle 2 --------
-- The symbolic input --
x = x#2
-- Symbolic Output --
c_A = 0
c_B = 0
rm_A = x#1
rM_A = x#1
rm_B = x#1
rM_B = min

=== End of counterexample ===
```

3.4 Printing MDGs

The printing format of MDGs are explained below with the following example:

```
-- v1_comp_A --

36 19 v3_A
  0: 30
  1: 31
  30 20 v1_A
    rm_A: t
  31 20 v1_A
    x: t
```




```
Graph Size statistics:   Nodes = 3  Abstract Edges = 2  Cases = 2
```

`v1_comp_A` is the reference of the MDG. For the nodes, three items are printed. The first is the unique identifier (ID) of the MDG. The second is the order number of the variable or the cross-op. The third is the label of the node. Below each node, there are a list of '*EdgeLabel* : *ID*' pairs, where the ID is the unique identifier of the subgraph that the edge *EdgeLabel* leads to. The graph size statistics are given in the last line, where Nodes and Abstract edges are the number of nodes and abstract edges in the MDG. Cases is number of all the paths in the MDG.

If the interactive printing option is chosen when the program is started, at each pre-set printing point, the program prompts the printing options as follows:

```
-- Print MDGs options --
     No printing (Return) / Statistics only (%) /
     To screen (Space)    / To file (FILE-NAME) :
```

- Hit the return key for no printing.
- Type "**%**" and then return for printing the statistics only.
- Hit the space bar and then return for printing the MDGs on the screen.
- Type a file name and then return for printing the MDGs to the file.

4. Practical Issues

In this section, we discuss several practical issues in using the applications.

4.1 Non-Termination Problem and Initial State Generalization

The abstract state enumeration method may not terminate. For example, if the program counter of a microprocessor is represented by a variable *pc* of abstract type, the initial value of *pc* is given by a generic constant *zero*, and a function symbol *inc* denotes the incrementing of *pc* by a non-branching instruction, then a node labelled by *pc* will have, in recursive invocations of the algorithms, an unbounded collection of edges *pc, inc(pc), inc(inc(pc))*, etc., and the algorithm will not terminate. This can be avoided by generalizing the initial state, using a variable *x* to denote the initial value of the program counter. Generalizing the initial state amounts to supplying an invariant. In the general case, however, there is no guarantee that an invariant can be found to avoid non-termination. A more detailed discussion about non-termination can be found in [1].

4.2 Controlling MDG Sizes

4.2.1 Symbol Ordering

Symbol ordering is also a critical part for MDG based verification methods. The symbol ordering task can be dissembled into two parts according to the verification process. The first part is the ordering of input, output, state and next state variables, together with the cross-operators. This ordering is crucial for the performance of reachability analysis where we only have those symbols in the MDGs. The second part is the ordering of internal signals which decides the performance of compiling transition and output relations. Instead of giving a total order for all the symbols, it is easier to do the job in the above two steps.

Besides the ordering requirements given in Section 2.2, we have developed several heuristics according to our experience.

- Although there are no requirements for the ordering of concrete variables, it is usually more efficient to put the concrete variable whose sort has a large enumeration set before other variables. This is because, when doing conjunction, the graphs can be restricted dramatically as early as possible.
- A certain abstract node may have a large number of edges. For example, an output of a data operation may have many different results according to different conditions. If such nodes could be identified, it is usually better to put them in the front of the symbol order list.

4.2.2 Partitioned Transition Relations

It is usually more efficient to use partitioned transition relations than monolithic transition relation. Although different partition strategies can be used with our interface, it is easier to use the built-in heuristic strategy.

4.2.3 ROM Declaration

In a microprogrammed microprocessor implementation, a ROM is used to store the micro-instructions. If it is declared as a single component, its MDG will be large and will take a long time to build. As each output of the ROM is used in different cases, it can be natural to break down the ROM into several pieces with each one dealing with only one output.

4.2.4 Rewrite Rules for Cross-terms

Rewrite rules for cross-terms can be used to shrink the MDG size on-the-fly. For example, if there is a path in an MDG which has cross-term `eq(x,x)=0` with `eq` intends for equality. We could use the xterm conditional rewrite rule `eq(X,X) -> 1` to eliminate this path.

4.3 Common Errors in Preparing the Description Files

Below, we list some common errors in preparing the description files (A list of error messages are provided in Appendix C):

- Missing various declarations.
- Not having the state and next state variables in a corresponding order in the order specification file.
- Input variables, state variables and next state variables are not disjoint. Usually this is caused by the use of registers without control. The input of the register is considered to be the next state variable. If the input of the register happens to be the primary input of the circuit, then it becomes both input and next state variables. The solution is to use register with control.
- The implementation and the specification share common signals and components other than the input signals. This problem can be avoided by always attaching different suffixes to the signals in circuits, e.g., '..._A' ('..._B') is used for a signal or a component in circuit A (circuit B).

5. Bugs

- Sometimes, when verification fails, the program is not able to provide the counterexample.

Please report any problems or questions about MDGs to *mdgverif@iro.umontreal.ca*

6. References

[1] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway decision graphs for automated hardware verification. To appear in the journal *Formal Methods in System Design.* Available as *IBM technical report RC19676*. T.J. Watson research center, IBM Research division, July 1994.

[2] F. Corella, M. Langevin, E. Cerny, Z. Zhouz and X. Song. State enumeration with abstract descriptions of state machines. In *Proceedings of IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (CHARME'95). Frankfurt, Germany. October, 1995.

[3] Z. Zhouz, X. Song, F. Corella, E. Cerny and M. Langevin. Partitioning transition relation automatically and efficiently. In *IEEE Proceedings of Fifth Great Lakes Symposium on VLSI* (GLSVLSI'95), Buffalo, USA. March, 1995.

[4] Z. Zhouz, X. Song, F. Corella, E. Cerny and M. Langevin. Description and verification of RTL designs using Multiway Decision Graphs. In *Proceedings of the Conference on Computer Hardware Description Languages and their applications* (CHDL'95). Chiba, Japan. August, 1995.

[5] Z. Zhou. MDG Tools (V1.0) Developer's Manual. December, 1995.

Appendix

A. File common.pl

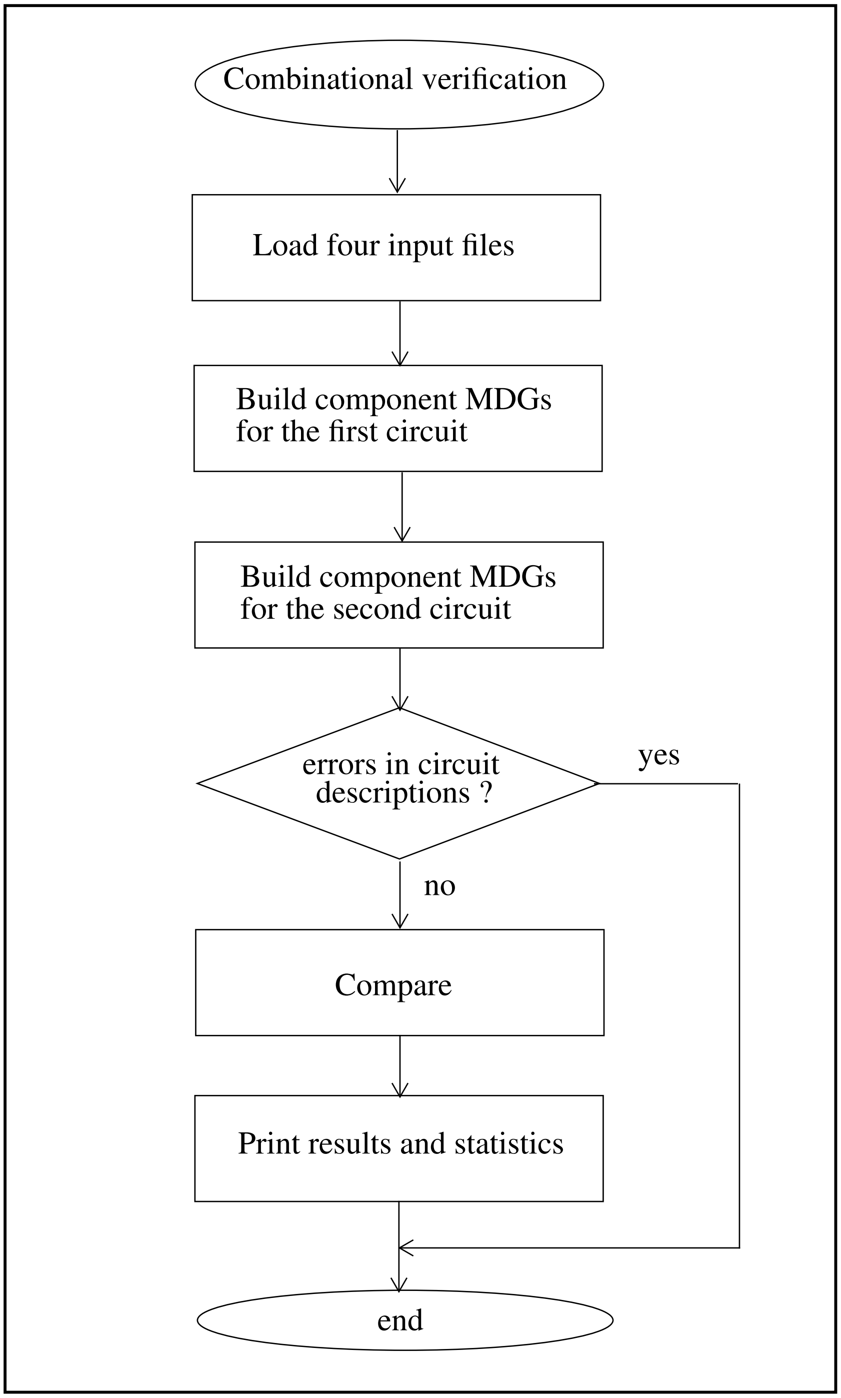
```
%============================================================
% File: common.pl: common algebraic specification.
%============================================================

% Standard head.
%
:- multifile abs_sort/1.
:- multifile conc_sort/2.
:- multifile function/3.
:- multifile gen_const/2.
:- multifile xtrr/3.
:- multifile rr/3.

:- multifile st_nxst/2.
:- multifile st_encode/1.

:- dynamic st_nxst/2.
:- dynamic st_encode/1.

% Algebraic specification
%
% Abstract sorts.
%
abs_sort(wordn).
abs_sort(wordp).

% Concrete sorts.
%
conc_sort(bool,[0,1]).
conc_sort(word2,[0,1,2,3]).
conc_sort(word3,[0,1,2,3,4,5,6,7]).
conc_sort(word4,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]).
conc_sort(word5,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]).
conc_sort(word6,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
 32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
 48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63]).
conc_sort(word7,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
 32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
 48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
 64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,
 80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,
 98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,
```

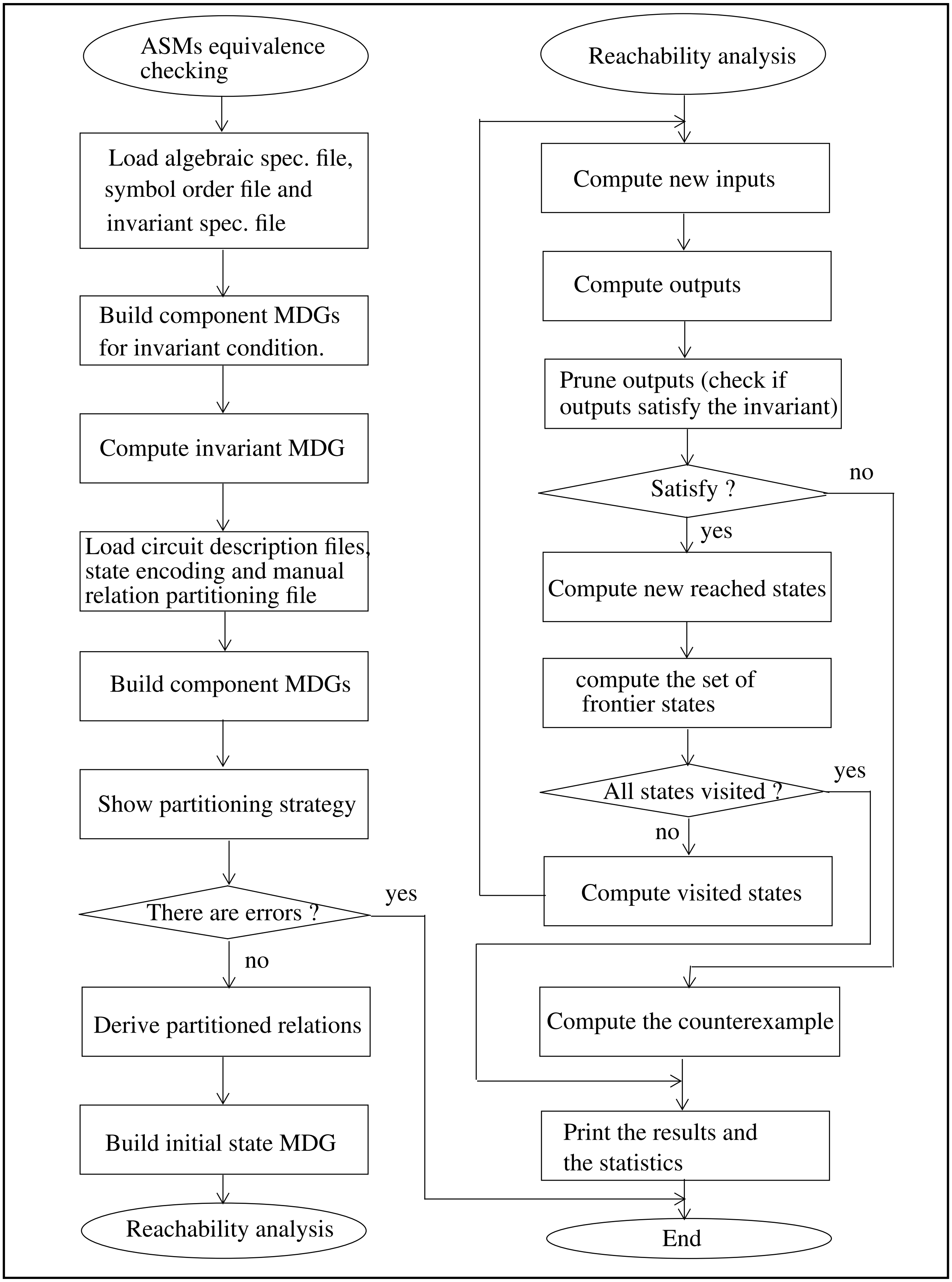
```
 113,114,115,116,117,118,119,120,121,122,123,124,125,126,127]).
conc_sort(word8,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
 32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
 48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
 64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,
 80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,
 98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,
 113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,
 128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,
 143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,
 158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,
 173,174,175,176,177,178,179,180,181,182,183,184,185,186,187,
 188,189,190,191,192,193,194,195,196,197,198,199,200,201,202,
 203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,
 218,219,220,221,222,223,224,225,226,227,228,229,230,231,232,
 233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,
 248,249,250,251,252,253,254,255]).

% Functions.
%
function(inc,[wordn],wordn).
function(add,[wordn,wordn],wordn).
function(sub,[wordn,wordn],wordn).
function(mul,[wordn,wordn],wordn).
function(iszero,[wordn],bool).
function(eq,[wordn,wordn],bool).
function(leq,[wordn,wordn],bool).
function(geq,[wordn,wordn],bool).

% Generic constants.
%
gen_const(zero,wordn).

%=== Rewrite rules ===
%
%--- Conditional rewrite rules for xterms ---
xtrr([],eq(X,X),1).
xtrr([],iszero(zero),1).

%--- Conditional rewrite rules dealing with zero ---
%
rr([(eq(X,Y),1),(iszero(X),Z)],iszero(Y),Z).
rr([(eq(X,Y),0),(iszero(X),1)],iszero(Y),0).

rr([(eq(X,Y),1),(iszero(Y),Z)],iszero(X),Z).
rr([(eq(X,Y),0),(iszero(Y),1)],iszero(X),0).
```

```
%--- Read-Write rewrite rule ---
% If Addr1 = Adrr2
%then readreg(writereg(Regfile,Addr1,Value), Addr2) -> Value;
%else readreg(writereg(Regfile,Addr1,Value), Addr2) ->
%readreg(Regfile,Addr2);
%
rr([],readreg(writereg(_,A,V),A),V).
rr([(eq(A,A1),1)],readreg(writereg(_,A,V),A1),V).
rr([(eq(A,A1),0)],readreg(writereg(Reg,A,_),A1),readreg(Reg,A1)).
rr([],readreg(writereg(Reg,_,_),A1),readreg(Reg,A1)).
```

B. Execution Flow

In this section, we give the execution flow of the combinational verification and the ASM equivalence checking. The ASM state exploration and safety property checking are special cases of the equivalence checking.




ASMs equivalence checking
Load algebraic spec. file, symbol order file and invariant spec. file
Build component MDGs for invariant condition.
Compute invariant MDG
Load circuit description files, state encoding and manual relation partitioning file
Build component MDGs
Show partitioning strategy
There are errors ?
yes
no
Derive partitioned relations
Build initial state MDG
Reachability analysis
Reachability analysis
Compute new inputs
Compute outputs
Prune outputs (check if outputs satisfy the invariant)
Satisfy ?
no
yes
Compute new reached states
compute the set of frontier states
All states visited ?
yes
no
Compute visited states
Compute the counterexample
Print the results and the statistics
End

C. Error Messages

This list contains most of the error messages.

Errors related with consistency checking and component MDG creation (../applications/utilities/parser_util1.pl & parser_util2.pl):

`Error 101: The target and argument sorts of function X are all concrete. (Then it is a concrete function symbol. In MDGs, we do not consider concrete function symbol as a concrete function can be fully expanded.)`

`Error 102: Function X has no arguments. (Use gen_const instead of function with no arguments.)`

`Error 103: Symbol X declared twice, as Y and Z.`

Y and Z may be functions, constants, etc.

`Error 104: Symbol X declared as Y is an individual constant in enumeration of sort Z.`

Y may be function symbol, generic constants, etc.

`Error 105: X declared as signal and as Y.`

Y may be function symbol, constants, etc.

`Error 106: X declared as signal is an individual constant in enumeration of sort Y.`

`Error 107: state component X in an invariant condition. (Invariant condition is expressed as a combinational circuit.)`

`Error 108: Abstract state variable X comes after its next state variable Y. (It should come before Y.)`

`Error 109: Renaming substitution is not order preserving.`

Refer to section 2.2.

`Error 110: Xs are state variables and next state variables. (state and next state variables should be disjoint.)`

`Error 111: the state variable X is an input. (Inputs and state variables should be disjoint.)`

`Error 112: the next state variable X is an input. (Inputs and next state variables should be disjoint.)`

`Error 113: Possible combinational loop with X.`

We assume the circuit does not contain combinational loop.

`Error 114: Function X is not defined.`

`Error 115: The signals in list Xs do not have the same sort.`

`Error 116: The elements in list Xs are not distinct.`

`Error 117: No st_nxst/2 declaration for state component Y.`

`Error 118: List Xs should be a subset of list Ys.`

`Error 119: list Xs contains duplicate item Y.`

`Error 201: Different components have the same name X.`

`Error 202: Component syntax error: X.`

Refer to section 2.5 for the syntax of predefined component modules.

`Error 203: Sort definition incompatible to bus signals.`

Refer to section 2.5 for bus declaration.

`Error 204: variable X is of abstract sort Y. (In the table declaration, the variables in the condition part must be of concrete sort.)`

`Error 205: the target sort of function symbol X is an abstract sort Y. (In the table declaration, only the cross-function symbol is allowed in the condition part.)`

`Error 206: The table definition for X is nondeterministic.`

There are two rows having the same condition part but different assignments.

Errors related with sequential verification (../applications/verify_seq/):

(../applications/verify_seq/init.pl):

`Error 301: X is not a state variable.`

It is not permitted to give initial values to variables that are not state variables

`Error 302: The state variable and the variable representing the initial state have different sort.`

`Error 303: The state variable and the generic constant have different sort.`

```
Error 304: The initial value X is not an element in the enumeration of sort
Y.
```

(../applications/verify_seq/compile2.pl):

```
Error 311: Partition for X is missing or ill-formed.
```

D. List of files in MDG tools

The MDG tools contain three sub-directories: *documents*, *examples* and *programs*. The document sub-directory contains manuals and algorithms. The example sub-directory provides several illustrative examples for combinational and sequential verification. There is a README file for each example. The program sub-directory includes all the Prolog source files. All files are listed below (directories are in **bold** style).

- **Documents (homeMDG/documents):**

  MDG_TOOL.ps — General introduction of MDG tools.

  MDG_UMAN.ps — MDG tools user's manual.

  MDG_DMAN.ps — MDG tools developer's manual.

  MDG_ALGO1.ps — MDG package algorithms I (basic set).

  MDG_ALGO2.ps — MDG package algorithms II (extended set).

- **Examples (homeMDG/examples):**

  /gcd — GCD.

  /minmax — MinMax.

  /tamarack — Tamarack-3 microprocessor.

  /itc — Island tunnel controller.

- **Programs (homeMDG/programs):**

  /applications (Including files and sub-directories for applications) :

  - make.pl — Top level commands for loading files.
  - **/utilities** (Various utilities) :
    - common.pl — Default algebraic specification.
    - util_appl.pl — Statistics and file search path maintenance.
    - util_parser1.pl — Various utilities for parsing MDG--HDL, including consistency checks.
    - util_parser2.pl — Various utilities for creating basic MDGs and component MDGs.
    - **/verify_comb** (Combinational circuit verification):
      - comb.pl — Module interface.
      - compile_comb.pl — Compiling input-output relations.
      - verify_comb.pl — Combinational circuit verification.
    - **/verify_seq** (Sequential circuit verification) :
      - seq.pl — Module interface.
      - compile_seq1.pl — Circuit compilation (component compilation).
      - compile_seq2.pl — Circuit compilation (deriving transition and output relations).

init.pl	Initial state set generation.
verify_seq.pl	Sequential circuit verification.

/mdgpac1.0 (MDG package) :

mdgpac.pl	Module interface.
abs_xterm.pl	Abstraction of redundant cross-terms.
dbys.pl	Difference-by-subsumption (used by counterexample facility).
disj.pl	Disjunction.
gene.pl	Generalization.
pbys.pl	Pruning-by-subsumption.
relp1.pl	Relational product (part 1), concrete and abstract variable cases.
relp2.pl	Relational product (part 2), cross-term case.
relp3.pl	Relational product (part 3), substitution.
simrelp.pl	Simple relational product.
rew1.pl	Rewriting algorithm 1 (no requirement on node ordering).
rew2.pl	Rewriting algorithm 2 (require proper node ordering).
util_mdgpac.pl	Utilities.

/printing (Utilities for printing MDGs) :

print.pl	Module interface.
util_print.pl	Utilities for printing MDGs.

/rean (Reachability analysis, including counterexample facility) :

rean.pl	Module interface.
ctex.pl	Counterexample facility.
reach_analysis.pl	Reachability analysis.